Andreas Fitzek, BSc.

Development of an ARM TrustZone aware operating system ANDIX OS

Master's Thesis

Graz University of Technology

Institute for Applied Information Processing and Communications
Head: Univ.-Prof.Dipl-Ing.Dr.techn. Reinhard Posch

Supervisor: Dipl-Ing. Daniel Hein
Evaluator: Prof. Roderick Bloem

Graz, April 2014

Abstract

On modern devices all different kind of applications are executed. These include games, browsers, or on-line banking applications. Some of these applications we trust, some we do not. The problem is that both trusted and untrusted applications are executed on the same device and we need to trust our device to protect sensitive informations, like for example our banking credentials. We want to isolate trusted and untrusted applications from one another. Therefore we use domain isolation that separates trusted and untrusted execution environments for applications. We want to enforce domain isolation that not even if it the operating system kernel is compromised, untrusted applications can access sensitive information inside the trusted execution environment.

The ARM TrustZone is a security mechanisms available in many ARM processors. It introduces two states into the processor, a secure and a normal state. This can be used to provide hardware backed domain isolation.

This thesis describes the development of ANDIX OS, an ARM TrustZone aware operating system. It operates in the secure state of the processor. ANDIX OS protects its own memory resources and peripheral devices from access of the normal state. In the normal state an other operating system is running parallel to ANDIX OS. ANDIX OS supports the execution of Linux and Android in the normal state. Linux and Android perform normal system operations, while ANDIX OS provides a trusted execution environment, which is strongly isolated from the normal world operating system.

ANDIX OS allows the development of Trusted Applications, which are executed in the trusted execution environment. ANDIX OS protects the resources of the Trusted Applications against attacks from the normal world. The secure and the normal world communicate via a Remote Procedure Call mechanism.

ANDIX OS is a free and open source ARM TrustZone aware operating system. It can be used for all kinds of TrustZone aware development projects, either academic or commercial. ANDIX OS runs on a software emulator and on a

low cost hardware platform, because we want it to be as easy as possible to start development with ANDIX OS for other parties. We hope that ANDIX OS drives the research in the field of ARM TrustZone aware systems.

Kurzfassung

Moderne Geräte werden für diverse Anwendungen verwendet. Dies können sein, Spiele, Web-browser oder Online Banking Anwendungen. Manchen dieser Anwendungen vertrauen wir, und manchen nicht. Das Problem ist, dass sowohl vertraute als auch nicht vertraute Anwendungen auf derselben Plattform ausgeführt werden. Dabei möchten wir auf unsere Plattform vertrauen können, dass diese sensitive Information, wie beispielsweise Zugangsdaten für Bankanwendungen vor unberechtigtem Zugriff schützt. Wir möchten die Anwendungen von einander isolieren. Dazu möchten wir Domänenisolation verwenden und den Anwendungen unterschiedliche Ausführungsumgebungen anbieten. Diese sollen geschützt sein, selbst, wenn eine Anwendung die volle Kontrolle über den Kernel des Betriebssystems gewinnt.

Die ARM TrustZone ist ein Sicherheitskonzept, welches in vielen ARM Prozessoren verfügbar ist. Dabei wird der Prozessor in zwei Zustände geteilt, einen sicheren und einen normal Zustand. Dies kann dazu verwendet werden, um eine hardwaregestützte Isolation zu gewährleisten.

In dieser Arbeit wird die Entwicklung von ANDIX OS, einem ARM TrustZone Betriebssystem, beschrieben. Das Betriebssystem arbeitet im sicheren Zustand des Prozessors. Mit ANDIX OS können Resourcen wie z.B. Speicher und Peripherigerärte vor Zugriffen aus dem normalen Zustand geschützt werden. Parallel zu ANDIX OS läuft ein weiters Betriebssystem in der normalen Welt. In der normalen Welt unterstützt ANDIX OS Linux und Android als Betriebssystem.

Für ANDIX OS können sogennante Trusted Applications enwickelt werden, welche in der sicheren Welt ausgeführt werden. Die Resourcen dieser Trusted Application werden dabei vor Zugriffen aus der normalen Welt geschützt. Die normale Welt und die sichere Welt kommunizieren mittels eines Remote Procedure Call Mechanismus.

ANDIX OS ist ein freies und quellen offenes ARM TrustZone Betriebssystem. Es kann für diverse ARM TrustZone Projekte, sowohl akademische als auch kommerzielle, verwendet werden. ANDIX OS läuft auf einem Emulator und einem günstigen Entwicklungsboard. Damit wollen wir den Einstieg in die Entwicklung mit ANDIX OS so einfach wie möglich gestalten. Wir hoffen, dass ANDIX OS die Forschung im Bereich der ARM TrustZone vorantreibt.

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, ______________________ ______________________

Date Signature

Eidesstattliche Erklärung[1]

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am ______________________ ______________________

Datum Unterschrift

[1]Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Acknowledgements

The author wishes to thank several people. I would like to thank my partner, Verena, for her love, kindness and support she has shown during the last year it has taken me to finalize this thesis. Furthermore, I would also like to thank my parents for their endless love and support. I would like to thank my supervisors Prof. Roderick Bloem and Dipl-Ing. Daniel Hein as well, for their assistance and guidance with this thesis. Furthermore I would like to thank Johannes Winter for introducing me to the topic as well for the support on the way.

Contents

List of Figures

List of Tables

1. Introduction

1.1. Motivation

Modern devices are used for many activities. Surfing the web, reading emails, playing games, consuming multimedia, office applications, or on-line banking and so on. These different use cases have different needs for security requirements. We want to be able to trust our platform to protect sensitive information, but also use it for untrusted rich applications, like playing games or consuming multimedia. For example your banking application should be protected from corruption by untrusted applications, such as malware.

Modern operating systems use mechanisms to isolate processes from each other. The concept of virtual memory is used to isolate memory of two processes. Virtual memory isolation gives every process its own virtual memory. To access other processes memory, the operating system provides system calls, which perform access control checks. Figure 1.1 shows a simple illustration of the concept.

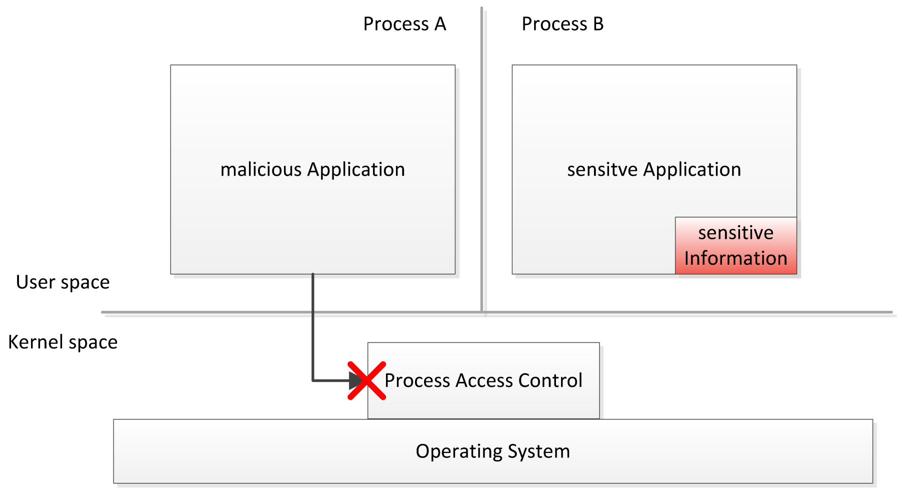

Figure 1.1.: Operating system process isolation

In all modern operating systems security vulnerabilities exist. Malware can use these vulnerabilities to increase their privileges, in order to circumvent access control mechanisms of the operating systems. When sensitive applications hold their sensitive information inside their memory, malware may be able to access these sensitive information, when the access control mechanisms has been circumvented. These vulnerabilities can exist directly in the operating system or in other processes with higher privileges than the untrusted application. How can the sensitive application protect its sensitive information? In the worst case scenario, not at all. If the malware has superuser level access to the system, the sensitive application cannot protect its sensitive information only with software. The sensitive information can be stolen by a malware. Figure 1.2 shows the circumvention of the process isolation of modern operating system via a security vulnerability.

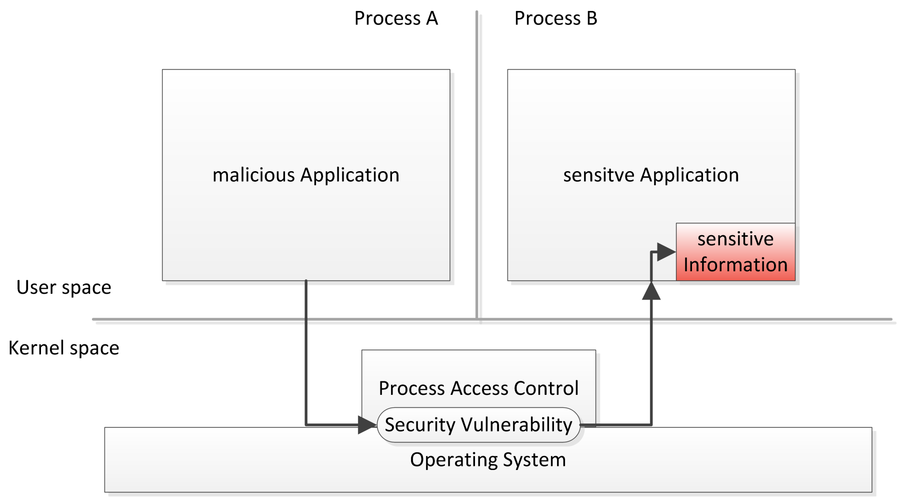

Figure 1.2.: Operating system process isolation circumvented

A system capable of protecting the sensitive information of the sensitive application while allowing the user to still use potential untrusted applications has to have security attributes. We use the attributes of a trusted computer defined by David Grawrock in [Gra09] . These attributes are:

- Isolation of programs,
- Separation of user processes from supervisor processes,
- Long-term protected storage,
- Identification of current configuration,
- A verifiable report of the platform identity and current configuration, and
- Provide a hardware basis for the protections.

Building blocks are associated with attributes. If a system implements a building block associated with an attribute, the system has this attribute. Off-the-shelf operating systems for the ARM platform contain building blocks that implement some of these attributes. Figure 1.3 shows these building blocks with their associated security attributes.

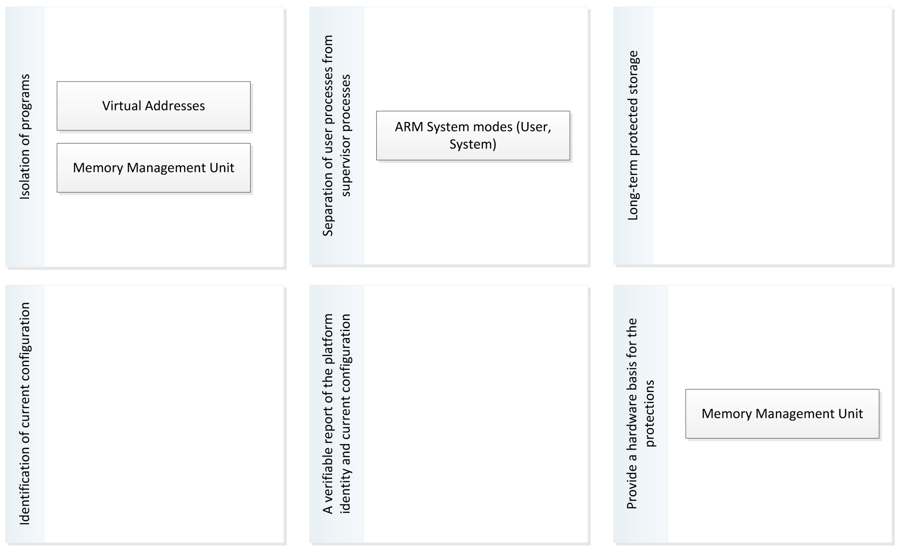
Isolation of programs
Virtual Addresses
Memory Management Unit
Separation of user processes from supervisor processes
ARM System modes (User, System)
Long-term protected storage
Identification of current configuration
A verifiable report of the platform identity and current configuration
Provide a hardware basis for the protections
Memory Management Unit

Figure 1.3.: Security Attributes and building blocks for off the shelf operating systems

In this thesis we build a TrustZone aware operating system called ANDIX OS. During normal operation, an off-the-shelf operating system, like Linux, is executed as rich operating system next to the ANDIX OS. This off-the-shelf operating system is the main operating system seen by the user. ANDIX OS runs in parallel to the off-the-shelf operating system and isolates itself from this operating system. We introduce more building blocks to provide security attributes for the ANDIX OS. We introduce a new layer of isolation, and provide more hardware based building blocks by utilizing the ARM TrustZone (TZ). ANDIX OS provides a hardware isolated execution environment. Therefore this environment is also protected from superuser access in the off the shelf operating system. This gives the user the possibility to execute a variety of low security applications, like multimedia applications or games, and high security applications, like on-line banking or business applications, on the same device. The high security application can utilize the execution environment provided by ANDIX OS to protect their sensitive informations.

1.2. ARM TrustZone

The ARM TrustZone (TZ) is a security concept, that divides the system into two partitions, isolated by hardware mechanisms.

In order to prevent an attack as shown in Figure 1.2, the concept of ARM TZ was developed. When the operating system is TZ aware, it can use the isolation provided by the TZ and the sensitive application is able to hide the sensitive information inside the TZ. Therefore it has to split the application into a Trusted Application and a front end application. The front end application is the similar to the application without TZ support, but a small sensitive part is moved into the Trusted Application. The part that is moved into the Trusted Application should be the only part using the sensitive information. The Trusted Application part of the application should be kept as small as possible to reduce the risk of introducing security vulnerabilities into the Trusted Application code. The concept of the Trusted Computing Base (TCB) applies to the Trusted Application code. The TCB consists of all components that have to be trusted, to trust a system. For our definition of TCB see section 2.1. Via the Remote Procedure Call (RPC) interface, the front end application and the Trusted Application can communicate with each other. The RPC interface should be designed to never hand out the sensitive information itself. It only provides functions to use the information. For example if the sensitive information is a secret cryptographic key, the interface should allow to use the key, but not to extract the key. With this method the application can protect its sensitive information from being extracted by malware. Even if malware has full superuser access to the normal world operating system, the sensitive information cannot be accessed. Figure 1.4 shows this scenario.

1.3. TrustZone-aware Systems

A TrustZone-aware system consists of the TrustZone hardware mechanisms (see Chapter 2.5) and of course software, using these mechanisms. In such a system, two operating systems (OSs) are actually running. One OS is executed in the secure world and the other OS is executed in the normal world. The idea is to keep the secure OS very small and only capable of performing security relevant tasks. The more complex the secure OS is, the more likely it becomes that it contains software security flaws that can be utilize to compromise the system. The secure OS is part of the TCB of the system. The user will only work with the normal world OS, which is not part of the TCB.

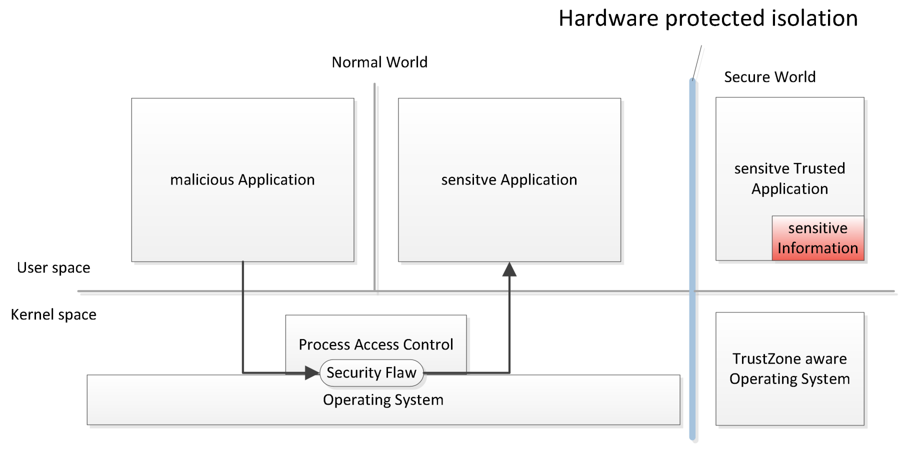

Figure 1.4.: Operating system process isolation with TrustZone backed security

The normal world OS is also called the rich OS, because it implements all kinds of features the user needs to work on the system. This includes a Graphical User Interface (GUI), maybe with 3D accelerations for games, a network stack, multimedia applications and so on. The normal OS is essentially an off-the-shelf OS like Windows, Linux, OS X or Android.

The secure operating system is responsible for security critical operations. Mostly cryptographic material is stored in the TZ and protected by its security mechanisms. The secure OS is part of the TCB of the whole system and therefore should be kept as small as possible. The secure OS also has to host the normal OS. In the boot process of a TrustZone aware system the secure OS boots up first and then the secure OS boots up the normal world operating system. Therefore acting as a boot loader for the normal world.

Both worlds need to communicate with each other, to implement a RPC mechanisms. The world communication is designed to work similar to an operating system call. A special operation mode exists in the TZ, which allows the current Central processing unit (CPU) to switch its state. This mode is called monitor mode and is implemented by the ANDIX OS. In this mode the system can switch between the two worlds. Both worlds can access this monitor mode by calling a trapdoor instruction called secure Monitor Call (SMC). This instruction works like to a software interrupt, which traps the user space into the kernel space, the SMC traps the kernel mode into the monitor mode. Information can be transferred to the monitor mode, by setting up the CPU registers before executing the SMC instruction. These registers can then be read in the monitor mode.

1.4. ANDIX Operating System

The secure operating system developed in this thesis is called ANDIX OS. It is a multitasking, non pre-emptive operating system and currently supports the iMX 53 Quick Start Board (iMX53QSB)[1] as a hardware platform and the Qemu TrustZone[2][3] implementation of Johannes Winter as an emulated hardware platform. ANDIX OS supports Linux and Android on both supported platforms as normal world operating systems. Android is still very unstable on the iMX53QSB and has very poor performance on the Qemu TrustZone. This means that development of TZ applications for Android based on ANDIX OS requires a some patients.

GlobalPlatform[4] is an organisation that provides standardized Application Programming Interfaces (APIs) for Trusted Execution Environment (TEE) runtime environments and APIss for TEE communications. Their mission statement according to their website on 1st of April 2014 is:

> GlobalPlatform works across industries to identify, develop and publish specifications which facilitate the secure and interoperable deployment and management of multiple embedded applications on secure chip technology. GlobalPlatform Specifications enable trusted end-to-end solutions which serve multiple actors and support several business models. [5]

We discovered their specifications during the development of ANDIX OS and decided to implement these APIss to provide source code compatibility with other TEEs.

The ANDIX OS consists of many components, which have to play together to provide a stable and secure TrustZone aware operating system.

The heart of every operating system is the kernel. The ANDIX kernel operates in the secure world in kernel mode. It is responsible for providing a secure execution environment to the Trusted Applications running in the secure world in user mode. The kernel has to set up the TrustZone Address Space Controller (TZASC) to protect the secure world memory region and to detect unauthorized access to these regions. If an attempt to access a secured memory region

[1] http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=IMX53QSB
[2] http://www.qemu.org
[3] https://github.com/jowinter/qemu-trustzone
[4] http://www.globalplatform.org/
[5] **http://www.globalplatform.org/aboutusmission.asp**.

is detected, ANDIX OS will take control of the system and stop it. The ANDIX OS kernel also has to be capable to secure certain interrupts and therefore take control over certain peripherals. It also provides process isolation via virtual memory to the Trusted Applications operating in user mode. ANDIX OS also implements a generic RPC mechanism to provide communication between the secure user space and the normal world.

An operating system is not just the kernel. It also consists of user space libraries, providing standard functions and system calls. In ANDIX OS the user space libraries include an adapted version of the newlib standard C library and the TropicSSL library providing cryptographic functionality. Upon these user space functions we implemented GlobalPlatforms API to provide interoperability to an other TEE [Glo11] . To be interoperable with other TEE Trusted Applications should only use functions defined in the standardized API. The TEE defines functions for trusted long term storage, cryptography, timing and arithmetic. These are the building blocks needed to implement a Trusted Application.

With these components a normal operating system is complete, but a TZ aware system is more complex. It also has to provide a way to allow normal world applications to talk to a Trusted Application. ANDIX OS implements this communication channel as a kernel module for Linux. The communication is based on SMCs. A SMC can only be executed in kernel mode. Since the interface is similar to a system call interface, the kernel module is similar to a user space system call implementation. The commands are issued from the normal world user space, will be forwarded by the kernel module to the ANDIX OS. The kernel module exposes an interface via a pseudo character device. Via this pseudo character device a user space library was developed, which abstracts the communication interface to a standardized API from GlobalPlatform [Glo10a] . This library can be used by front end applications, to communicate with their Trusted Application. The Trusted Application component runs in the secure world in user mode, protected by ANDIX OS.

ANDIX OS currently supports the Freescales iMX53 Quick Start Board and the QEMU Trustzone implementation developed by Johannes Winter ([6]). As rich operating systems, ANDIX OS supports Linux and Android.

[6] `https://github.com/jowinter/qemu-trustzone`

2. Preliminaries

This chapter provides basic knowledge about the ARM system and about the ARM TrustZone (TZ). It also contains definitions of terms used later in the thesis and introduces the Domain Isolation Concept. We presume the reader has basic knowledge of common operating system (OS) concepts as in [Tan07] .

2.1. Definitions

Now following are terms and their definition in the context of this thesis:

user/kernel space In modern operating systems the kernel and user applications are executed in different operating modes of the Central processing unit (CPU). This isolates the kernel and user application from one another and protects the system from faulty and malicious applications.

system call A system call is a mechanism for applications running in the user space to utilize functions that are only available in the kernel space. The kernel exposes a system call interface and allows application to use these functions.

interrupt Interrupts the processor in its current execution and notifies the processor about the occurred event, by jumping to the handler address.

System-on-a-Chip A System on Chip (SoC) is a system where its components are integrated into one circuit. Modern smart phones are also SoCs.

secure world The ARM TrustZone partitions the SoC into two partitions. The partition of the SoC which is protected by the ARM TrustZone is called secure world

normal world The ARM TrustZone partitions the SoC into two partitions. The partition of the SoC which is not the secure world is called normal world

rich operating system A rich operating system is an operating system capable of providing a rich user interface. In this thesis we will refer to a rich operating system as the operating system running in the normal world and providing the normal user interface.

secure In this thesis we define secure, when the confidentiality and integrity of data is ensured and cannot be circumvented only by software. For example a secure environment is secure if the environment protects the data inside the environment against the outside of the environment and that the data inside the environment cannot be altered from outside the environment without detection, under the premise, that no hardware changes where made.

Trusted Execution Environment A Trusted Execution Environment (TEE) is an execution environment for software, that protects the memory of the running software and the Input/Output (IO) functionality of the software.

Trusted Application A Trusted Application is a piece of software, that can be executed inside TEE. It utilizes the runtime environment inside the TEE. Often a Trusted Application is a small piece of software which performs only security critical parts of a bigger application. The bigger application out sources the security critical parts into a Trusted Application to protect sensitive information.

Trusted Computing Base The Trusted Computing Base (TCB) consists of all components, hardware, firmware and software that have to be trusted to trust the system. The TCB should be kept as small as possible, because this reduces the risk of security flaws, which may evolve into vulnerabilities.

2.2. Domain Isolation

Modern computer systems have to provide multiple functions. These include multimedia playback, video gaming, office applications and on-line banking. These different functions need different security levels. One way to provide different security levels for the applications is to isolated them from one another. We will call this domain isolation.

The following overview on domain isolation techniques in this section is based on the survey of Arun Viswanathan and B.C. Neuman [Aru] . Domain isolation can be achieved by multiple techniques. These techniques can be categorized into five main categories:

Language-based Language-based isolation utilizes programming language semantics like type systems and certifying compilers, to ensure the application does not leave its domain. For more information on Language-based isolation see [SMH01]

Sandbox-based A sandbox provides an execution environment that tries to limit the possibilities of the application inside this environment. A sandboxed application should not be able to exit this sandbox and only use functions authorized by the sandbox. For more information on Sandbox-based isolation see [Wah+93]

Virtual Machine-based Virtual Machine-based isolation provides a full runtime environment to the domain to be isolated. This environment is managed to provide access control to system resources. For more information on Virtual Machine-based isolation see [SN05]

Kernel-based Kernel-based isolation trusts the operating system kernel to provide isolation between domains. Commonly these domains are processes. In this concept the operating system kernel is part of the TCB. The smaller the TCB the better, because there is less code that has to be trusted, and a smaller code base is less likely to contain security vulnerabilities. One form of Kernel-based isolation is called Microkernel [Acc+86] . In a Microkernel only the most important systems are implement in the kernel. Only a virtual memory system, basic Inter-Process-Communication (IPC) mechanisms and scheduling are implemented in the kernel code. System drivers, file systems and network stacks are implemented as user space processes in the system. This reduces the kernel code base and therefore provides a smaller TCB. An even more reduced kernel system is called Exokernel [Kaa+97] . An Exokernel manages the access to hardware resources to prevent simultaneous access to hardware components, or to prevent unauthorized access to hardware components. The processes executing on top of an Exokernel will have to know how to access and use the hardware components. This reduces the kernel code base basically to an access control layer for the hardware components. This reduces the kernel code size and therefore the TCB. The XEN Hypervisor[1] is one example of an Exokernel.

Hardware-based Hardware-based isolation utilizes hardware functionality of the system to provide domain isolation.

One example of Hardware-based isolation is the Memory Management Unit (MMU). The MMU allows the system to provide virtual address spaces to the different domains and therefore can be utilized for memory isolation.

Another example of Hardware-based isolation is the Input Output Memory Management Unit (IOMMU). An IOMMU can be used to isolate devices for certain domains.

Both the MMU and the IOMMU are used by operating system kernels

[1]http://www.xenproject.org/

to provide domain isolation for their processes. This implies that the operating system kernel is part of the TCB, because if the kernel cannot be trusted, neither the MMU isolation nor the IOMMU can be trusted, since the kernel is responsible for utilizing these hardware devices. Muli Ben-yehuda et al. provide a good introduction into the concept of IOMMU in [Ben+06] .

ARMs TrustZone also provides Hardware-based isolation on ARM systems. It divides the system into two parts one secure and one non-secure part. In this thesis we develop an operating system to be executed in the secure part of the ARM TrustZone. It should only contain the most important systems, to keep the TCB as small as possible. To provide Hardware-based isolation, the system developed in this thesis provides a trusted execution environment, and allows application developers to divide their applications into two domains, with a communication interface.

We will give a more detailed description of the relevant domain isolation techniques for ANDIX OS later in this chapter.

2.3. GlobalPlatform Trusted Execution Environment

ANDIX OS implements the GlobalPlatform Application Programming Interfaces (APIs) to provide source code compatibility with other TEEs. In this chapter we want to give an overview about the GlobalPlatform APIs.

2.3.1. Introduction

The GlobalPlatform TEE specifications define a Client Application and a Trusted Application. The Client Application is not executed inside the TEE and the Trusted Application is executed inside the TEE. The Client Application can use the Trusted Execution Environment Client (TEEC) API to communicate with the Trusted Application inside the TEE. The Trusted Application should only use the TEE API functions. If the Trusted Application only uses these functions, the Trusted Applications are source code compatible with other TEEs that implement the GlobalPlatform specifications [Glo10b] . Figure 2.1 shows how the GlobalPlatform API specifications are used. By implementing the specifications, Client Applications and Trusted Applications only have to be developed once and can be build for different TEEs.

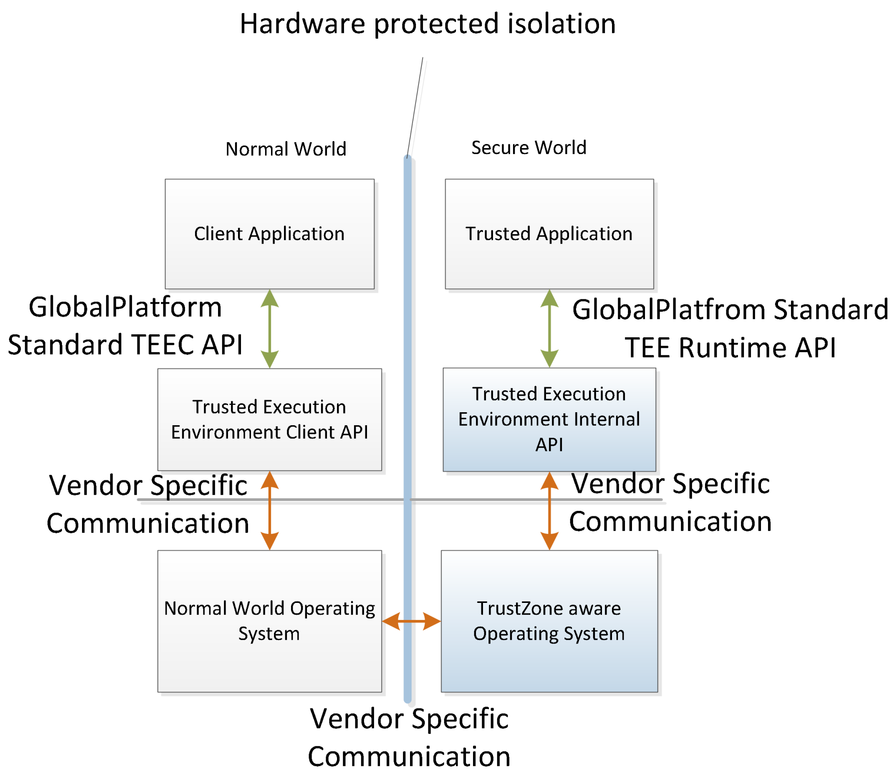
Hardware protected isolation
Normal World
Secure World
Client Application
Trusted Application
GlobalPlatform Standard TEEC API
GlobalPlatfrom Standard TEE Runtime API
Trusted Execution Environment Client API
Trusted Execution Environment Internal API
Vendor Specific Communication
Vendor Specific Communication
Normal World Operating System
TrustZone aware Operating System
Vendor Specific Communication

Figure 2.1.: GlobalPlatform API Specification Concept

2.3.2. Trusted Execution Environment Client

The TEEC is a standardized API defined by GlobalPlatform to allow communication between the Client Application and the Trusted Application. The first step for a Client Application is to connect to a TEE. This allows a platform to support multiple TEEs. To connect to a TEE the Client Application calls the TEEC_InitializeContext function. This function initializes a TEEC_-Context. The TEEC_Context is a handle to a specific TEE. In a TEE there can be multiple Trusted Applications. When connected to a TEE a Client Application can connect to a Trusted Application. To do this the Client Application calls the TEEC_OpenSession. This opens a session to the Trusted Application. The session is identified in the Client Application with a TEEC_Session. When the Client Application is connected to a Trusted Application, the Client Application can invoke commands in the Trusted Application, by calling the TEEC_InvokeCommand. The TEEC_InvokeCommand functions allows to send and receive memory reference to and from the Trusted Application. Listing A.1 shows a sample TEEC application code. It initializes a TEEC_Context, connects to TEEC_Session and issues a command to the Trusted Application [Glo10a]
.

2.3.3. Trusted Execution Environment Internal API

The Trusted Execution Environment Internal API is a standardized set of functions and data types. A Trusted Application should only use the functions and data types defined in the API. If the Trusted Application only uses these functions and data type it will be source code compatible with other TEEs that implement GlobalPlatforms TEE specifications.

A Trusted Application has a different life cycle than a normal application. A normal application has one main entry point usually the main function. A Trusted Application has to implement five functions that get called by the runtime environment depending on the current state of the Trusted Application. The first function to be called is the TA_CreateEntryPoint. This function is called exactly one time when the Trusted Application is first started. A clean up function the TA_DestroyEntryPoint of the Trusted Application is called exactly one time when the Trusted Application is closed again. In terms of object orientation the Trusted Application can be seen as one class instance where TA-_CreateEntryPoint is the constructor and TA_DestroyEntryPoint the destructor. When a Client Application opens or closes a session to a Trusted Application the TA_OpenSessionEntryPoint or TA_CloseSessionEntryPoint of the Trusted

Application is called by the runtime environment. The Trusted Application receives a session object for each session, which can be used to store arbitrary data in a session context. The TA_InvokeCommandEntryPoint of a Trusted Application is called for each command invocation from a Client Application. Instead of one main entry point like a normal application a Trusted Application contain five entry points. Listing A.2 shows a sample Trusted Application, that is compatible with the Client Application code in listing A.1 [Glo11] .

2.4. ARM Architecture Basics

This section gives an introduction into the ARM (ARM) v7 architecture. For a more detailed explanation, please see [ARM12] .

An ARM processor can operate in different modes. The current mode determines the privilege level of the current operation. In the ARM architecture up to three privilege levels are available. Table 2.1 show which ARM mode has which privilege level.

Table 2.1.: ARM Modes and privilege levels

privilege level	ARM Mode
PL0	User
PL1	System
PL1	Supervisor
PL1	FIQ
PL1	IRQ
PL1	Abort
PL1	Monitor
PL2	Hyp

The monitor mode only exists in SoCs with security extensions. Every mode has it own fixed registers in the ARM CPU. These registers are used to hold the current stack pointer and the link register. The link register holds the current return address of a function.

The privilege levels determine the current access rights to some features of the system. PL0 is the privilege level for unprivileged executions. PL1 gives access to all features of the system except virtualization features. PL2 gives access to all features of the system including the virtualization features. PL0 usually is used for user applications. PL1 usually is used by operating system kernels and PL2 is used for hypervisors like Xen[2].

An exception in ARM is an interruption of the current execution, which gives the operating system the chance to react on an event. Such an exception can be an invalid memory access, also call a data abort, an interrupt etc. Depending on the type of the exception, the processor enters a different mode. When an exception occurs, the processor changes to a mode responsible for handling the event and transfers the control to a predefined entry point. These entry points are structured as a vector. The base address of the vector is registered at the processor and the entry points are defined as offsets to the registered vector. The first operation done by a handler in the vector is to save the current context. The current context consists of the common CPU registers. These registers are shared between the different modes and have to be restored to the values they had before the exception occurred, so the interrupted application can resume the normal execution after the system handled the exception.

An interrupt in ARM is modelled as an exception. If an interrupt occurs, the current execution is stopped, the mode is set the IRQ and the control is handed over the interrupt handler entry point in the vector. ARM also supports Fast Interrupt Request (FIQ)s. These exceptions are basically the same as an interrupt, but in the FIQ mode there are more dedicated registers. These dedicated registers are only available in the FIQ mode. Because the FIQ mode has more dedicated registers, a context switch to this mode is faster. In a normal context switch all common registers have to be saved, but since the FIQ mode only uses less common registers only these have to be saved.

When an invalid access to memory happens, an exception occurs and the CPU changes to the Abort mode. In the Abort mode, the system can investigate the system and determine which error occurred. It is often possible for the system to recover from such an exception.

2.5. ARM TrustZone

2.5.1. Introduction

ARM's TrustZone is a set of hardware backed security mechanisms. It is integrated deep into the SoC infrastructure and the processor core and provides

[2]http://www.xenproject.org/

security mechanisms in the ARM processor, the bus system and the system peripherals. This deep integration allows a wide range of security systems. [ARM09]

The ARM TrustZone introduces two states into the processor. These states are the secure state and the normal state. In the normal state consists of all modes and privilege levels from table 2.1 expect the Monitor mode, are available. In the secure state all modes and privilege levels from table 2.1 expect the Hyp mode and PL2, are available. The idea is to have two partitions in the system. The partition in the secure state can utilize the TZ mechanisms to isolate itself from the partition in the normal state. We use the ARM TZ to run two OSs. One OS is used as a secure environment, the other is the normal operating system, which enables the system specific tasks. For example on a smart phone, the normal OS could be Android or IOS. In the secure world a specialized OS will run as the secure OS.

The secure world is a privileged state of operation of the processor. It gives access to additional control registers of the ARM processor. The secure and the normal state contain different modes similar, but orthogonal to the ring architecture (see section 2.4). So that in each state a kernel and a user space are be implemented. Figure 2.2 illustrates the different system partitions. One special mode is the Monitor mode. In the Monitor mode is shared between the two states and the processor can switch between the two states when in Monitor mode. The monitor mode is used to communicate between the two states. Software in both states can issue secure Monitor Call (SMC). This is a software interrupt that traps into the Monitor mode. The software running in monitor mode decides how to handle the SMC.

The Secure Configuration Register (SCR) is one of these protected registers. It allows the secure world to control the behaviour of some normal world operations. Through the SCR the secure world is able to control the abort and interrupt behaviour of the system. The secure world can use the SCR to intercept all Interrupt Request (IRQ)s and FIQs and redirect external aborts to the secure world. [ARM07]

2.5.2. Memory Isolation

Dividing the processor into two states is not enough to provide hardware security. The secure world also needs a way to protect its system memory. To achieve this, there is a physical line on the system bus signalling the current state of the processor to all peripheral devices. The memory controller has to

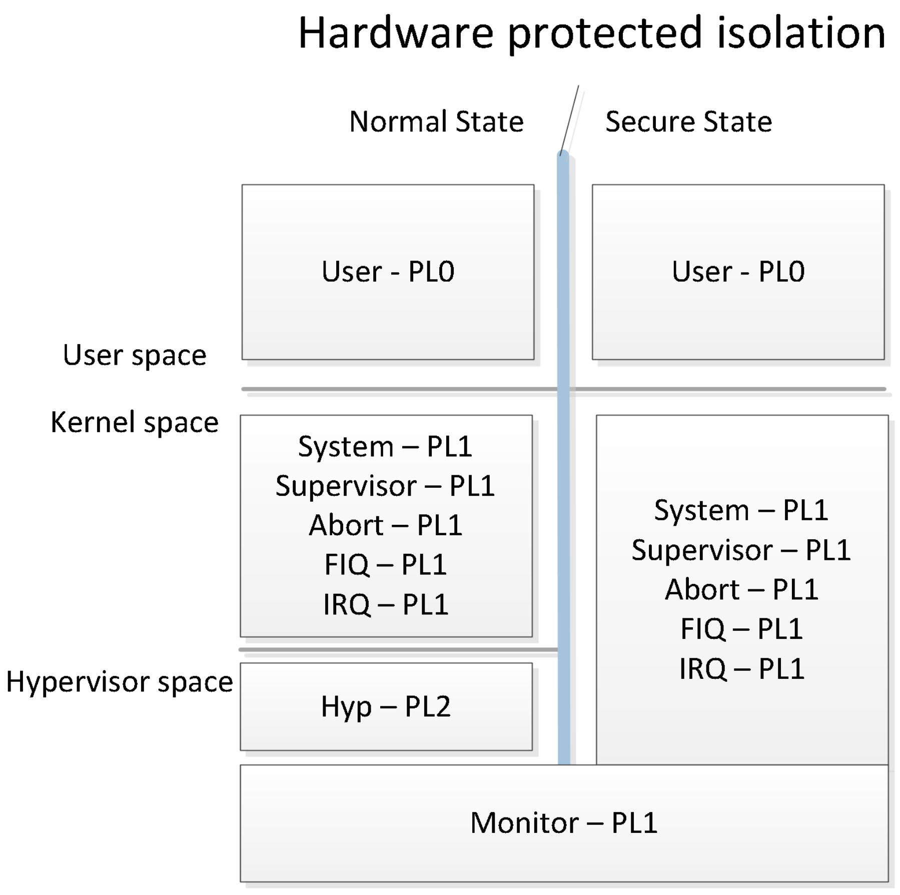

Figure 2.2.: ARM TrustZone paritions

be aware of the TrustZone nature of the system. These memory controllers are commonly called TrustZone Address Space Controller (TZASC). When the processor is in the secure state, the memory controller is used to allocate protected memory regions. If such a region is allocated and the processor tries to access a memory address in the region while the bus signals normal state, the memory controller will issue an external data abort to the processor. The value of the SCR determines if this abort will be handled by the secure world. With this mechanism, the secure world is able to protect its memory and secret values during computations [ARM09] .

2.5.3. Input/Output Isolation

The ARM TZ is integrated deeply in the whole SoC. This allows the TrustZone secure usage of peripheral devices. This is accomplished in a similar way to the TrustZone memory protection mechanisms. The current security state is signalled through the system bus to all peripheral components. The same way a TZASC is aware of the security state, a TrustZone aware interrupt controller can be used to protect only certain interrupts from peripheral devices. Furthermore, the bus controller implements an access control for master and slave peripherals on the system bus. This controller can be reprogrammed in the secure state. With this controller master components can be defined as secure or non-secure on the bus. For slave components the controller assigns the access control rights, for secure and non-secure states. By securing a peripheral device with the bus controller, normal world components cannot access the peripheral any more. This allows the TrustZone aware system to secure certain peripherals in order to provide real end to end security. One example would be to secure the image processor and the touch screen input of a smart phone to provide a secure PIN entry to the user [ARM09] .

2.5.4. Processor

When the processor is in the secure state, it can lock down certain functions to protect the integrity of the system. The core can be configured to trap all external data aborts to the monitor mode. Unauthorized access to secured memory regions or secured devices are signalled as external data aborts. When an external data abort occurs the system state is examined to find the exact source and target of the abort. With this information counter measures against a potential security breach can be performed. ARM suggests to use FIQs in the secure state. When FIQs are used in the secure state, FIQs are not available in the normal state. The ARM core can be configured to disable the FIQ bit in the Current Program Status Register (CPSR) for the normal state. This means that if the normal state tries to activate FIQs, by setting the FIQ bit to one in the CPSR, it has no effect on the system. There also exists a special register which can be read by software, which indicates the current state of the system, meaning if it is secure or in the normal state.

2.6. High Assurance Boot

We can only provide a secure environment if the SoC can ensure that the correct software is loaded. This is called load-time integrity and can be enforced with a High Assurance Boot (HAB).

To establish a secure environment, the SoC starts in the secure state and the secure software is executed. This software can use the TrustZone infrastructure to build a secure environment. When the secure software is booted and secured the resources it needs, it will prepare the normal OS, switch the processor to the normal state and transfer the execution to the normal OS [ARM09] .

Figure 2.3 shows the boot process of an ARM TZ aware system. After the device is powered on, a first stage boot loader is loaded from read only memory. This first stage boot loader is the ROM SoC Bootloader in figure 2.3. This boot loader tries to load the secure world OS from a known location. This could be a memory card, on board flash memory or a network device. The secure world OS is digitally signed by the device manufacturer. This signature is verified by the first stage boot loader. It uses a public key for the signature verification, stored in a read only memory on the SoC. It is essential for the security in terms of origin integrity that this key can not be modified. The mechanisms for keeping this key secure depend on the SoC. We can only trust that this key is not modified. When the signature can be verified successfully, the secure world OS is executed. The secure world OS boots up and than verifies and executes the normal world boot loader in the normal world. This again loads and executes the normal world OS. These process description is an abstraction of a real world secure boot process. The process details may differ from platform to platform.

In ANDIX OS we did not use a HAB, because we only had a limited number of development platforms available and we could not afford to brick one of these platforms. To do a HAB a public key has to be stored into a read only memory

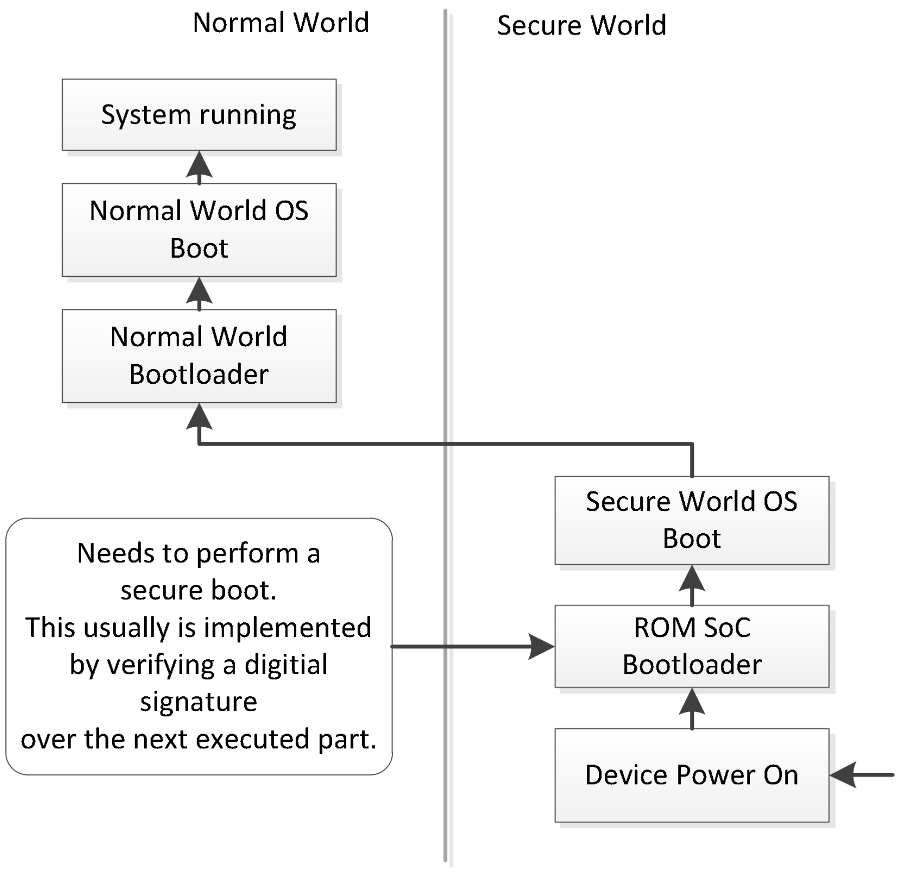

Figure 2.3.: ARM TrustZone boot sequence

on the SoC. This operation is not reversible and if performed wrong may brick the board. In [Fre] the secure boot process of our development platform is described.

3. Related Work

3.1. Isolation for Security

This section gives an introduction into relevant domain isolation techniques for targeted on improving security.

3.1.1. Sandbox-based Isolation

Sandbox-based isolation separates the domains, by keeping the application inside the sandbox domain. This can be enforced by access control lists, by manipulating the executing code to ensure it cannot jump out of its own code segment, or by limiting the access to the systems Application Programming Interface (API).

Sandboxes keep applications inside itself and keeps them from accessing other system resources. In ANDIX OS we use a different approach for isolation. We do not keep potential malicious applications inside a sandbox, but protect sensitive parts of applications from access from outside.

One example of the sandbox-based isolation would be the Dune sandbox use case.

3.1.1.1. Dune

Dune enables user space applications to access privileged Central processing unit (CPU) features. Dune is currently implemented as a 64-bit x86 Linux kernel module. It utilizes virtualization extensions of the x86 platform to build a virtual execution environment for user space applications. This is the same way a hypervisor would act for a virtual guest machine. This virtual execution environment allows the application to operate in kernel execution mode on the CPU, but keeps the host kernel protected by the virtualization extensions. System calls to the real kernel can be issued via a VMCALL instruction. The

VMCALL instruction acts like a system call instruction and trap the execution back to the hypervisor. In the case of Dune the real kernel. [Bel+12]

One use case of Dune is to execute untrusted code with sandbox isolation. The sandbox runtime is executed via Dune in a virtual execution environment. This means the sandbox runtime runs in the kernel execution mode. The untrusted application is executed from the sandbox runtime in the user execution mode. This way the sandbox can utilize a hardware based isolation to isolate the untrusted application, the same way a kernel isolates itself from the user space [Bel+12] .

In contrast to Dune, ANDIX OS provides a Trusted Execution Environment (TEE) for sensitive applications and not sandboxed environment for untrusted applications. This gives an advantage, because an application can take active action and utilize the TEE from ANDIX OS. It does not have to rely on the user, to sandbox all untrusted applications.

3.1.2. Virtual Machine-based Isolation

Virtual Machine-based isolation provides a managed execution environment for the application. This execution environment can perform checks on access of system resources to keep the domain isolated. There exists different types of Virtual Machine-based isolation.

The execution environment may define its own instruction set. The execution environment then translates its own instruction set to the instruction set understood by the hardware. During this translation the environment can ensure that the application does not leave the domain boundaries. Example for this isolation technique is the Java virtual machine and the Common Language Runtime.

We are also currently working on providing a Mono runtime[1] for the ANDIX OS user space. The Mono runtime is an open source implementation of a Common Language Runtime. This would improve the isolation between two running Trusted Applications and the isolation between the ANDIX OS user space and the ANDIX OS kernel space.

The execution environment may also simulate a full hardware platform, for executing a complete operating system inside the domain. Access to simulated memory or simulated devices is translated to the physical memory and real

[1] `http://www.mono-project.com/Main_Page`

devices. During this translation the execution environment can enforce domain boundaries. Examples of this isolation technique are XEN[2] and KVM[3]. These execution environments may even simulate a different instruction set. Furthermore these execution environments may utilize hardware virtualization functions to increase performance of the simulated environments.

3.1.3. Next-Generation Secure Computing Base

The Next-Generation Secure Computing Base (NGSCB) is a platform architecture developed by Microsoft. It proposes a split in the operating system into an untrusted mode and a trusted mode. In the untrusted mode the general purpose applications and kernel will operate. In the trusted mode a special kernel part of the operating system called Nexus, will operate. The Nexus operating system depends on curtain memory feature of CPUs. This memory feature is CPU vendor specific, but provides memory protection even against the operating system. Intel implements this feature in its Trusted Execution Technology (TXT). Security sensitive applications can operate on top of the Nexus part and therefore in a isolated environment. [Mic03]

The concept of NGSCB is very similar to ARMs TrustZone (TZ) concept. In Microsofts NGSCB concept the Nexus part is the equivalent to ANDIX OS.

3.1.4. Trusted Platform Module based Isolation

We take a look at three operating systems, which achieve domain isolation. These operating systems are Terra, acTvSM and Nexus. Terra was the first concept of verifying itself and using virtualization for domain isolation. All three operating systems base their security on ensuring their own integrity using a Trusted Platform Module (TPM). If their implementation is correct, domain isolation is achieved. ANDIX OS runs next to the rich operating system, and isolates itself from the rich operating system, via hardware backed mechanisms provided by the TZ. Nexus and acTvSM both do not explicitly isolate them self from the rich operating system by hardware backed mechanisms. acTvSM has to trust the used virtualization mechanisms, to isolate itself. Nexus acts as rich operating system itself.

[2]Xen Hypervisor (`http://www.xen.org/`)

[3]Kernel-based Virtual Machine (`http://www.linux-kvm.org/`)

3.1.4.1. Terra

Terra is a platform that provides isolated execution environments for sensitive applications. It achieves this using a Trusted Virtual Machine Monitor (TVMM). This TVMM partitions the system into isolated virtual machines. Terra knows two types of virtual machines, open-box machines and closed-box machines. Open-box machines are common general purpose machines. Closed-box machines are isolated environments. Closed-box machines provide hardware protection for memory and cryptographic protection of the long time storage. Terra uses a TPM to protect its own integrity as well as the integrity of closed-box machines. Only if the integrity of the TVMM and the integrity of the closed-box machine are ensured by the TPM, the TPM allows access to the protected resources [Gar+03] .

In contrast to Terra, ANDIX OS does not allow multiple isolated execution environments. Terra uses virtualization mechanisms to provide domain isolation between the different execution environments.

3.1.4.2. acTvSM

acTvSM is a Linux based platform, that provides security-oriented virtualization. It implements the same concept as presented by Terra. acTvSM uses a TPM to ensure the integrity of its code and configuration. It uses the sealing mechanisms of a TPM to enforce an exact state of its trusted kernel code and its main disk partition. When the base system is booted up, it provides a virtualization platform with Quick EMUlator (QEMU) and the Kernel-based Virtual Machine (KVM) to run different isolated environments. These environments are isolated by the platforms virtualization architecture. Each of these environments consists of a full virtual machine. [TPG11]

In contrast to ANDIX OS, acTvSM is based on the x86 architecture and not on the ARM (ARM) architecture. Also acTvSM can provide multiple isolated environments for different applications. ANDIX OS has the advantage, that it makes no assumption about the normal world operating system and operates independent from it. AcTvSM has to confirm the integrity of all of its environments and their configuration. Whenever the configuration changes, the whole system has to be resealed, this operation takes some time. This renders acTvSM interesting for server environments, where the configuration does not change frequently, but not practical for every day multi purpose devices. ANDIX OS can provide a trusted execution environment, without the need to verify the full system state of the normal world operating system.

3.1.4.3. Logical Attestation with Nexus

The Nexus operating system provides an access control policy called logical attestation. It lifts the TPMs hash-based attestation to a logically based attestation. The kernel of Nexus implements access control facilities based on logical formulas. The kernel itself uses a TPM to ensure it own integrity and therefore the integrity of its access control facilities. In the Nexus operating system each process has multiple labels. These labels are logical statements, and called credentials. In the Nexus operating system resources are protected by goal formulas. These goal formulas are logical formulas and can be set by the owner of the resource. When a process or a user wants to access a resource it has to construct a logical proof with the labels and logical statements associated with him, to deduct the goal formulas for the resource. The access control in the Nexus kernel will just check if the proof is correct and grants or prohibits access to the resource. [Sir+11]

In contrast to ANDIX OS Nexus is based on the x86 architecture and not on the ARM architecture. Nexus provides a logic based access control system for platform resources, like memory, network, or Inter-Process-Communication (IPC). This enables Nexus to provide a fine grained access control mechanism.

3.2. Isolation for Stability

A new isolation method is implemented in VirtuOS (see [NB13]). VirtuOS is based on XEN[4] and utilizes hardware based isolations like Input Output Memory Management Unit (IOMMU) and Memory Management Unit (MMU). VirtuOS is not designed to isolate domains from on another for security reasons, but for stability reasons. It splits the kernel into service domains. Each of these service domains runs in a different virtual environment. This reduces the impact of kernel faults. In classical operating systems kernel faults often crash to whole system. In VirtuOS, if there is a faulty device driver or a other faulty implementation in one of the service domains, the system can recover, by restarting the service domain [NB13] .

The service domain isolation of VirtuOS also increase the security of the system. Security flaws in device drivers inside a service domain does not necessary compromise the integrity of the whole system, because the attacker may be able to execute code as kernel, but just inside a special service domain.

[4]Xen Hypervisor (`http://www.xen.org/`)

ANDIX OS utilizes hardware based isolation not for stability, but to provide a TEE, so to increase the security of the system.

3.3. Information Flow Control

3.3.1. Introduction

Information flow control allows tracks data through systems, by attaching labels to data objects. With this labels the information flow control can categorize the data objects. The access decisions are based on the label of the information. This can be used to isolate sensitive information stored in sensitive applications on the platform. Information flow control enables complex control policies to be enforced.

3.3.2. HiStar and LoStar

Zeldovich et al. developed HiStar[5] in [Zel+06] . HiStar is an operating system that performs information flow control on data communications. In the case of HiStar, the kernel takes care of enforcing the information flow control. HiStar enforces information flow through long term storage, by labeling data stored to the hard disk. It also enforces information flow to network interfaces. If HiStars kernel is compromised, the information flow control can be circumvented. To overcome this limitation, Zeldovich et al. propose in [Zel+08] a new hardware architecture called Loki, that allows tagging of physical memory pages. Loki is based on the Sparc architecture. The Loki architecture contains a security monitor. The security monitor is a more privileged processor mode and is used to handle illegal access to tagged physical memory and to set up the information flow control policies for the tag values. On top of the Loki architecture, Zeldovich et al. developed a operating system called LoStar, which is a port of the operating system HiStar to the Loki architecture. LoStar consists of the LoStar kernel and the LoStar security monitor. LoStar uses the hardware memory tagging feature of Loki, to move the enforcement of information flow control to hardware. The labels of data objects in LoStar are implemented by hardware tagging physical memory. The LoStar security monitor is a software, that operates in the Loki security monitor and handles illegal access and sets up information flow control policies. The LoStar kernel handles the common

[5]http://www.scs.stanford.edu/histar/

operating system functionalities. If the LoStar kernel is compromised, the ability to enforce information flow control is not lost, as long as the LoStar security monitor is not compromised.

LoStar supports hardware backed information flow control, which can be seen as domain isolation. It supports complex information flow policy, but cannot currently isolation specific devices for isolated Input/Output (IO) like ANDIX OS.

3.4. ARM TrustZone Aware Operating Systems

3.4.1. Trustonic

The company Trustonic developed a TEE called <t$^{\text{TM}}$-base. <t$^{\text{TM}}$-base is closed source and we have no knowledge about the implemented APIs or the available hardware platforms. <t$^{\text{TM}}$-base seems to be a professional solution for productive TZ based applications. [6]

3.4.2. Open Virtualization

Open Virtualization is the first free and open source operating system that implemented TZ. It comes in two licenses one open source, under the GNU General Public License (GPL), and one commercial license. The commercial product is called SierraTEE. In the open source implementation many important features are missing, like user space task isolation, kernel and user space separation, multitasking, dynamic application loading, secure boot and a POSIX compliant libc.

Open Virtualization only supports the Versatile Express board and Emulation Baseboard. The Versatile Express is very hard to order and also very expensive. The Emulation Baseboard involves also high costs and high effort to retrieve the real view emulation software. [7]

The functionality on the open source implementation is very limited. We wanted to have a TrustZone aware operating system available for a real hardware platform, which is much easier affordable.

[6] `http://www.trustonic.com/`

[7] `http://www.openvirtualization.org/`

4. ANDIX Architecture

This chapter will give a overview about the architecture of the ANDIX operating system and its components. The role and the responsibilities of each component will be discussed.

ANDIX OS was developed as a multitasking, non-preemptive operating system.

Figure 4.1 provides an overview of the ANDIX operating system and its components. The components are grouped by the system mode they operate in. In the secure world userspace we split the components into two groups, because the Trusted Execution Environment (TEE) runtime library is provide the standardized runtime library. A list of the component groups follow:

- Secure World Kernel: The secure world kernel of ANDIX. (4.1)
- Secure World userspace libraries: A collection of userspace runtime libraries for ANDIX. These include Secure libc syscalls, newlib and tropicSSL. This component is the base for implementing the TEE runtime library. (4.2)
- Secure World TEE library: The TEE runtime library for Trusted Applications. This component provides the standardized runtime functions for the TEE, as defined in GlobalPlatforms Internal Application Programming Interface (API). (4.4)
- Normal World Kernel module: A Kernel module for the normal world operating system, responsible for communicating with the secure world. (4.6)
- Normal World Trusted Execution Environment Client (TEEC) library: The Trusted Execution Environment Client (TEEC) runtime library for userspace applications. It abstracts the specifics of ANDIX OS to standardized library functions [Glo10a] . (4.3)

ANDIX and its supporting infrastructure consists of many components. These components are designed to provide a trusted execution environment for the so-called Trusted Applications. Trusted Applications should be implemented using the TEE Internal API Specification [Glo11] . Every Trusted Application

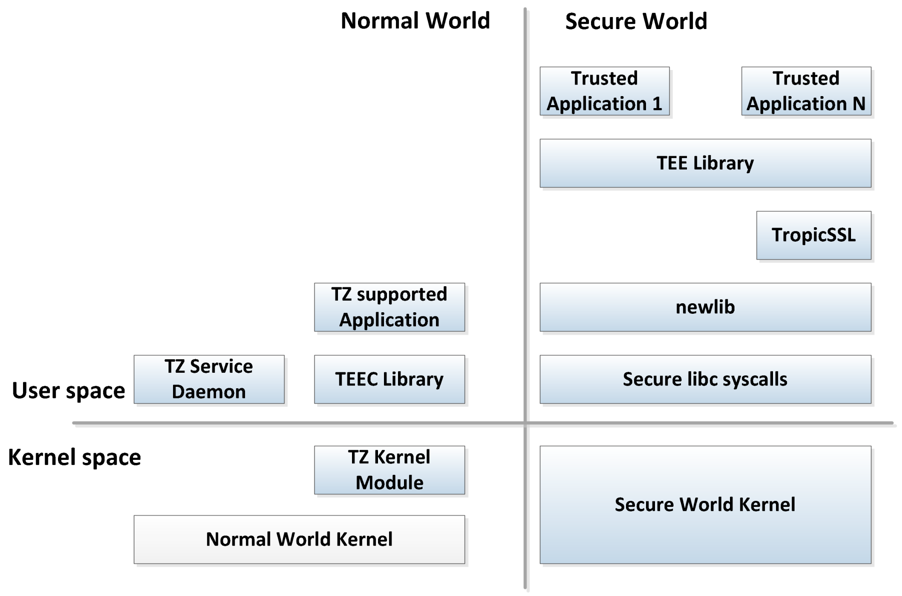

Figure 4.1.: Infrastructure components overview

implements five predefined functions, which act as the Trusted Application Interface (see 2.3.2). This interface implements the Trusted Application Lifecycle. The infrastructure will call these functions according to the specifications [Glo11] . The Trusted Applications can access a set of functions also defined by [Glo11] . These Trusted Applications are protected pieces of code and can be compared to smartcard applications. The Trusted Application interface exposes a remote procedure call based design. Trusted Applications usually offer small functions which can be used by other Trusted Applications or normal world applications. These functions are application specific and encapsulate sensitive informations. There is a defined API for normal world applications for calling Trusted Application functions [Glo10a] . By instrumenting this API normal world applications with corresponding Trusted Applications can hide security critical functions inside the Trusted Application, and therefore in a secured environment. The functions exposed by a Trusted Application should be called via the TEEC API.

4.1. Secure World Kernel

The secure world kernel is the most fundamental part of ANDIX with respect to security. The whole infrastructure is build around the secure world kernel. It has to manage the security functions provided by the ARM TrustZone to achieve isolation. This includes protection of the secure world memory and protection of secure devices. The normal world is not authorized to access the physical memory that is used by the secure world. The isolation of the two worlds is fully controlled by the secure world kernel. The kernel not only has to isolate itself from the normal world, it also has to manage the secure world Trusted Applications. This includes isolation between the Trusted Applications, initialization of Trusted Applications and providing the system call interface for the Trusted Applications. This system call interface abstracts common requests for memory management, file based input and output, hardware abstraction and special requests to implement the trusted execution environment. The secure world kernel also has to prepare the system for the normal world kernels boot process. Therefore, the secure world kernel becomes the bootloader for the normal world and can verify the integrity of the normal world system before its boot process. Another important task of the secure world kernel is to detect possible attacks on the system security and to perform countermeasures against these attacks. When the secure world kernel detects a possible attack, for example an unauthorized access to secured memory, it receives an data abort exception. The secure world kernel will handle the exception by taking full control of the system and stopping. The kernel is explained in more detail in chapter 5.

4.2. Secure World Userspace Libraries

This section describes the userspace programming libraries to enable standardized C programs. The most important library is the C Standard library also called libc.

The libc is a library for C that provides the basic runtime functionality for C programming. The secure world userspace libc implementation is an adapted version of the newlib[1] library. To provide certain basic functions, libc needs to use functionality only the kernel can provide. One example would be Input and Output functionality. The userspace application therefore has to delegate

[1]https://sourceware.org/newlib/

certain tasks to the kernel. This delegation is accomplished by performing a Software Interrupt. How this mechanism works differs from operating system to operating system. These delegations are called system calls. The newlib library defines method stubs for system calls, which have to be implemented to port the newlib to a new operating system. The secure world system call library implements these system call stubs using ANDIX kernel system call interface. Therefore, it acts like glue between the secure world libc (newlib port) and the secure world kernel system call interface. We chose newlib to implement the userspace c runtime, because newlib is simple to port to new operating systems.

Another important part of userspace library collection is the C runtime initialization also called crt0. The crt0 depends on the operating system (OS) and this small code is executed first when a new userspace application is loaded. It defines the standard bootstrap code for userspace applications. Usually it is used to call the main function in the userspace application and store the result as the exit value of the application. In the ANDIX OS it is used to instrument the TEE Interface functions, which are the special entrypoints for Trusted Applications.

4.3. Normal World Trusted Execution Environment Client Library

The normal world Trusted Execution Environment Client (TEEC) library implements the API defined in the TEEC API specification [Glo10a] . This enables source code compatibility to other TEEs. The TEEC library uses the pseudo character device exposed by the kernel module to abstract the ANDIX OS specific world communication up to the API from GlobalPlatform [Glo10a] (see 2.3.2). Normal world applications should use the functions provided by this library to communicate with Trusted Applications.

4.4. Secure World Trusted Execution Environment Library

The TEE library implements parts of the standardized TEE Internal API Specification [Glo11] . This API provides functions usable by Trusted Applications. The API itself is implemented on top of the libc implementation and the tropicSSL library. The tropicSSL library is needed to provide cryptographic functions. Trusted Applications must only use functions from the TEE library. This ensures source compatibility to other TEEs. The TEE library provides the following functions. We only implemented parts of the API, because the implementation of the full API was out of scope of this thesis. We focused more on building the underlying OS and the supporting libraries. With ANDIX OS we reached a point, where the full GlobalPlatform TEE Internal API can be implemented in the secure userspace. We cannot describe the full GlobalPlatform API in this thesis, for detailed information on the GlobalPlatform TEE Internal API please see [Glo11] . Here is a list of the GlobalPlatform TEE Internal API parts with a short description:

memory management Memory management provides functions to allocate and to free memory. The ANDIX OS TEE runtime library implements the memory management part of the GlobalPlatform TEE Internal API.

trusted storage Trusted storage provides functions to store data persistently or temporarily in a well structured form. The ANDIX OS TEE runtime library does not implement the trusted storage part of the GlobalPlatform TEE Internal API.

timing Timing provide function for timing. The ANDIX OS TEE runtime library does not implement timing operations of the GlobalPlatform TEE Internal API.

arithmetic and logic operations on big integers Arithmetic and logic operations on big integers provide functionality for big integer operations. The ANDIX OS TEE runtime library does not implement arithmetic and logic operations on big integers as defined in the GlobalPlatform TEE Internal API.

cryptographic operations Cryptographic operations provide functionality for symmetric and asymmetric cryptography. Also cryptographic hash functions are provided. This contains also data types to store cryptographic primitives in the trusted storage. The ANDIX OS TEE runtime library does not implement the cryptographic operations as defined in the GlobalPlatform TEE Internal API, but we provide cryptographic operations via the TropicSSL library. To support the GlobalPlatform TEE Internal API, we would need to implement wrapper functions around the TropicSSL library.

inter process-communication Inter process-communication provides functions communicate with other Trusted Applications. The ANDIX OS TEE runtime library does not implement inter process-communication as defined in the GlobalPlatform TEE Internal API.

4.5. World Communication

The goal of the thesis is to provide a TEE. But normal world applications have to be able to communicate with the Trusted Applications running inside the TEE. Figure 4.2 shows the logical communication channel, where a normal world application communicates with a Trusted Application and the real communication channel.

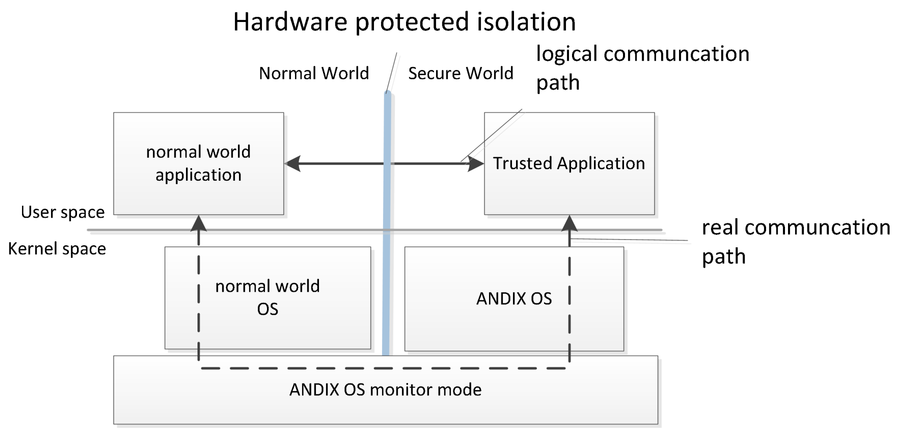

Figure 4.2.: World communications

The real communication channel is realized by world switches. The two worlds communicate via a system call like interface. Both worlds use the monitor mode to switch between modes of operations. To do this the normal and the secure world can issue a secure Monitor Call (SMC) instruction. This SMC instruction will trap the execution flow into the monitor mode. 32 bit values can be exchanged via the core registers, because the register values are kept during this trap into the monitor mode. One of these 32 bit values is used as a memory location in the physical memory. The normal world cannot access the secure memory, but the secure world can access all memory. Therefore the normal world will allocate a memory block containing the relevant information for the logical communication. It will load the physical memory location of this memory block into the core register zero. After the SMC instruction the monitor mode will transfer the given memory location to the secure world, which in order can access the memory block. A possible response of the secure world also has to be written into the memory block provided by the normal world. So the normal world application has to allocate enough memory for the response.

On top of the real communication channel a logical communication channel is implemented. The logical communication channel implements the GlobalPlatform TEEC API in the normal world and the GlobalPlatform TEE Internal API in the secure world.

4.6. Normal World Kernel Module

The normal world kernel module is used to abstract the communication mechanisms between the normal world and the secure world. It is implemented for the Linux kernel of the rich OS and provides a pseudo character device, to expose communication capabilities to normal world applications. The communication between the two worlds is based on a system call like interface. The normal world kernel module implements the logical channel of the world communication on top of the real communication channel. When we look at the GlobalPlatform TEEC API, a Client Application can open sessions to Trusted Applications. The kernel module has to keep track of these sessions. According to GlobalPlatforms TEEC API, a Client Application can invoke commands in a Trusted Application. This invoke can take up to four parameters, that can either be plain integer values or memory buffers. The kernel module has to semantically understand the type of the parameters and prepare the secure world request accordingly, because memory buffers have to be allocated in the kernel space, so that they remain in the same physical memory location. The normal world kernel module implements the real communication channel with the secure world and provides the functional foundation to implement the normal world TEEC library.

5. Secure World Kernel Design

5.1. Overview

The heart of the whole ANDIX OS is the secure world kernel. The kernel consists of multiple subsystems. We want to provide a Trusted Execution Environment (TEE). To achieve this we need to provide memory and Input/Output (IO) isolation for the secure world to protect the data of the Trusted Applications. All of the kernel subsystems play a vital role in securing the TEE. Figure 5.1 shows the main subsystems of the ANDIX kernel.

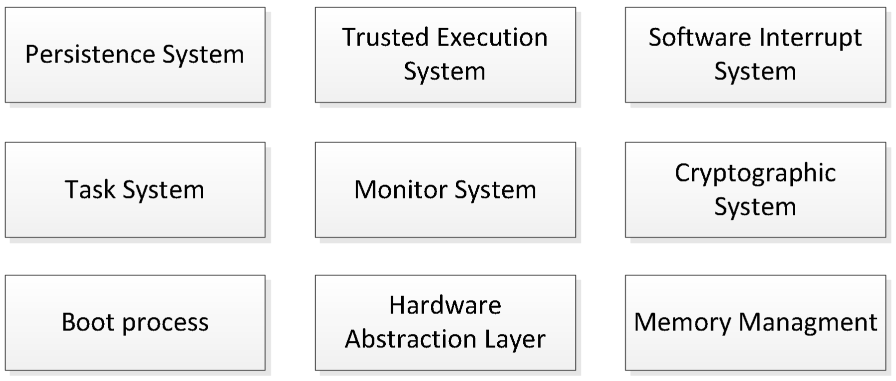

Figure 5.1.: ANDIX Kernel subsystems

The following listing will give a short description of the subsystems responsibilities. Every subsystem and its implementation is discussed in more detail in the following sections 5.2 - 5.9.

- The Boot Process is used to set up the basic execution environment including a temporary kernel stack and to initialize the main subsystems.
- Memory Management is used to control physical and virtual memory in the secure world kernel space and user space.

- Abstracting the hardware functions to a hardware independent Application Programming Interface (API) is the purpose of the Hardware Abstraction Layer
- Monitor system provides functionality for switching between the normal and the secure world and also for task switches inside the secure world.
- Task system provides support for multiple tasks in the kernel and user space.
- The Cryptographic system provides basic cryptographic functions including big number calculations.
- The Persistence system provides a file based persistent storage for the secure world.
- The Trusted Execution Environment system provides the functional foundation to implement logical world communication as proposed by the GlobalPlatform APIs [Glo10b] .

5.2. Boot Process

The boot process of the ANDIX OS starts after the boot loader. A sequence diagram of the basic boot procedure is shown in figure 5.2. The sequence diagram shows the order in which the subsystems of ANDIX OS are initialized. The boot loader places the ANDIX kernel in a predefined position of the Random-Access memory (RAM). ATAGS are data structures used to pass boot arguments to the Linux kernel. The boot loader loads ATAGS into the RAM and puts the physical address of the ATAGS into processor register 2. Register 1 holds the platform identification number, which identifies the hardware platform. This boot process is one possible boot process for the Linux kernel [Rus02] . Therefore, existing boot loaders can be used. When this basic environment is set up, the boot loader transfers control to the initial boot code in the ANDIX kernel. In the future we will utilize the boot loader to not only load ANDIX, but to also load the rich operating system. This way we can reduce the code size of ANDIX and the Trusted Computing Base (TCB) of the system, but this change was out of scope for this thesis.

The initial boot code is implemented in the ARM assembler language and relocates the kernel to a different location in the physical memory. This location is predefined and depends on the platform. To find the correct location the initial boot code uses the platform identification number and looks up the location for this platform. After the relocation is done the initial boot code also sets up the virtual memory by initializing the Translation Table (TTBL). A one to one mapping between the current physical memory location and the virtual memory is created. Furthermore, physical memory is also mapped to the virtual memory address starting at 0xC0000000. This virtual memory address is the base address the kernel is linked to. Also a platform depend IO memory location is mapped to the virtual memory. This provides basic IO functionality until the Hardware Abstraction Layer (HAL) subsystem is initialized. When the TTBL is initialized the initial code turns on the Memory Management Unit (MMU). Then the platform is operating with virtual memory in the secure world. The last responsibility of the initial boot code is to set up a stack. This is accomplished by simply setting the current stack pointer to a preallocated static memory region. The register values of the boot loader are restored with the location of the boot parameters (ATAGS) and the platform identification number. With an absolute jump instruction the initial boot code enters the main boot code.

The main boot code is implemented in the C programming language. First of all it stores the platform identification number and the physical location of the ATAGS. It is furthermore responsible for initializing the most basic subsystems in a particular order. The first system to be initialized is the memory management system. A detailed description of the system can be found in the next section 5.3. After the initialization of the memory management system, the dynamic memory, mapable memory and stack memory are available. The second system to be initialized is the monitor subsystem (see section 5.6). When this system is ready, secure monitor calls can be performed. The third system to be started is the HAL. The HAL uses the mapable memory to map IO memory of devices to virtual memory and the TEE uses mapable memory to map shared memory to the kernel space. If a TrustZone aware memory controller is available, it will be used to secure the physical part of the memory, used by the ANDIX OS. This memory controller is the device, which fetches memory data from the RAM banks to the system bus. The last system to be initialized in the main boot code is the task system (see section 5.5). After the task system initialization, the main boot code is able to create the main kernel task. It will use the monitor system to change to the main kernel task and consequently complete the basic boot procedure.

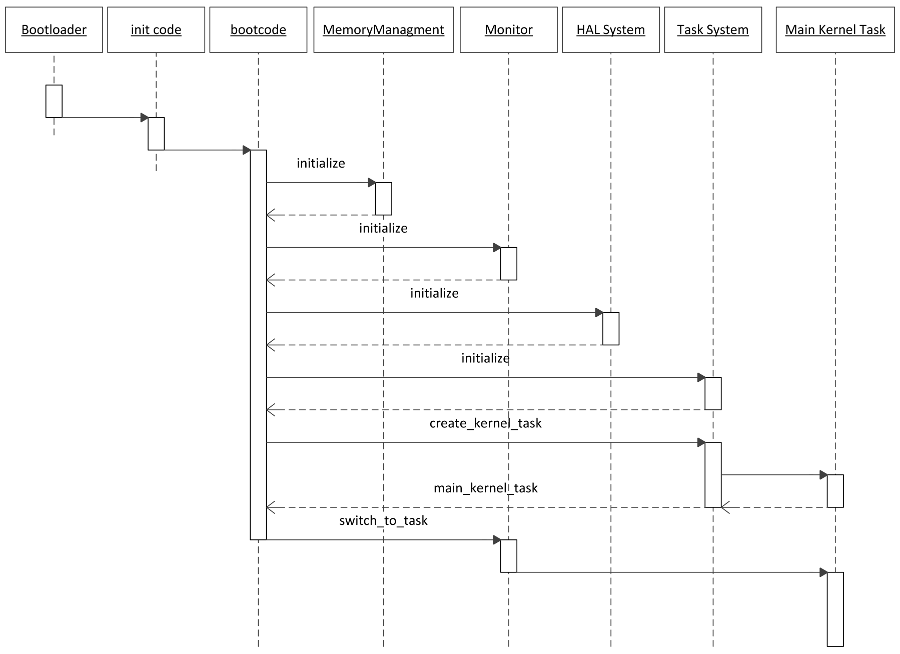

Figure 5.2.: ANDIX boot sequence

5.3. Memory Managment

5.3.1. Overview

The memory management system of the ANDIX kernel manages virtual as well as physical memory. The memory management is providing isolation between the ANDIX OS kernel and user space application, but also inbetween the different user space applications. This provides an important part of the domain isolation for the security concept of ANDIX OS. This system consists of multiple subsystems. The different subsystems will be discussed in detail in the subsequent sections of this chapter. The virtual memory is managed with the MMU (cf. [ARM12]). A Translation Table (TTBL) is a data structure that is used by the MMU to translate virtual memory addresses to physical memory addresses. A TTBL is better known as page directory in the x86 architecture. To split user and kernel space memory, two TTBL are used. One maps all virtual memory from the memory address 0x80000000 to the memory address 0xFFFFFFFF. This is the whole kernel space of the virtual memory layout. The second one maps the virtual memory from the memory address 0x00000000 to the memory address 0x7FFFFFFF. This is the whole user space of the virtual memory layout. We decided to use this much virtual memory for the kernel, because two gigabyte of virtual memory will suffice for Trusted Applications. The Trusted Applications should only be small applications and should not need this much memory. Figure 5.3 shows the virtual memory layout of every process in the secure world in ANDIX OS. The kernel space memory (0x80000000 - 0xFFFFFFFF) always maps to the same physical memory for every process. Only the lower part of the virtual memory, the user space part (0x00000000 - 0x7FFFFFFF) maps to different memory for the different user space processes. By splitting the memory space with the two TTBL at exactly this boundary makes user space switches faster, because only the first TTBL, responsible for the user space has to be exchanged.

Following is a list describing the memory areas of the virtual memory layout (see fig. 5.3):

unmapped The unmapped area in the user space from 0x00000000 - 0x00008000 is not mapped. Accessing memory in this area will lead to a data abort exception and can be used to catch null pointer errors. ANDIX OS does not handle null pointer errors yet, but this area is reserved for the future use to do so.

User code/static data In this virtual memory area the user space code and the static data segment of the user space application is located.

User heap This virtual memory area the dynamic memory is located.

User stack This virtual memory area contains the stack for the user space application.

Mappable Memory The mappable memory area is a virtual memory area inside the kernel space, that is available for memory mapped IO and normal world physical memory. It is used to make physical memory, that is not constantly available in the virtual memory layout, accessible, for example normal world physical memory.

Kernel code/static data This virtual memory area contains all the kernel code and the kernel static data segment.

Kernel Heap The kernel heap area is a fixed size memory area, that is pre mapped to physical memory. This area is used by the kernel to dynamically allocate memory.

Kernel Stacks In this virtual memory area the stacks for the different kernel tasks are located. The kernel space if the same for every task, so in this area all kernel stacks are co located.

The memory system initialization starts by analysing the memory ATAGS. It will initialize the kernel heap memory just after the end of the memory of the static kernel data with a constant size.

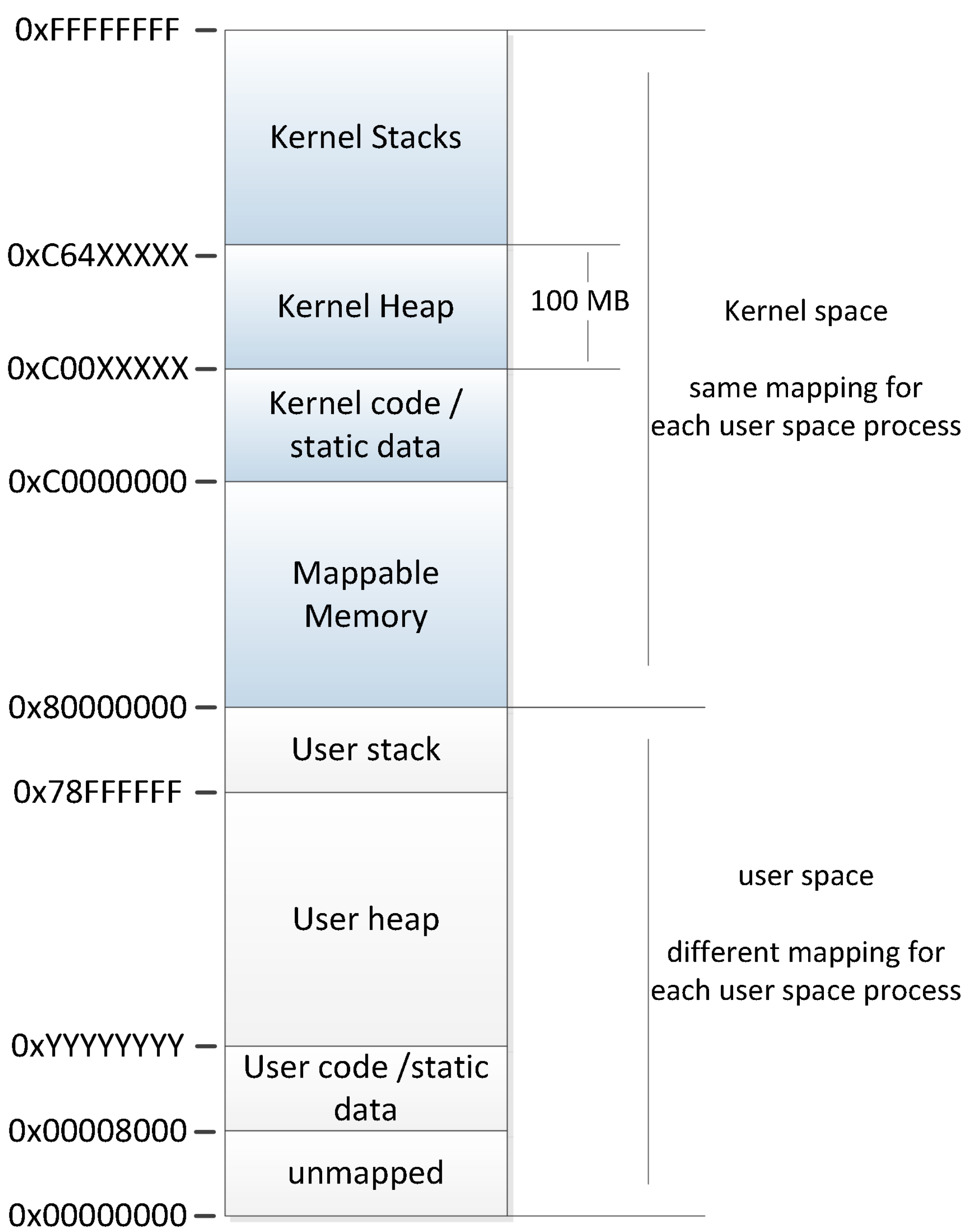

Figure 5.3.: ANDIX virtual memory layout

5.3.2. Physical Memory Manager

The physical memory manager is responsible for keeping track of the physical memory usage. It will detect the physical memory address range ANDIX OS is located in. This physical memory has to be protected from normal world access. A virtual list of memory address ranges is generated and one memory address range of a predefined size is marked as secure (see 2.5.2). The remaining memory in the secure memory address range, after subtracting the static size of the kernel and the size of the kernel heap, is used for the physical memory manager.

The physical memory manager divides the secure memory address range into pages with the size of 4096 bytes each. These pages can be in use or free. A simple bitmap is used to track the state of each page. To allocate physical memory pages the requested number of free pages has to be found in the bitmap. Only one bit in the bitmap has to be set to release a physical page. Calculating the physical memory address from the bitmap position can be done by knowing the start address of the physical memory segment. Equation 5.1 gives the physical memory address of a memory page of any memory page. We took the concept *Memory Management with Bitmaps* from [Tan07] .

$$pageAddress = physicalStartAddress + (position) * 4096 \tag{5.1}$$

5.3.3. Virtual Memory

ANDIX OS uses multiple managers for different parts of the virtual memory layout. The virtual memory space of the kernel heap is managed by the kernel heap manager (5.3.3.1). The kernel stacks area is managed by the stack memory manager (5.3.3.3), the mappable memory area is managed by the mappable virtual memory manager (5.3.3.2), and the virtual memory of the user space is managed by the user virtual memory manager (5.3.3.4). The mappable memory manager, the kernel stack manager, the user virtual memory manager and the physical memory manager are all implemented the with the same principal of a bitmap (see 5.3.2).

5.3.3.1. Kernel Heap Manager

The Kernel Heap Manager manages the dynamic kernel memory, also called the kernel heap (see fig. 5.3). It is pre-mapped into virtual memory. We based the implementation of the kernel heap memory of ANDIX OS on Doug Lea (cf. [Lea]) heap concept. The heap memory is organized in blocks of memory. Each of these blocks contains a simple header followed by the memory space for the actual data. This header contains the size of the previous block, a simple flags field and the size of the current block. After this header the actual data starts. The complete heap memory consists of these blocks. Figure 5.4 depicts this structure. The flags field of the structure, keeps the state of the heap block up to date, whether if it is used or not used. Given this structure, it is easy to navigate the blocks backward and forward.

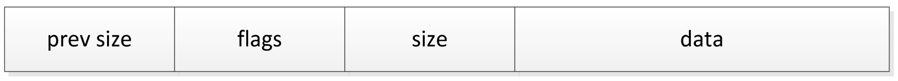

Figure 5.4.: Kernel heap structure

During initialization, one heap block that spans over the whole heap memory is generated and marked as free.

To allocate memory on the heap, a free block with enough space has to be found. To find a free block we start looking at the first block in the heap memory. We check if the block is free and has enough space available. If not, we add the size of this block to our current block address to move to the next block. Before accessing the next block we need to check the address if we already reached the end of the kernel heap. If such a block is found, we need to decide if the block is split into two blocks. The block is not split if the remaining memory of the block can contain at least one heap structure with 1 byte of data. The first block must be large enough to hold the requested amount of data. This first block is marked as used and the memory address of the data part of the block is returned. The second block with the remaining memory size is just marked as free, by setting the used bit in the flags field of the header. The following memory block has to be changed as well to adopt the previous size in this block.

To release heap memory, the used bit in the flags field of the heap block has to be cleared. After this operation, the previous and the next block are checked and free blocks are merged to avoid memory fragmentation. Figure 5.5 shows a situation, where memory fragmentation is avoided during a freeing operation.

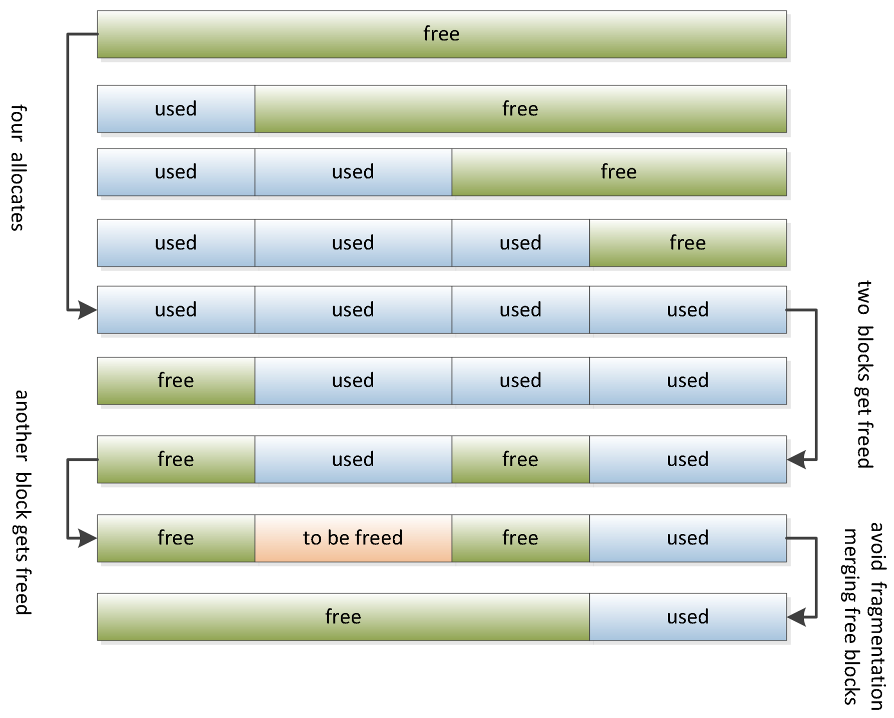

Figure 5.5.: Kernel heap release

5.3.3.2. Mappable Virtual Memory Manager

The mappable memory can be used to access physical memory, that is not always available in the trusted kernel. The Mappable Virtual Memory Manager is responsible for managing the mappable memory area of the kernel (see fig. 5.3). The mappable memory area is part of the kernel space virtual memory layout. This virtual memory is available for dynamic memory mapping. It can be used to map physical memory to the trusted kernel. This physical memory can be an IO memory are for hardware peripherals or it can be a physical RAM memory used for communication between the secure and normal world (see 4.5).

The mappable virtual memory manager also uses a bitmap to keep track of the 4096 byte pages of the virtual memory. For a detailed explanation of the bitmap concept please check the previous section. The virtual memory area is located between 0x80000000 and 0xC0000000. This gives the ANDIX OS one gigabyte of freely mappable virtual memory, that is used to access normal world physical memory for world communication and for device drivers for mapped IO.

5.3.3.3. Stack Memory Manager

The Stack Memory Manager is responsible for managing the kernel stacks. The stack memory is located after the end of the kernel heap memory (see fig. 5.3). This section of the virtual memory layout is used to store kernel stacks. This includes stacks for different operating modes like monitor or abort mode and also for different kernel tasks.

Also the stack memory manager uses a bitmap to manage the available memory pages in its memory. For a detailed explanation of the bitmap concept please check section 5.3.2.

5.3.3.4. User Virtual Memory Manager

This memory manager is responsible for managing user space part of the virtual memory layout. Every task in the ANDIX OS has its own TTBL (see 5.3.1) and its own bitmap. The user memory manager uses a bitmap for every user space task to manage virtual user space memory. It uses MMU functions to map the needed pages to the tasks TTBL. To map the needed user space when a task is scheduled to execute, the tasks TTBL is registered in the ARM core for the lower half of the virtual memory, the user space part of the virtual memory. As soon as the TTBL is registered, the MMU will use it to translate the virtual memory addresses to physical addresses.

5.4. Hardware Abstraction Layer

5.4.1. Overview

The Hardware Abstraction Layer (HAL) abstracts the access to hardware peripherals. The HAL controls the hardware based isolations of the system. It therefore provides security critical functionality for the domain isolation concept, for the memory isolation between the secure and normal world, but also for the IO isolation to provide secure input.

Different systems use different peripherals to achieve the same functionality. This layer separates the common functionality from the different implementations. It abstracts the common functionality to abstract device types. Drivers are used to implement the functionality for concrete hardware. During the initialization of the HAL, a device map is loaded based on the platform identification number. This device map describes the devices available on the specific platform. For each device the device type, which driver to use and where in the system memory the real device is located is described. Given these informations, all devices are initialized and saved in a list of platform device instances. A platform device instance is a structure containing all vital information for the device: the device type, the physical memory location, the mapped virtual memory location, the driver code and a driver-specific data pointer to store device specific information, which is necessary for the driver to work.

Figure 5.6 gives a structure diagram of the HAL system concept. Different device types will have different high level functions, depending on their purpose. For example a serial device is able to write or read a character. These high level functions use the HAL layer methods. The HAL layer delegates the function call to a concrete implementation of the HAL driver interface. If the concrete implementation of the HAL driver does not implement the needed function an error code is returned. To get hold of the correct implementation of the HAL driver interface, the HAL layer method receives a platform device instance. This platform device instance holds a reference to the concrete implementation. This concrete implementation is the real device driver. The platform device instance also holds a reference to the device data structure, that contains driver-specific data for the real device. With this HAL architecture not every driver has to implement the full HAL driver interface, because of the HAL layer.

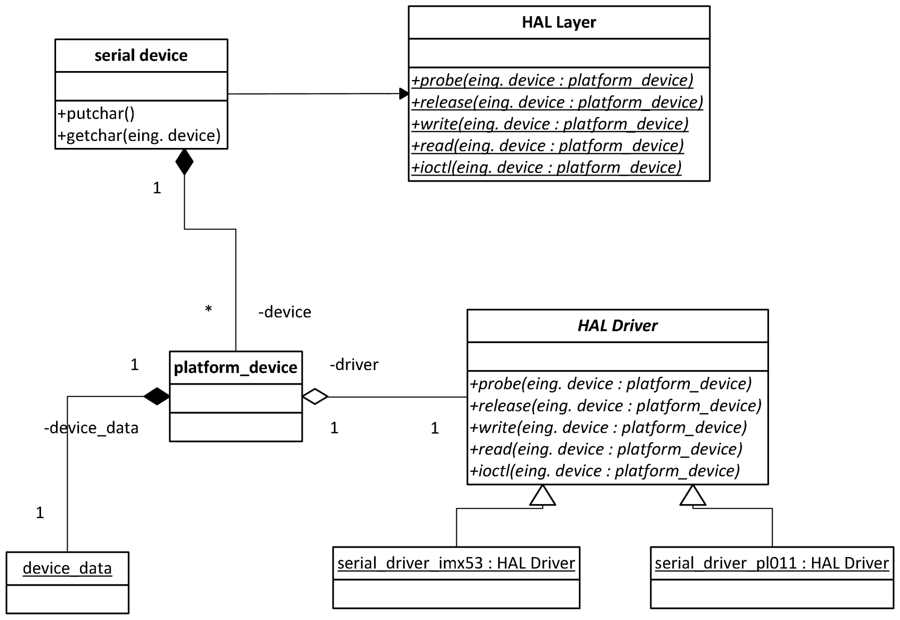
HAL Layer
+probe(eing. device : platform_device)
+release(eing. device : platform_device)
+write(eing. device : platform_device)
+read(eing. device : platform_device)
+ioctl(eing. device : platform_device)
serial device
+putchar()
+getchar(eing. device)
1
*
-device
HAL Driver
+probe(eing. device : platform_device)
+release(eing. device : platform_device)
+write(eing. device : platform_device)
+read(eing. device : platform_device)
+ioctl(eing. device : platform_device)
1
platform_device
-driver
-device_data
1
1
1
device_data
serial_driver_imx53 : HAL Driver
serial_driver_pl011 : HAL Driver

Figure 5.6.: ANDIX Hardware Abstraction Layer subsystems concept

5.4.2. Drivers

Every driver is organized as a structure containing functions pointers for HAL functions. These functions are probe, release, read, write and Input/Output control (IOCTL). The implementation of these functions are hardware specific.

The probe function is used to initialize a device instance. The driver has the opportunity to perform hardware specific initialization. For example, set up a clock for the hardware device.

The release function is used to free a device instance again. It is used to let the driver perform hardware specific release codes on the device. For example, disable a clock just used for this device.

The read function is used to read data from the device. The write function is used to write data to the device.

Because ANDIX OS was developed as a non-preemptive operating system in this thesis, we do not have to take concurrent access to devices into account.

The IOCTL function is used to perform arbitrary commands on the device. This function takes a number to encode the command and a pointer to any data structure to provide arguments. This data structure can also be used to return values from the function. The command codes and the arguments data structure are not hardware specific but specific for the abstracted device (5.4.3).

As described at the beginning of this section, the HAL has common HAL functions (probe, release, read, write and IOCTL), which take a device instance and simply use the function pointer of the associated driver to provide the hardware specific functionality. These functions therefore simply abstract the delegation process of the driver functions and ease the development of new device types.

Drivers are implemented to fulfil the vital functions on all supported hardware platforms (see chapter 1.4).

5.4.3. Devices

Depending on the type of a device high level function are available. The HAL layer provides functions to delegate hardware specific tasks of these high level function to hardware devices. The high level functions either need a reference to a platform device instance or these functions have to search the list of available platform device instances for devices of specific types. With the specific platform device instance, the HAL layer functions delegate the needed functionality to the driver and therefore hardware specific functions. The HAL layer functions need a reference to a platform device instance and call the driver specific functions.

Following is a list of supported device types:

Serial Device A serial device is a simple serial IO device. It supports reading and writing data from and to it.

Watchdog Device A watchdog device will reset the platform, if the watchdog not gets reset after a specific time. The watchdog emits an interrupt to wake up the kernel again. This can interrupt the normal world operating system (OS) or a user space application. This ensures that the kernel periodically gets the control of the system. This functionality is currently not used, but will be needed when ANDIX OS will become preemptive.

Interrupt Controller Device An interrupt controller device is used to control hardware interrupts. In a TrustZone (TZ) aware system, hardware interrupts have to be secured. An interrupt is called secured, if the interrupt controller notifies the secure world when the interrupt occurs and not the normal world. When the ANDIX OS kernel registers an interrupt handler function for a specific interrupt, ANDIX OS automatically secures this interrupt. An interrupt handler function is a function, that is called when a the interrupt occurs, it is registered for.

Secure Memory Controller Device A secure memory controller device is used to protect physical RAM regions. These regions can not be accessed, neither reading nor writing, from the normal world.

Central Security Unit Device A central security unit device is used to manage the access to the peripheral devices on the system. The peripheral device can be secured or not. If a device is secured the normal world cannot access it. (see section 2.5.3)

5.5. Task System

ANDIX OS is developed as multi-tasking, non-preemptive operating system.

The task system provides multi-task support. ANDIX OS supports multiple kernel and user space tasks in the secure world. To switch between tasks, ANDIX OS uses the monitor system 5.6. The monitor system provides context switches between the normal and the secure world. ANDIX OS uses these same world switches to switch between kernel and user tasks.

There are 3 important kernel tasks in ANDIX OS.

The main kernel task is responsible for high level initializations. It queries the user for the system password to generate the file system master key and creates the two other secure world kernel tasks. Then it prepares the statically linked user space tasks. ANDIX OS can load Executable and Linking Format (ELF) files. Currently, only statically linked user space tasks, are supported in ANDIX OS, but the ELF loader is capable of loading arbitrary executables. These static tasks are created and initialized by the main kernel task. It was out of scope of this thesis to implement a dynamic application loader, but in future work we want to implement a dynamic loading facility. Finally, main kernel task queries the user to select which non-secure guest operating system should be started and prepares the non-secure system task. The ANDIX OS is capable of hosting Linux and Android as non secure guest operating system. The available operating system kernel for the non-secure guest operating system is statically linked into the ANDIX OS kernel. ANDIX OS acts as a boot loader for these operating systems. It supports boot parameters via ATAGS and an initial ram disk.

The TEE task is the second kernel task, and it is responsible for handling the TEE requests from the non secure world. More information on this task is available in section 5.9.

The service task is the third kernel task and it is responsible to provide secure services, that have a non-secure back end. More information on this task is available in section 5.9.1.

Scheduling in ANDIX OS is simple, because ANDIX OS is preemptive. ANDIX OS is only scheduled if the normal world operating system invokes ANDIX OS. When ANDIX OS is scheduled, the normal world has requested some service from it. This means, that ANDIX OS knows which task has to be executed in order to service the request from the normal world. The scheduler in ANDIX OS looks up all available tasks and schedules the first task, that is ready to execute. ANDIX OS organizes itself, so that every time only one task is ready for scheduling, when the scheduler is called.

5.6. Monitor System

The monitor system performs world switches between the secure and the normal world. It operates in the monitor mode. The monitor system performs world switches and task switches. It detects possible attacks against the TEE, and performs countermeasures. Because the monitor system is the interface between the normal and secure world, it takes an important role in the security concept.

The monitor system offers three operations to the secure world:

1. Task switch: The task switch operation is used to switch between two ANDIX OS tasks. The target task is provided as a pointer to the task structure as the first argument.
2. Task schedule: The task schedule operation is used to call the scheduler. The scheduler searches for the next non-blocking task and hands control over to this task.

3. Non-secure service: The Non-secure service operation is used to switch to the non-secure world. The non-secure world also has its own task in the secure world.

As discussed in section 4.5 the normal and the secure world can communicate with each other, via a system call like interface. Both worlds can issue an secure Monitor Call (SMC) instruction to trap into the monitor mode. The monitor system is the code that is executed, when the SMC instruction is executed. We can think of the monitor system as a small layer of software below the normal world operating system and ANDIX OS. But it still is part of ANDIX OS. The current state of the system, secure or normal, is determined by the Non-Secure Bit (NS-BIT) in the Secure Configuration Register (SCR). But when the system is in monitor mode, the state of the system is always secure. This is a powerful property, because with this property the monitor system, which is always running in monitor mode, can switch between the two states, by manipulating the NS-BIT. When one of the worlds traps into the monitor system by issuing an SMC instruction, the monitor system is able to determine the world, that trapped into the monitor system, by inspecting the value of the NS-BIT. Trapping into the monitor system only makes sense, if some functionality of the monitor system is required.

The monitor system provides different operations for the different worlds. In ANDIX OS the requested operation is encoded as a 32 bit value and transferred to the monitor mode in register 12. This is similar to the system call number in a system call interface. The three operations available to the secure world, are enumerated earlier in this section. The arguments for the operation are stored in the registers zero to three. To pass on complex data structures, pointers to memory structures must be used. When the SMC instruction is issued from the secure world the monitor mode can simply dereference the pointer, because it operates in the same virtual memory environment as the secure world. When the SMC is issued from the normal world, the memory address has to be translated into a physical memory location and this physical memory location has to be mapped into the virtual memory layout of the secure world. To temporary map a physical memory location from the normal world into the secure world virtual memory, ANDIX OS uses the mappable memory area (see 5.3.3.2).

When an unauthorized access to secured memory occurs, the memory controller, throws an exception, that is handled by the monitor system. The monitor system, checks if the accessed memory location is within a secure memory region. It the accessed memory location is within a secure memory region, it is a possible attack against the TEE. ANDIX OS performs countermeasures against this attack, by keeping the control of the system and stopping.

5.7. Cryptographic System

To provide cryptographic functionality we ported TropicSSL to the ANDIX OS kernel space. The cryptographic system provides us with necessary tools to implement mechanisms for confidentiality and integrity in our security concept. It provides a base functionality utilized by high level systems to implement their security critical functions. TropicSSL is a fork from PolarSSL. It is the last version of PolarSSL, which was licensed under the liberal Berkeley Software Distribution (BSD) license. To port TropicSSL we removed file IO functions and added a simple implementation of the Password-Based Key Derivation Function 2 (PBKDF2) according to [Kaloo] . We used the TropicSSL library because it does not rely on any external libraries. Only a few standard C functions are required to implement the cryptographic functionality. The library is perfect for porting to a bare metal project like the ANDIX kernel. ANDIX OS generates one master key. Subsystems inside the kernel can use this key as a root secret. The master key is currently derived, using the PBKDF2 function, from a password entered by the user via a trusted IO channel, and a machine unique id. The machine unique id is unique for each platform instance. With the current implementation TropicSSL provides support for the following cryptographic functions.

Symmetric cryptographic algorithms:

- Advanced Encryption Standard (AES),
- Data Encryption Standard (DES), and
- Extended Tiny Encryption Algorithm (XTEA)

Asymmetric cryptographic algorithms:

- RSA

Message digest algorithms:

- Message-Digest Algorithm 5 (MD5),
- Secure hash algorithm (SHA1), and
- Secure hash algorithm 2 (SHA2)

Message encoding algorithms:

- Base64

5.8. Persistence System

The persistence system of ANDIX OS provides persistent storage for the secure world. To simplify development of the system and to remove hardware dependencies we decided to implement the persistence system with a non-secure back end. This means, that the persistence system uses the normal world to store the data to some long term storage device. It provides confidentiality and integrity for the data store in with this system. We cannot ensure availability, because we rely on a non-secure back end. But the usage of the non-secure back end saved a lot of functionality in the ANDIX OS kernel. No block device driver and no file system had to be implemented. This keeps the TCB smaller.

The persistence system provides a simple file system. Persistent data from the secure world is confidential and has to be protected. The system has to protect the data against off-line attacks. The persistence system uses cryptography to protect its data. The persistence system provides private root file systems to every user space task in ANDIX OS. These root file systems are implemented as prefix to the file name. The prefix is generated by the SHA2 hash of Universally Unique Identifier (UUID) of the requesting Trusted Application. A specific file is identified by the SHA2 hash of the file name. In the normal world, the data is organized in files and directories. Each root file system in the secure world is a directory, named after the hash of the prefix. The file name in the normal world is the SHA2 hash of the file name in the secure world. Dictionary attacks against the file names in the normal world are possible, but would only reveal the file name of the secure world. We process the file names, to normalize the file names for the normal world and protect against file name attacks. One example would be to create a file name containing '..' to change to a different directory.

ANDIX OS protects the data by splitting it up into blocks. The first block of each file contains an AES key data, the hashed file name and the hashed store name. Every block has a header containing an initialization vector, a Keyed-Hash Message Authentication Code (HMAC) and a hash value. The HMAC is used to ensure authenticity and integrity. The HMAC ensures correctness of the decryption function. The initialization vector is used for AES encryption and decryption, which protects the block's confidentiality. To ensure that the normal world has to answer with the correct information block when reading data back again, the key to decrypt and verify the block is derived from the initialization vector, the block number and the file specific key from the first block. Instead of the file specific key, which is encrypted in the first block the master key (see section 5.7) is used for the first block. With this mechanisms the persistence system can provide the confidentiality and integrity for data stored with it.

The service task is used to write blocks and to read blocks. The responsibility of the normal world TZ service daemon is to store and retrieve these blocks somewhere. A simple implementation just uses the common file based IO.

5.9. Trusted Execution Environment System

The Trusted Execution Environment (TEE) system is responsible for providing communication between the normal world and Trusted Applications. The TEE system takes care about the logical communication channel (see section 4.5) in the secure world. It acts as counterpart to the normal world kernel module (see section 4.6). The TEE system is invoked when a TEE request is received in the secure world. The TEE system interprets the normal world communication data and is therefore also a security critical component. It has to make decisions based on data from the normal world and therefore provides a potential attack surface. The TEE task processes the current TEE memory and extracts the requested TEE operation and parameters. This system handles context establishment, session establishment and memory registrations from the TEE API (see section 2.3.2). If the requested operation has to be handled by the Trusted Application. The request will be transferred to the user space task, that is running the requested Trusted Application. At this time the user space task is pending in a SoftWare Interrupt (SWI) call in the kernel, waiting for a TEE request.

5.9.1. TrustZone Services

Also the service task of ANDIX OS is part of the TEE system. It can be used to develop services to the secure world, which are backed by normal world components. With the service task, the secure world can call pre defined functions in the normal world. We can use this service to remove complex systems, for example full file system implementations, or a network stack, from the secure world and therefore from the TCB. Currently, the only example for such a service is the persistent system. The secure service works together with a normal world kernel module and the normal world TZ service daemon. When the normal world calls into the secure world, it hands over a physical memory block for TEE remote procedure calls, and a memory block for TZ-control remote procedure calls. The later block is used by the service task to communicate with the normal world. It fills the remote procedure call structure, marks the control block pending and hands back control to the normal world. The normal world recognizes that the response from the secure world is actually a service request and dispatches it to the normal world TZ service daemon. The normal world TZ service daemon is an application in the normal world, that polls the normal world kernel module for service requests. When the daemon receives a request, it tries to handle the request and then issues a response back to the kernel module. When the module gets the response it sends it back to the secure world. In the secure world the service task is invoked again and the response is encoded into the current communication memory, where the task can read and process the result. The persistence system uses this mechanism to store data blocks. In future work we plan to also develop a network service based on this service architecture.

The service task with the normal world backed services helps us to reduce the TCB of the platform, but on the other hand cannot reach the performance of implementations, that would simply operate in the secure world, because of the world switches. This holds especially for IO intensive operations.

6. Secure World Userspace Design

All Trusted Applications operate in the userspace in the secure world. ANDIX OS provides libraries for Trusted Applications, that run in the secure world userspace. See Figure 6.1 for the components of the secure world userspace. These components are located in the upper right quadrant of the figure. We used newlib to provide a libc runtime in the userspace environment. The secure libc syscall library implements the syscall stubs from the newlib library. The secure libc syscall library also implements specific system calls for the Trusted Execution Environment (TEE) user space implementation.

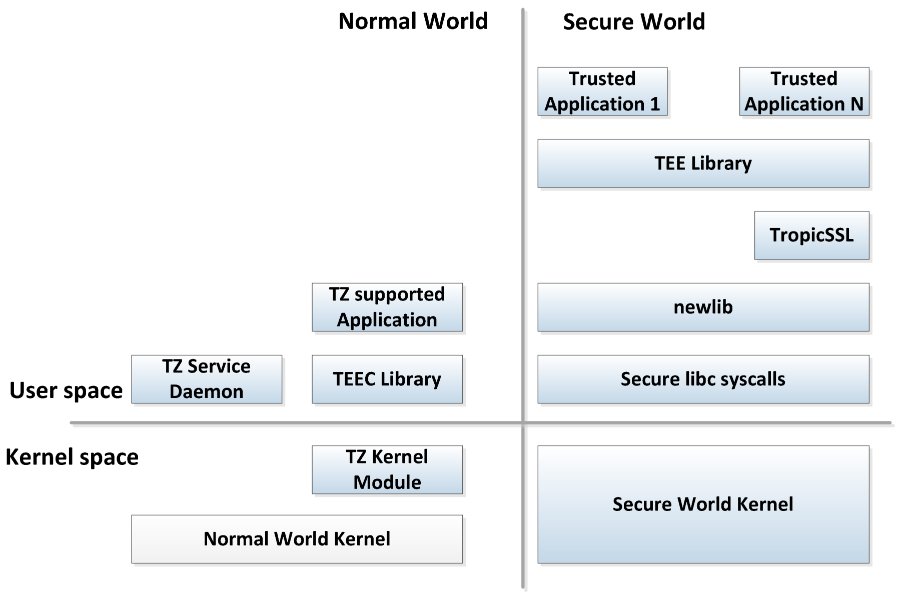

Figure 6.1.: Infrastructure components overview

Trusted Applications in the secure world user space are similar to normal C programs, but they have a different life cycle. In a normal C program the entry point is the main function. In a Trusted Application this function is not the entry point. A Trusted Application implements a five functions as defined in [Glo11] . These functions are:

- `TA_CreateEntryPoint`: This function is executed once on start up of the Trusted Application.
- `TA_DestroyEntryPoint`: This function is executed once when the Trusted Application is destroyed.
- `TA_OpenSessionEntryPoint`: This function is executed when a TEE session is created.
- `TA_CloseSessionEntryPoint`: This function is executed when a TEE session is closed.
- `TA_InvokeCommandEntryPoint`: This function is used to invoke a command issued from the normal world.

These functions implement the TA interface from [Gl011] . ANDIX OS utilizes this interface to call Trusted Application code in the secure userspace. The available userspace libraries in the secure world are listed below:

Secure Libc Syscall Library The secure libc syscall library implements the newlib syscalls functions. Furthermore, it provides the C runtime zero code, which is the main entry point into a Trusted Application. The C runtime zero code is the first code executed for every application in an operating system, it sets up an operating system specific environment. In the case of ANDIX OS it behaves very different to other operating system, because of the different life cycle of a Trusted Application. The C runtime zero code starts a loop, which polls the kernel with a non standard system call for a new TEE request. These requests are then dispatched to the TA interface functions of the Trusted Application.

Newlib Newlib is an open source libc implementation. To port newlib only the predefined syscall functions for Input/Output (IO), timing and memory manipulation like sbrk, have to be implemented. Steve Chamberlain et al. explain in [Ste10] how to port the newlib to a new operating system. For more information on the newlib library please see [1].

TropicSSL TropicSSL library is used to provide cryptographic functionality for the user space. There were two reasons why we chose the TropicSSL library.
First of all the TropicSSL library is licensed under the liberal Berkeley Software Distribution (BSD) license. It is a fork of the PolarSSL library. The fork is from the last version of the PolarSSL under the BSD license. This allows the source code of the TropicSSL library to be included and distributed in ANDIX OS, which is also licensed under the BSD license.
The second reason for TropicSSL is that it has no dependencies to other libraries. It just uses the standard c library which is available through

[1]http://sourceware.org/newlib/

newlib. Thats why the TropicSSL library is easy to port to a new operating system like ANDIX OS.

Trusted Execution Environment Library The Trusted Execution Environment (TEE) library provides a common Application Programming Interface (API) throughout different TEEs for Trusted Applications. In [Glo11] such a standardized API is described. In ANDIX OS we started to implement this API, but most functionality is not implemented yet. The full implementation of the API was out of scope of this thesis. We implemented basic memory management functions and the Trusted Application life cycle functionality. The current Trusted Application implementations utilize the the TropicSSL for cryptographic functions and the newlib library for IO functionality.

7. Normal World Components

This chapter describes the normal world components of ANDIX OS and it explains why ANDIX OS needs components in the normal world to provide a TEE.

7.1. Overview

To use the functionality provided by the ANDIX and the Trusted Applications operating inside the TrustZone (TZ), the normal world has to know about these functions. ANDIX OS provides a communication interface via the monitor mode as described in section 5.6. The necessary instruction to trap into the monitor mode, the secure Monitor Call (SMC) instruction, can only be executed in the Supervisor mode. Meaning the normal world kernel must execute this instruction. So for a normal user space application to use TZ functionality, the normal world operating system (OS) has to provide some kind of method to issue such an SMC instruction. Figure 7.1 shows the full communication path from a normal world application to a Trusted Application.

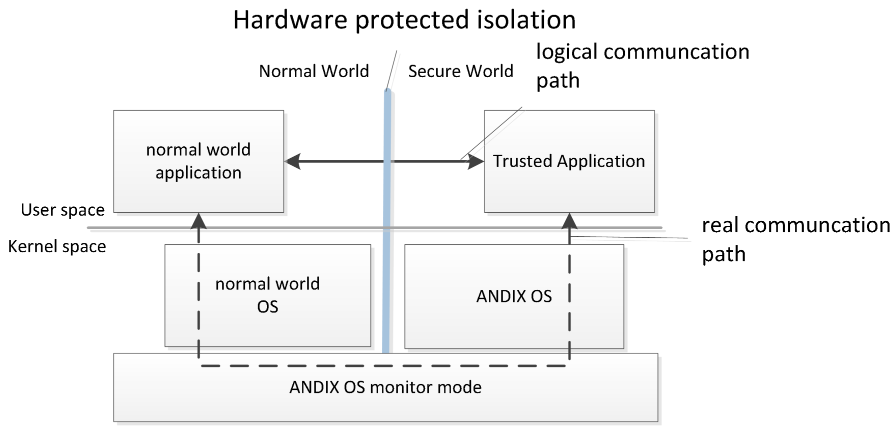

Figure 7.1.: World communications

In ANDIX OS, a Linux kernel module was developed to take care of the TZ communication. This kernel module is available in a plain Linux system, but also available for an Android system. This module allows the user space applications to use the TZ functionality. The interface between the module and user space application is specific to the ANDIX OS. The functionality of the kernel module is explained in more detail later in the next section. To remove the implementation specific interface for normal world applications, also a C library was developed. This C library provide the standardized Global Platform Trusted Execution Environment Client (TEEC) API to normal world applications. It hides the ANDIX OS specifics behind a standardized API. This API is described in section 7.3. The full documentation of the API is found in [Gl010a] . Applications developers therefore don't have to known about ANDIX OS specifics, but can simply use the standardized Global Platform TEEC API. In section 5.8 we stated, that the persistence system has a non-secure back end. This back end is implemented in the TrustZone service daemon. The TrustZone service daemon operates in the normal world user space. It uses the kernel module to communicate with the secure world, and offers services to the secure world. Currently a long term storage is implemented. This storage saves blocks of data organized in files in the normal worlds file system.

7.2. Kernel Module

The Linux kernel module provides a pseudo character device and handles Input/Output control (IOCTL) calls on this device. The IOCTL calls enable applications to push TEE requests to the secure world. These requests are send to the TZ via a physical memory location. This memory location cannot be the original memory blob from the user space application, because the kernel cannot guarantee the stability of the physical memory location. For example in a multi processor system the kernel could start to swap out the memory of the user space process containing the TEE request. When the memory is swapped in again the physical memory location is probably a different one. ANDIX OS is unable to recognize this change. The solution to this problem is that the Linux kernel module has to understand the TEE request structure and provide kernel memory locations that guaranteed to stay in the same physical memory location. The user space requests are copied to these kernel space locations and the physical address of the kernel space location is handed to the TZ. The TEE request structure may contain multiple memory blocks and all of these blocks have to be exchanged with kernel space memory blocks. This is accomplished by copying the data, from the user space memory block to the kernel space memory block.

Another advantage when the kernel module understands the basic communication methods, is that the kernel module can keep track of the TEE sessions, shared memory regions and TEE contexts. If a user space application disconnects from the TZ device without cleaning up all TZ resources, the kernel module is able to inform ANDIX OS and therefore close all TZ resources probably. So the kernel module also acts as a garbage collector for TZ resources.

ANDIX OS also utilizes the normal world for persistence storage tasks. See 5.8 for a detailed explanation of the persistence system in the TZ. The Linux kernel module also provides two IOCTL calls to support TZ service routines. One IOCTL call polls the TZ service request queue and returns a service request if available. The second one posts a service response for a specific service request, which is again transferred to the TZ in response for the service request. In ANDIX OS currently only the persistent system uses this service mechanism.

7.3. Application Library

The application library for the normal world implements and exposes the standardized API defined in [Glo10a] . It uses the Linux kernel module and abstracts the ANDIX OS specific TEE message formats and IOCTL calls from normal world applications. So normal world applications can use the standardized API to communicate with the TZ. For an introduction into the API defined in [Glo10a] see Section 2.3.2.

The defined API methods are:

- `TEEC_InitializeContext(const char* name, TEEC_Context* context)`: This method initializes a new context for the TZ communication. In the TEEC API multiple TEEs are supported. A context is a handle to one of these TEEs.
- `TEEC_FinalizeContext(TEEC_Context* context)`: This method finalizes a TZ context and cleans up all resources associated with it.
- Memory management functions:
  `TEEC_RegisterSharedMemory(TEEC_Context * context,TEEC_SharedMemory * sharedMem)`: This method registers a new shared memory region with the TZ OS. The Linux kernel module has to create a kernel space memory region and register this in the TEE TZ system.

Then this kernel space memory region is associated with the user space memory region.

`TEEC_AllocateSharedMemory(TEEC_Context* context, TEEC_SharedMemory* sharedMem)`: This method behaves the same as the TEEC_RegisterSharedMemory method, but it allocates the user space memory for the shared memory location as well.

`TEEC_ReleaseSharedMemory (TEEC_SharedMemory* sharedMem)`: This method releases the resources associated with the shared memory region including the kernel memory space and associated TZ resources.

- `TEEC_OpenSession (TEEC_Context* context, TEEC_Session* session, const TEEC_UUID* destination, uint32_t connectionMethod, const void* connectionData, TEEC_Operation* operation, uint32_t* returnOrigin)`: This method opens a new session with the specified Trusted Application. The Trusted Application is identified by the destination parameter, which is a Universally Unique Identifier (UUID). The connection parameters allow the implementation of an authentication mechanism, where the normal world application authenticates itself to the TEE. These authentication mechanisms are currently not implemented in ANDIX OS. The returnOrigin parameter describes the origin from where the call returned. In case of a success it is always the Trusted Application.
- `TEEC_CloseSession(TEEC_Session* session)`: This method closes a TEE session and releases all resources associated with it.
- `TEEC_InvokeCommand( TEEC_Session* session, uint32_t commandID, TEEC_Operation* operation, uint32_t* returnOrigin)`: This method remotely invokes a command in a Trusted Application. The command to invoke is identified by the commandID parameter. This is an application specific value. The operation parameter contains a structure with parameter types and parameter values. These values can either be simple integer values or shared memory blocks. These memory blocks can be temporary memory blocks or can be registered shared memory blocks created with the TEEC_RegisterSharedMemory or TEEC_AllocateSharedMemory methods. Temporary memory blocks do not have to be registered before use, but also need a little overhead in the TEE communication system, because they are internally registered and unregistered for every call in the TEE communication system.
- `TEEC_RequestCancellation(TEEC_Operation* operation)`: This method allows pending remote procedure calls into the TZ to be cancelled. ANDIX OS does not support asynchronous TZ operations and therefore also does not implement this method.

7.4. TrustZone Service Daemon

The TrustZone service daemon polls the TZ pseudo device for service requests, like service requests for the persistent system. When such a request appears in the queue, the service daemon receives it and handles the request. The service daemon issues a service response to every handled request and pushes this response through a IOCTL call into the Linux kernel module. The kernel module again sends the response back to the secure world.

Currently the daemon allows the TZ OS to read and write files in a predefined directory. Also the size of a file can be queried. This is the non-secure back end of the persistence system (see section 5.8). This daemon is actual responsible for storing the secured data blocks from the persistence system. It simple saves these blocks into files, and allows the persistence system to read and write these blocks again.

The daemon can also be extended, to provide network functionality to the secure world. This would allow the secure world to use networking, but without the need to implement of a full network stack.

The TrustZone service daemon acts as the back end system to secure world services. Secure services backed by this daemon, lack performance in comparison to full implementations in the secure world, because of the world switches, needed for each request. But depending on the system, they can reduce the Trusted Computing Base (TCB) of ANDIX OS enormously. The persistence system, is currently the only system, that is backed by the TrustZone service daemon. But to provide a long term storage natively in the secure world, ANDIX OS would have to implement a block device driver, for each platform and a file system layer.

8. Case study

We developed a sample application. This sample application provides a trusted encryption system, which encrypts arbitrary data using Advanced Encryption Standard (AES), where the secret key never leaves the secure world. It consists of a command line interface application operating in the normal world and a small Trusted Application, operating in the secure world. The application parts communicate via a predefined Remote Procedure Call (RPC) interface. The Trusted Application itself utilizes the ANDIX OS persistence system to store the keys. Figure 8.1 shows the system with all components.

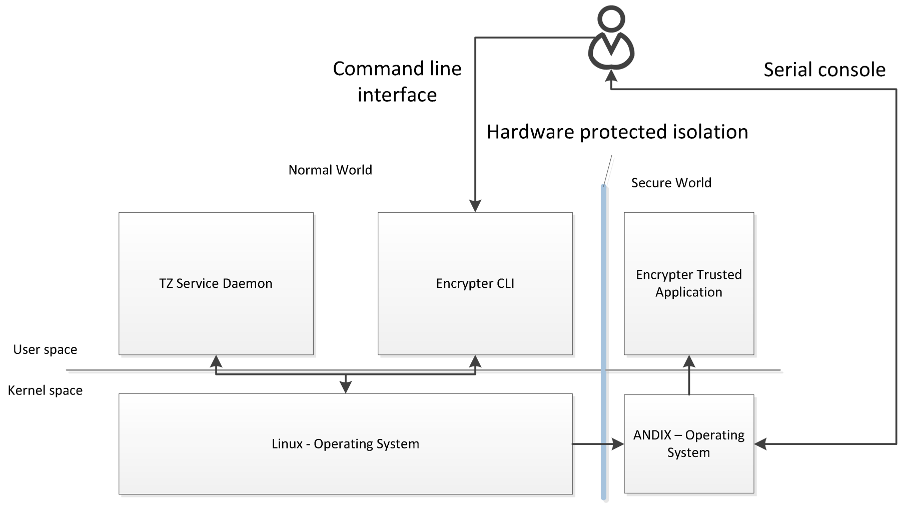

Figure 8.1.: Example encryption application

The Trusted Applications provides an RPC interface for communication with the normal world. This interface defines three operations:

- *Generate new secret key*: This operation expects an integer as a key identification to identify this specific secret key. During generation, the user is queried for a Personal identification number (PIN) to secure the generated key. This PIN input is done directly from the secure world. When using a secret key this PIN is queried directly from the secure world via trusted Input/Output (IO). This means, that the normal world kernel has no possibility to enter this PIN automatically, without modification of the hardware.

- *Encrypt data*: This operation takes a data blob as the data to encrypt and an integer as the key identifier. The data blob length has to be a multiple of 16 in bytes. Before the secret key, that is identified by the key identifier, is used to encrypt the data, the secure world queries the secret PIN via trusted IO, that is associated with the secret key. This operation authorizes the use of the secret key. The result of the operation is a initialization vector of 16 bytes and the encrypted data blob.
- *Decrypt data*: This operation takes a data blob as the data to decrypt, an integer as the key identifier and an initialization vector. The data blob length has to be a multiple of 16 in bytes. Before the secret key, that is identified by the key identifier, is used to decrypt the data, the secure world queries the secret PIN via trusted IO, that is associated with the secret key. This operation authorizes the use of the secret key. The result of the operation is the decrypted data blob.

The Trusted Application also encrypts the secret keys with AES before storing them in the persistence system. The secret key for this key encryption is generated by applying a key derivation function on the PIN. This PIN is entered directly into the TrustZone via the serial console. So this PIN is never seen in the normal world. In our current development environment (the freescale i.MX53 QSB [1]) we were not able to fully secure the serial console, by securing the device. We had only one serial console available on the board and this serial console was also used by the normal world operating system. So in our development environment we simulated a trusted user input via a shared serial console.

By encrypting the secret keys with a PIN the Trusted Application also ensures, for each operation involving a secret key, that the users consent is achieved. The user manually has to authorize the key usage by entering the PIN via the serial console.

The Trusted Application does not define an operation for secret key extraction. Secret keys cannot leave the Trusted Execution Environment.

To use the sample encryption application, the user starts the command line interface in the normal world. The users issues the command to create a new

[1] http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=IMX53QSB

key. Figure 8.2 shows a sequence diagram of the components included in the operation for creating a new secret key. First the application calls the Linux kernel to issue a new key request to the Trusted Execution Environment (TEE). This request is forwarded into the secure world to the ANDIX OS. The ANDIX OS dispatches the request to the sample Trusted Application. The Trusted Application generates a new secret key and queries the user for a PIN. It uses this PIN to encrypt the secret key and saves the encrypted secret key and a Secure hash algorithm 2 (SHA2) checksum of the plaintext secret key through the ANDIX OS persistent system. If the key was successfully saved, the Trusted Application returns a success message back to the command line interface application in the normal world.

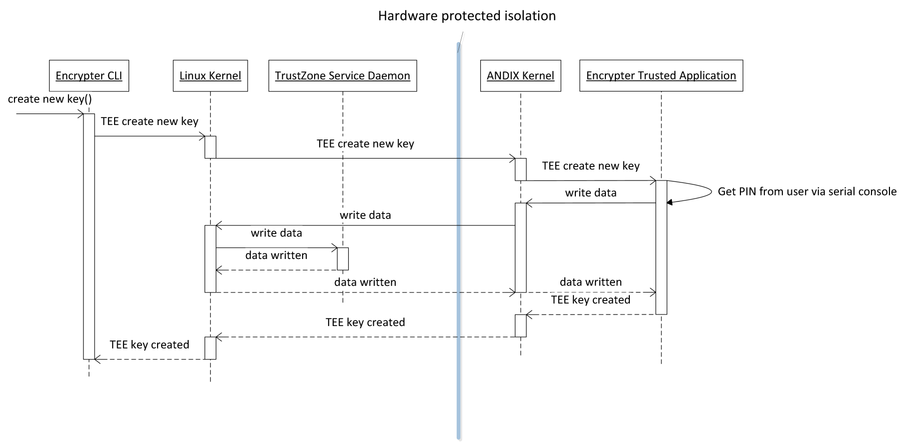

Figure 8.2.: Secret key generation

Figure 8.3 shows a sequence diagram of the components included in the encryption and decryption operation. First the user issues the encrypt or decrypt command in the command line application. The application will call the Linux kernel to issue an encrypt or decrypt operation to the TEE. This request is forwarded to the secure world to the ANDIX OS. The ANDIX OS dispatches the request to the sample Trusted Application. The Trusted Application tries to read the key data, identified by the key identifier issued with the encrypt and decrypt operation, from the ANDIX OS persistence system. The Trusted Application queries the user via the serial console for the PIN and derives a decryption key from the PIN. This key is used to decrypt the secret key. If the SHA2 checksum of the decrypted secret key matches the saved checksum, the entered PIN is correct and the use of secret key is authorized. The user gets four attempts to enter the correct PIN. If the data was encrypted or decrypted, it is returned through the TEE RPC mechanism to the command line application and presented to the user. If the user fails to enter the correct PIN an error is returned to the command line application.

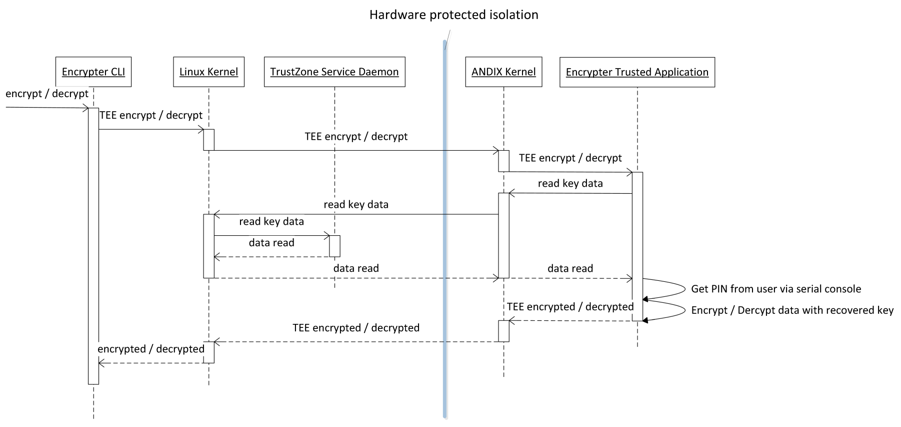

Figure 8.3.: Encryption or decryption of data

This sample application shows the separation of security critical components from user interface components. The command line interface is the user interface and the Trusted Application takes care of security critical components. All sensitive information, the secret AES keys, and the PIN cannot leave the trusted execution environment provided by ANDIX OS. Even when the Linux kernel is fully compromised it cannot access these secret informations. This sample application is a proof on concept application for solving the problem shown in the the introduction with ANDIX OS.

9. Conclusion and Future Work

9.1. Conclusion

ANDIX OS includes a monolithic kernel, which is responsible for common operating system (OS) functionality. It provides multi-tasking for kernel and user space tasks. It also implements a memory management facility, including process isolation through virtual memory. It furthermore implements software interrupts to provide a system call interface for the user space. The kernel is capable of performing asymmetric and symmetric cryptography and hash functions. It contains an Hardware Abstraction Layer (HAL), to abstract hardware devices from the concrete hardware implementation to high level interfaces. Other features of the ANDIX OS kernel are the normal world communication through the monitor mode and the Trusted Execution Environment (TEE) system.

We have ported the newlib to the trusted world user space. We developed a small libc glue code. This glue code implements the system call functions necessary for the correct operation of the newlib on top of ANDIX OS. With newlib ANDIX OS has a libc implementation available for the trusted world user space. TropicSSL is located on top of this libc implementation. TropicSSL introduces cryptographic capabilities to the trusted world user space. Based on these two libraries the ANDIX OS TEE Application Programming Interface (API) is developed. At the time of the writing of this thesis, the TEE library is not fully implemented yet, but none the less, it acts as a sufficient base for Trusted Applications as demonstrated.

We developed a Linux kernel module to ease communication between the normal world and the secure world. It abstracts the implementation specifics of the secure Monitor Call (SMC) interface. The kernel module provides a pseudo character device to access the TrustZone (TZ). The kernel module functions exposed via the pseudo character device are ANDIX OS specific. To comply with the TEE API for client applications [Glo10a] , we developed a user space library to abstract the ANDIX OS pseudo device specifics and provide an standardized way for client applications to communicate with the corresponding Trusted Application.

9.2. Security Attributes

We introduced six security attributes for trusted computers defined by David Grawrock in [Gra09] . These attributes are:

- Isolation of programs
- Separation of user processes from supervisor processes
- Long-term protected storage
- Identification of current configuration
- A verifiable report of the platform identity and current configuration
- Provide a hardware basis for the protections

ANDIX OS provides multiple building blocks to gain and enforce four of these six attributes. ANDIX OS improves the *Isolation of programs*, the *Separation of user processes from supervisor processes*, the *Long-term protected storage*, and the *Provide a hardware basis for the protections* attributes with hardware backed mechanisms. ANDIX OS currently is not capable of providing all security attributes as can be seen in Figure 9.1. There are currently no building blocks for providing the *Identification of the current configuration* and therefore there can't be a building block for *A verifiable report of the identity and the configuration*. But future development of ANDIX OS the necessary building blocks can be developed to also achieve these attributes.

The building blocks ANDIX OS adds to off-the-shelf operating systems are:

- *TrustZone Address Space Controller*: This building block allows the hardware based access control list for system memory. It is implemented in the system by a ARM TrustZone Address Space Controller and utilized by ANDIX OS via the Hardware Abstraction Layer (see section 5.4.3). By providing a hardware based access control for system memory this building block improves the isolation of programs. ANDIX OS enables developers to en capsule the security critical code and information of his application into a separate application, that is executed in a Trusted environment. The memory of this environment is access protected with the *TrustZone Address Space Controller* from the normal operating system

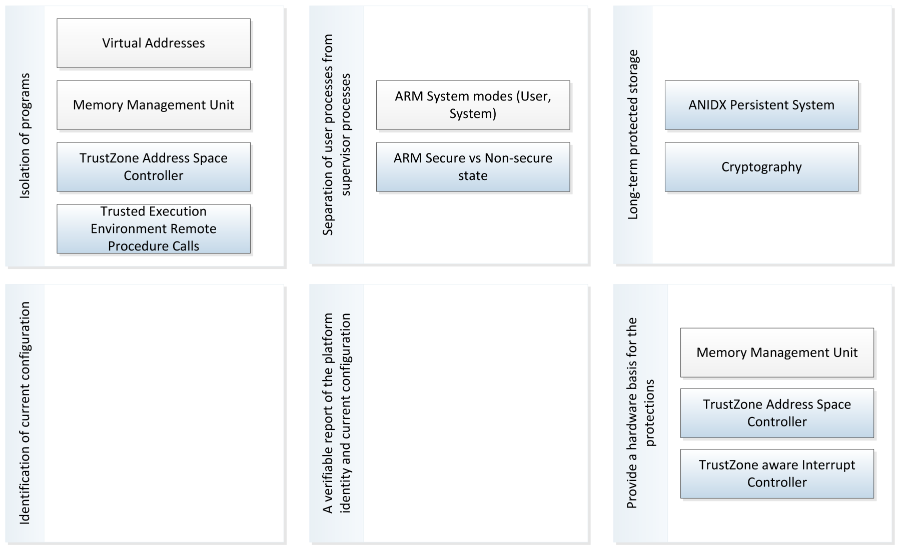
Isolation of programs
Virtual Addresses
Memory Management Unit
TrustZone Address Space Controller
Trusted Execution Environment Remote Procedure Calls
Separation of user processes from supervisor processes
ARM System modes (User, System)
ARM Secure vs Non-secure state
Long-term protected storage
ANIDX Persistent System
Cryptography
Identification of current configuration
A verifiable report of the platform identity and current configuration
Provide a hardware basis for the protections
Memory Management Unit
TrustZone Address Space Controller
TrustZone aware Interrupt Controller

Figure 9.1.: Security Attributes and building blocks for ANDIX operating system

and therefore this building block provides a hardware based isolation of programs.

- *Trusted Execution Environment Remote Procedure Calls*: This building block allows separated application parts, as described above, to communicate with each other via an interface. ANDIX OS implements this building block by implementing the Trusted Execution Environment Client API from GlobalPlatform [Glo10a] .
- *ARM Secure vs Non-secure state*: The ARM TrustZone splits the system into two states one secure and one non-secure state. ANDIX OS uses the hardware isolations provided by this system split to isolate the ANDIX OS kernel from the normal operating system. Therefore, this building block achieves the *Separation of user processes from supervisor processes* attribute utilizing hardware to separate the normal operating system from the ANDIX OS, as ANDIX OS can be seen as supervising the normal operating system.
- *ANDIX Persistent System* and *Cryptography*: The ANDIX Persistent System uses cryptography to provide a Long-term protected storage. ANDIX OS is transparently encrypts and decrypts data, before storing it. The secret keys for the encryption are derived from a user secret. Therefore without knowledge of the user secret the data is protected from access via cryptography. The user secret is acquired directly in ANDIX OS, even before the normal world is activated the first time. The user secret

and an unique platform id is than put into a key derivation function to produce the secret master key. ANDIX OS ensures, using the hardware backed memory isolation, that the user secret never leaves the TZ.

- *TrustZone aware Interrupt Controller*: This building block allows ANDIX OS to secure and isolate interrupts. This is important for securing peripheral devices for exclusive access from the secure world. When secured these interrupts are always routed to ANDIX OS by the interrupt controller and can only be unsecured by ANDIX OS again. ANDIX OS utilizes special hardware functions which have to be implemented by the hardware interrupt controller and abstracts these in its Hardware Abstraction Layer (see section 5.4.3). In conjunction with the *TrustZone Address Space Controller* the *TrustZone aware Interrupt Controller* can be used to fully secure hardware devices. And can therefore be seen as a hardware device access control system. This gives ANDIX OS full control over the platform and the possibility to securely access every device on the system. Therefore the building blocks *TrustZone Address Space Controller* and *TrustZone aware Interrupt Controller* provide a good basis for hardware protection.

We demonstrated ANDIX OS capabilities by developing a secure encryption application, that is able to hide secret keys inside the ARM TZ and using these key safely. This application can be run on real hardware platform the iMX 53 Quick Start Board (iMX53QSB) development board, with real memory isolation activated. To simplify development we ported ANDIX OS to an emulator platform.

ANDIX OS is already used in follow-up projects. A dynamic OpenSSL engine library, that is backed by ANDIX OS was already developed by Florian Archleitner. This library moves RSA operations into the TEE provided by ANDIX OS. The secret RSA keys are stored within ANDIX OS and never leave the TZ. So the RSA keys are bound to the platform. As proof of concept implementation a trusted web server based on the Apache web server[1] was developed. The trusted web server authenticates it self during an Secure Sockets Layer (SSL) connection via the ANDIX OS backed OpenSSL library. In an ongoing project Florian Archleitner is already porting the mono runtime to the secure world user space of ANDIX OS. This project is going to improve isolation inside the ARM TZ between Trusted Applications and ease the development process of Trusted Applications.

[1]http://httpd.apache.org/

ANDIX OS is an open source TZ aware OS. It was developed to run on a cheap, TZ enabled development board, the iMX53QSB. During the development process thought was put into the portability of the OS and compatibility with standards for TEE. ANDIX OS can provide a highly functional development platform for researchers and industry to develop TZ aware applications, without the need to sign an non-disclosure agreement (NDA) or invest money into proprietary development systems.

ANDIX OS provides a hardware isolated execution environment. Therefore this environment is also protected from superuser access in the rich operating system. This gives the user the possibility to executed a variety of low security applications, like multimedia applications or games, and high security applications, like on-line banking or business applications, next to each other. The high security application can utilize the execution environment provided by ANDIX OS to protect their sensitive informations.

9.3. Future Work

ANDIX OS enables many possibilities for future work. Future work includes applications based on ANDIX OS, security improvements of ANDIX OS, porting ANDIX OS onto more platforms, and supporting more rich world operating systems.

As one example application, ANDIX OS can be used to implement a TZ protected citizen card environment for the Austrian e-government infrastructure. A citizen card environment is a middle ware to allow access to the Austrian electronic identification card. The electronic identification card is a smart card that can generate digital signatures. In a TZ implementation the Trusted Application would act as smart card part and the normal world application would act as the middle ware.

Another possible extension to ANDIX OS would be to provide a trusted user interface. This would provide a trusted user interface library for Trusted Applications to use in the TZ. Global platform released an API specification for a trusted user interface library[2], which could be implemented in ANDIX OS.

We are currently porting ANDIX OS to the SABRE Lite iMX6[3] development board.

[2]http://www.globalplatform.org/specificationform.asp?fid=7779
[3]http://boundarydevices.com/products/sabre-lite-imx6-sbc/

We plan to port ANDIX OS to Samsungs Chromebook[4]. When this port is successful we plan to support Chromium OS[5] as rich operating system.

[4]http://www.samsung.com/us/computer/chrome-os-devices/XE303C12-A01US
[5]http://www.chromium.org/chromium-os

Appendix

Appendix A.

Global Platform Sample Codes

A.1. Trusted Execution Environment Client

Listing A.1: Trusted Execution Environment Client sample code

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <tee_client_api.h>
#include <tee_utils.h>
#include <trustlets/hello_world_trustlet.h>

#define RESPONSE_BUFFER_SIZE 200

int main(int argc, char** argv) {
  TEEC_Result result;
  TEEC_Context context;
  TEEC_UUID trustlet_uuid;
  TEEC_Session session;
  uint32_t origin;
  TEEC_Operation operation;
  char response_buffer[RESPONSE_BUFFER_SIZE];
  char name_buffer[USER_NAME_BUFFER_SIZE];
  int answer;

  printf("[NW]:␣Starting␣trusted␣hello␣world\n");

  memset(name_buffer, 0, USER_NAME_BUFFER_SIZE);

  // Parse Trustlet UUID
  if (TEE_uuid_parse(TRUSTLET_UUID, &trustlet_uuid) != 0)
  {
    printf("[NW]:␣Failed␣to␣parse␣UUID:␣%s\n",
        TRUSTLET_UUID);
    return (0);
```

```
}

// Ask user for user name
printf("Whats␣your␣user␣name:\n");
scanf("␣%49s", name_buffer);
getchar();

printf("Whats␣your␣answer␣to␣the␣Ultimate␣Question"
    "␣of␣Life␣(Tip␣its␣a␣number):\n");
while(scanf("␣%d", &answer) != 1) {
  while (getchar() != '\n');
  printf("Please␣input␣a␣number!\n");
}
getchar();

result = TEEC_InitializeContext(ANDIX_TEE_NAME,
    &context);

if (result == TEEC_SUCCESS) {
  // TEE Contex successfully connected
  result = TEEC_OpenSession(&context, &session,
      &trustlet_uuid, 0, NULL, NULL, &origin);

  if(result == TEEC_SUCCESS) {
    // Session to Trustlet open
    // Answer to the Ultimate Question of Life?
    operation.params[0].value.a = answer;

    // Setup response buffer
    operation.params[1].tmpref.buffer =
        (void*)name_buffer;
    operation.params[1].tmpref.size =
        USER_NAME_BUFFER_SIZE;

    // Setup response buffer
    operation.params[2].tmpref.buffer =
        (void*)response_buffer;
    operation.params[2].tmpref.size =
        RESPONSE_BUFFER_SIZE - 1;

    operation.paramTypes = TEEC_PARAM_TYPES(
        TEEC_VALUE_INOUT,
        TEEC_MEMREF_TEMP_INPUT,
        TEEC_MEMREF_TEMP_OUTPUT,
        TEEC_NONE);

    printf("Calling␣Hello␣World␣Trustlet\n");
    result = TEEC_InvokeCommand(&session,
        HELLO_WORLD_CMD, &operation, &origin);
      printf("back␣from␣Trustlet\n");
      if(result == TEEC_SUCCESS) {
        printf("Trusted␣Answer␣on␣Ultimate␣Question"
            "␣of␣Life:␣%d\n",
            operation.params[0].value.b);
        printf("Answer␣from␣Trustlet:␣%s\n",
            response_buffer);
      } else {
        // Failed to connect to Trustlet
        printf("[NW]:␣Failed␣to␣invoke␣command:␣%s␣"
            "from␣%s\n",
          TEEC_StringifyError(result),
          TEEC_StringifyOrigin(origin));
        goto cleanup;
      }
    } else {
      // Failed to connect to Trustlet
      printf("[NW]:␣Failed␣to␣connect␣to␣Trustlet:␣"
          "%s␣from␣%s\n",
        TEEC_StringifyError(result),
        TEEC_StringifyOrigin(origin));
      goto cleanup;
    }
  } else {
    // Failed to connect to TEE
    printf("[NW]:␣Failed␣to␣connect␣to␣TEE:␣%s\n",
        TEEC_StringifyError(result));
    goto cleanup;
  }

  cleanup:
  // Cleanup Trustlet Session
  TEEC_CloseSession(&session);

  // Cleanup TEE context
  TEEC_FinalizeContext(&context);
  return (0);
}
```

A.2. Trusted Execution Environment Trusted Application

Listing A.2: Trusted Execution Environment Trusted Application sample code

```
#include <tee_internal_api.h>
#include <client_constants.h>
#include <stdio.h>
```

```
#include <trustlets/hello_world_trustlet.h>

TEE_Result TA_CreateEntryPoint() {
  printf("HELLO␣WORLD:␣Started\n");
  return TEE_SUCCESS;
}

void TA_DestroyEntryPoint() {
  printf("HELLO␣WORLD:␣Destroyed\n");
}

TEE_Result TA_OpenSessionEntryPoint(uint32_t paramTypes,
                TEE_Param params[4], void** sessionContext) {
  (*sessionContext) = NULL;
  printf("HELLO␣WORLD:␣SESSION␣OPENED!\n");
  return (TEE_SUCCESS);
}

void TA_CloseSessionEntryPoint(void* sessionContext) {
  printf("HELLO␣WORLD:␣SESSION␣CLOSED!\n");
}

TEE_Result hello_world_command(void* sessionContext,
    uint32_t paramTypes,
    TEE_Param params[4]) {
  char in = 'x';
  char username[USER_NAME_BUFFER_SIZE];
  char* userptr;

  printf("HELLO␣WORLD:␣Hello␣World␣Command!\n");

  memset(username, 0, USER_NAME_BUFFER_SIZE);

  if (paramTypes
      != TEE_PARAM_TYPES(TEEC_VALUE_INOUT,
        TEEC_MEMREF_TEMP_INPUT,
        TEEC_MEMREF_TEMP_OUTPUT, TEEC_NONE)) {
    printf("HELLO␣WORLD:␣Invalid␣Parameters\n");
    return (TEEC_ERROR_BAD_PARAMETERS);
  }

  // Set the trusted answer to the Ultimate Question
  // of Life
  if (params[0].value.a == 42) {
    printf("The␣send␣answer␣%d␣is␣correct!\n",
                params[0].value.a);
  } else {
    printf("The␣send␣answer␣%d␣is␣not␣correct!\n",
                params[0].value.a);
  }

  params[0].value.b = 42;

  // Check User Name:
  while (!(in == 'y' || in == 'n')) {
    printf("Is␣your␣user␣name␣really:␣%s\n␣(y|n)\n",
      params[1].memref.buffer);
    read(0, &in, 1);
  }

  // Not real user name read trusted username
  if (in == 'n') {
    in = 'x';
    printf("Well␣we␣cannot␣trust␣the␣normal␣world␣with␣"
      "getting␣your␣real␣user␣name.\n");
    while (in != 'y') {
      printf("So␣whats␣really␣your␣username?\n");
      read(0, username, USER_NAME_BUFFER_SIZE);
      printf("Is␣your␣user␣name␣really:␣%s\n␣(y|n)\n",
                  username);
      read(0, &in, 1);
    }
    userptr = (char*) username;
  } else {
    userptr = (char*) params[1].memref.buffer;
  }

  snprintf((char*) params[2].memref.buffer,
                  params[2].memref.size,
    "Hello␣%s␣from␣the␣trusted␣World!\n", userptr);

  printf("HELLO␣WORLD:␣Sending␣message:␣%s\n",
    (char*) params[2].memref.buffer);

  printf("HELLO␣WORLD:␣Hello␣World␣Command␣done!\n");
  return (TEE_SUCCESS);
}

TEE_Result TA_InvokeCommandEntryPoint(void* sessionContext,
                uint32_t commandID,
  uint32_t paramTypes, TEE_Param params[4]) {
  printf("HELLO␣WORLD:␣Parameter␣Types␣0x%x\n",
                  (unsigned) paramTypes);
  printf("HELLO␣WORLD:␣Command␣invoked␣0x%x\n",
                  (unsigned) commandID);

  switch (commandID) {
    case HELLO_WORLD_CMD:
      return hello_world_command(sessionContext,
        paramTypes, params);
  }

  return (TEE_ERROR_NOT_SUPPORTED);
}
```

Bibliography

[Acc+86] Mike Accetta et al. "Mach: A New Kernel Foundation for UNIX Development." In: 1986, pp. 93–112 (cit. on p. 11).

[ARM07] ARM Limited. *Cortex-A8 Technical Reference Manual*. Tech. rep. ARM DDI 0344D. 2007, p. 748 (cit. on p. 17).

[ARM09] ARM Limited. *ARM Security Technology Building a Secure System using TrustZone Technology*. Tech. rep. PRD29-GENC-009492C. 2009, p. 108 (cit. on pp. 17, 19, 20).

[ARM12] ARM Limited. *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition*. Tech. rep. ARM DDI 0406C.b (ID072512). 2012, p. 2734 (cit. on pp. 15, 42).

[Aru] Arun Viswanathan and B.C. Neuman. *A survey of isolation techniques*. URL: "http://www.arunviswanathan.com/survey_isolation_techniques.pdf" (cit. on p. 10).

[Bel+12] Adam Belay et al. "Dune: Safe User-level Access to Privileged CPU Features." In: *10th USENIX Symposium on Operating Systems Design and Implementation*. Hollywood, CA: USENIX, 2012, pp. 335–348. ISBN: 978-1-931971-96-6. URL: https://www.usenix.org/conference/osdi12/dune-safe-user-level-access-privileged-cpu-features (cit. on p. 24).

[Ben+06] Muli Ben-yehuda et al. "Utilizing IOMMUs for virtualization in Linux and Xen." In: *In Proceedings of the Linux Symposium*. 2006 (cit. on p. 12).

[Fre] Freescale. *Secure Boot on i.MX50, i.MX53, and i.MX 6 Series using HABv4*. URL: "http://cache.freescale.com/files/32bit/doc/app_note/AN4581.pdf" (cit. on p. 22).

[Gar+03] Tal Garfinkel et al. "Terra: A Virtual Machine-based Platform for Trusted Computing." In: *SIGOPS Oper. Syst. Rev.* 37.5 (Oct. 2003), pp. 193–206. ISSN: 0163-5980. DOI: 10.1145/1165389.945464. URL: http://doi.acm.org/10.1145/1165389.945464 (cit. on p. 26).

[Glo10a] GlobalPlatform. *TEE Client API Specification*. 2010 (cit. on pp. 7, 14, 31, 32, 34, 64, 65, 73, 75).

[Glo10b] GlobalPlatform. *TEE System Architecture*. 2010 (cit. on pp. 12, 40).

[Glo11] GlobalPlatform. *TEE Internal API Specification*. 2011 (cit. on pp. 7, 15, 31, 32, 34, 35, 59–61).

[Gra09] David Grawrock. *Dynamics of a Trusted Platform: A Building Block Approach*. Intel Press, 2009. ISBN: 1934053171, 9781934053171 (cit. on pp. 2, 74).

[Kaa+97] M. Frans Kaashoek et al. "Application Performance and Flexibility on Exokernel Systems." In: *SIGOPS Oper. Syst. Rev.* 31.5 (Oct. 1997), pp. 52–65. ISSN: 0163-5980. DOI: 10.1145/269005.266644. URL: http://doi.acm.org/10.1145/269005.266644 (cit. on p. 11).

[Kal00] B. Kaliski. *RFC 2898 - PKCS #5: Password-Based Cryptography Specification Version 2.0*. Tech. rep. IETF, Sept. 2000. URL: http://tools.ietf.org/html/rfc2898 (cit. on p. 55).

[Lea] Doug Lea. *A Memory Allocator*. URL: \url{http://g.oswego.edu/dl/html/malloc.html} (cit. on p. 46).

[Mic03] Microsoft. "NGSCB: Trusted Computing Base and Software Authentication." In: (2003). URL: http://www.microsoft.com/resources/ngscb/documents/ngscb_tcb.doc (cit. on p. 25).

[NB13] Ruslan Nikolaev and Godmar Back. "VirtuOS: an operating system with kernel virtualization." In: *SOSP*. Ed. by Michael Kaminsky and Mike Dahlin. ACM, 2013, pp. 116–132. ISBN: 978-1-4503-2388-8 (cit. on p. 27).

[Rus02] King Russell. *Booting ARM Linux*. May 2002. URL: https://www.kernel.org/doc/Documentation/arm/Booting (cit. on p. 40).

[Sir+11] Emin Gün Sirer et al. "Logical Attestation: An Authorization Architecture for Trustworthy Computing." In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal: ACM, 2011, pp. 249–264. ISBN: 978-1-4503-0977-6. DOI: 10.1145/2043556.2043580. URL: http://doi.acm.org/10.1145/2043556.2043580 (cit. on p. 27).

[SMH01] Fred B. Schneider, J. Gregory Morrisett, and Robert Harper. "A Language-Based Approach to Security." In: *Informatics - 10 Years Back. 10 Years Ahead.* London, UK, UK: Springer-Verlag, 2001, pp. 86–101. ISBN: 3-540-41635-8. URL: http://dl.acm.org/citation.cfm?id=647348.724331 (cit. on p. 10).

[SN05] James E. Smith and Ravi Nair. "The Architecture of Virtual Machines." In: *Computer* 38.5 (2005), pp. 32–38. ISSN: 0018-9162. DOI: http://doi.ieeecomputersociety.org/10.1109/MC.2005.173 (cit. on p. 11).

[Ste10] Red Hat Support Jeff Johnston Steve Chamberlain Roland Pesch. *The Red Hat newlib C Library*. 2010. URL: ftp://sourceware.org/pub/newlib/libc.pdf (cit. on p. 60).

[Tan07] Andrew S. Tanenbaum. *Modern Operating Systems*. 3rd. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007. ISBN: 9780136006633 (cit. on pp. 9, 45).

[TPG11] Ronald Toegl, Martin Pirker, and Michael Gissing. "acTvSM: A Dynamic Virtualization Platform for Enforcement of Application Integrity." In: *Trusted Systems*. Ed. by Liqun Chen and Moti Yung. Vol. 6802. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 326–345. ISBN: 978-3-642-25282-2. DOI: 10.1007/978-3-642-25283-9_22. URL: http://dx.doi.org/10.1007/978-3-642-25283-9_22 (cit. on p. 26).

[Wah+93] Robert Wahbe et al. "Efficient Software-based Fault Isolation." In: *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*. SOSP '93. Asheville, North Carolina, USA: ACM, 1993, pp. 203–216. ISBN: 0-89791-632-8. DOI: 10.1145/168619.168635. URL: http://doi.acm.org/10.1145/168619.168635 (cit. on p. 11).

[Zel+06] Nickolai Zeldovich et al. "Making Information Flow Explicit in HiStar." In: *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*. OSDI '06. Seattle, WA: USENIX Association, 2006, pp. 19–19. URL: http://dl.acm.org/citation.cfm?id=1267308.1267327 (cit. on p. 28).

[Zel+08] Nickolai Zeldovich et al. "Hardware Enforcement of Application Security Policies Using Tagged Memory." In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California: USENIX Association, 2008, pp. 225–240. URL: http://dl.acm.org/citation.cfm?id=1855741.1855757 (cit. on p. 28).

Glossary

ANDIX OS ARM TrustZone operating system developed as part of this thesis. iii, iv, 3, 5–7, 12, 20, 23–29, 31, 35, 39–43, 45, 46, 48, 51–57, 59–61, 63–67, 69, 71–78

Android Open source operating system based on LINUX operating system. iii, 7, 52, 64

ATAGS Kernel boot parameters. 40, 41, 43, 53

Client Application Is an application not executed inside a trusted execution environment, but communicating with a Trusted Application.. 12, 14, 15, 37

libc C Standard library as defined in ANSI C Standard C99. 73

Linux Open source operating system developed by Linus Torvalds. iii, 3, 7, 26, 40, 52, 64, 65, 67, 71–73

newlib A small libc implementation developed by Red Hat and specialized for portability and small embedded systems.. 7, 73

OpenSSL Cryptographic library (`https://www.openssl.org/`). 76

PolarSSL Cryptographic library with a DUAL license (`https://polarssl.org/`). 55, 60

Qemu TrustZone TrustZone implemenation for qemu emulator by Johannes Winter (`https://github.com/jowinter/qemu-trustzone`). 6

RSA Public-key cryptosystem developed by Rivest, Shamir und Adleman. 55, 76

TropicSSL Fork of PolarSSL. Cryptographic library with a BSD license (`https://gitorious.org/tropicssl`). 7, 35, 55, 60, 61, 73

Trusted Application Is a small application executed inside a trusted execution environment, which encapsulates small security critical functions.. iii, 4, 6, 7, 10, 12, 14, 15, 24, 31–37, 39, 43, 56, 57, 59–61, 63, 66, 69–74, 76, 77

Acronyms

AES Advanced Encryption Standard. 55, 56, 69, 70, 72
API Application Programming Interface. 6, 7, 12, 14, 23, 29, 31, 32, 34, 35, 37, 40, 57, 61, 64, 65, 73, 75, 77
ARM ARM. 15, 17, 26, 27

BSD Berkeley Software Distribution. 55, 60

CPSR Current Program Status Register. 20
CPU Central processing unit. 5, 9, 15, 16, 23, 25

DES Data Encryption Standard. 55

ELF Executable and Linking Format. 52

FIQ Fast Interrupt Request. 16, 17, 19, 20

GPL GNU General Public License. 29
GUI Graphical User Interface. 5

HAB High Assurance Boot. 20
HAL Hardware Abstraction Layer. 41, 49–51, 73
HMAC Keyed-Hash Message Authentication Code. 56

iMX53QSB iMX 53 Quick Start Board. 6, 76, 77
IO Input/Output. 10, 29, 39, 41, 43, 48, 49, 51, 55, 57, 58, 60, 61, 70
IOCTL Input/Output control. 50, 51, 64, 65, 67
IOMMU Input Output Memory Management Unit. 11, 12, 27
IPC Inter-Process-Communication. 11, 27
IRQ Interrupt Request. 17

KVM Kernel-based Virtual Machine. 26

MD5 Message-Digest Algorithm 5. 55
MMU Memory Management Unit. 11, 12, 27, 41, 42, 48, 49

NDA non-disclosure agreement. 77

NGSCB Next-Generation Secure Computing Base. 25
NS-BIT Non-Secure Bit. 54

OS operating system. 4, 5, 9, 17, 20, 34, 35, 37, 51, 63, 65, 67, 73, 77

PBKDF2 Password-Based Key Derivation Function 2. 55
PIN Personal identification number. 69–72

QEMU Quick EMUlator. 26

RAM Random-Access memory. 40, 41, 48, 52
RPC Remote Procedure Call. 4, 5, 7, 69, 71

SCR Secure Configuration Register. 17, 19, 54
SHA1 Secure hash algorithm. 55
SHA2 Secure hash algorithm 2. 55, 56, 71
SMC secure Monitor Call. 5, 7, 17, 36, 54, 63, 73
SoC System on Chip. 9, 15, 16, 19, 20, 22
SSL Secure Sockets Layer. 76
SWI SoftWare Interrupt. 57

TCB Trusted Computing Base. 4, 5, 10–12, 40, 56–58, 67
TEE Trusted Execution Environment. 6, 7, 10, 12, 14, 24, 28, 29, 31, 34–37, 39, 41, 53, 55, 57–61, 63–66, 71, 73, 76, 77
TEEC Trusted Execution Environment Client. 12, 14, 31, 32, 34, 37, 64, 65
TPM Trusted Platform Module. 25–27
TTBL Translation Table. 40–43, 48, 49
TVMM Trusted Virtual Machine Monitor. 26
TXT Trusted Execution Technology. 25
TZ TrustZone. 3–7, 9, 17, 19, 20, 25, 29, 52, 57, 58, 63–67, 73, 76, 77
TZASC TrustZone Address Space Controller. 6, 19

UUID Universally Unique Identifier. 56, 66

XTEA Extended Tiny Encryption Algorithm. 55