Introspection for ARM TrustZone

Miguel Correia Guerra

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor(s): Prof. Miguel Nuno Dias Alves Pupo Correia
               Prof. Hans Reiser

Examination Committee
Chairperson: Prof. Alberto Manuel Rodrigues da Silva
Supervisor: Prof. Miguel Nuno Dias Alves Pupo Correia
Member of the Committee: Prof. Ibéria Vitória de Sousa Medeiros

June 2018
Acknowledgments

I would like to express my gratitude to my supervisors, Miguel Correia and Hans Reiser, for the support, the guidance, and the help provided along this entire journey.

I extend my sincere gratitude to Benjamin Taubmann, for his encouragement, insightful comments, and help during the different stages of my thesis.

A special thanks to Anabela Borges, for the caring and patience, and for helping me get through every moment of this adventure.

Additionally all my friends also deserve a place in the spotlight for turning the college years into an amazing journey.

Last but not the least, I would like to thank my brother and mother for their support during this thesis, which would not be possible without them.

This work was supported by the Deutsche Forschungsgemeinschaft (DFG) through the project ARADIA (grant RE 3590/3-1), and by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013 (INESC-ID).
Resumo

Nesta dissertação exploramos a TrustZone, uma extensão da arquitectura ARM que permite que software em execução nesse tipo de processador seja dividido em dois mundos: o mundo seguro para o subsistema de segurança, e o mundo normal para tudo o resto. Este trabalho tem como objectivo providenciar uma solução para analisar o estado da segurança do mundo normal, onde um sistema operativo comum (ex., Linux) e as suas aplicação são executadas. Com este propósito, foi explorado o tema de virtual machine introspection (VMI), isto é, foi usada a arquitectura TrustZone para realizar introspeção a partir do mundo seguro (que normalmente corre apenas um pequeno sistema operativo e alguns serviços de segurança), e analisar o mundo normal em busca de comportamento malicioso. Em última análise, vamos-nos focar na criação de uma biblioteca de introspeção compatível com a TrustZone, que vai providenciar acesso à memória e a funções de gestão.

Palavras-chave: Virtualização, Introspecção, Memória, TrustZone
Abstract

In this dissertation we explore TrustZone, an extension of the ARM architecture that allows software running in such processors to be split in two worlds: the secure world for the security subsystem, and the normal world for everything else. This work aims to provide a solution for analyzing the security status of the normal world, where a common OS (e.g., Linux) and applications are executed. For this purpose, we explore virtual machine introspection (VMI), i.e., we leverage the TrustZone architecture to perform introspection from the secure world (which runs typically just a slim OS and a few security services), and analyze the normal world searching for malicious behavior. Ultimately, we focus on the creation of a TrustZone compatible introspection library, that provides memory access and management functions.

Keywords: Virtualization, Introspection, Memory, TrustZone
# Contents

Acknowledgments ................................................................. iii
Resumo .................................................................................. v
Abstract ................................................................................ vii
List of Tables ............................................................................ xiii
List of Figures ............................................................................ xv
Nomenclature ............................................................................ xvii
Glossary ................................................................................... 1

1 Introduction ........................................................................... 1
   1.1 Motivation ................................................................. 1
   1.2 Topic Overview .......................................................... 2
   1.3 Objectives ..................................................................... 3
   1.4 Thesis Outline ............................................................ 3

2 Related Work ......................................................................... 5
   2.1 Virtual Machine Introspection ....................................... 5
      2.1.1 Static VMI .......................................................... 7
      2.1.2 Dynamic VMI .................................................... 9
   2.2 Dynamic Analysis ....................................................... 11
   2.3 Trusted Execution Environments .................................... 14
      2.3.1 ARM Trustzone .................................................. 14
      2.3.2 Other TEE’s ...................................................... 16
      2.3.3 Uses of the Trustzone .......................................... 17
   2.4 Introspection Libraries ............................................... 22
      2.4.1 LibVMI ........................................................... 23
      2.4.2 Virtual Introspection for Xen (VIX) ..................... 24
      2.4.3 IntroLib .......................................................... 25
3 ITZ Library Design

3.1 Use Cases ......................................................... 29
3.2 Genode OS Framework ........................................... 30
3.3 Assumptions and Threat Model .................................. 31
3.4 Architecture .................................................... 32
3.5 ITZ Library .................................................... 33
  3.5.1 Comparing ITZ to LibVMI .................................. 34
  3.5.2 Main Functions ............................................. 35
3.6 Interaction Between Worlds ..................................... 38

4 Implementation .................................................. 41

4.1 System Configuration ........................................... 41
  4.1.1 World Switch ............................................... 42
  4.1.2 Normal World Components ................................. 43
4.2 Implementation Challenges ..................................... 44
4.3 Secure World Components ..................................... 45
  4.3.1 ITZ Library ............................................... 45
  4.3.2 Other Functions ........................................... 47
  4.3.3 Implementation Differences to LibVMI ................. 48
4.4 Discussion on Different Implementation Attempts .......... 49

5 ITZ Tools ........................................................ 51

5.1 Kernel Integrity Check .......................................... 51
5.2 Tools Based on the LibVMI Examples ......................... 52

6 Evaluation ....................................................... 55

6.1 Methodology ................................................... 55
6.2 Context-Switch Overhead ........................................ 55
6.3 Micro Benchmarks of the Library Functions ................ 56
6.4 Microbenchmarks of the Hash Functions ...................... 59
6.5 Trusted Computing Base Size ................................ 60
6.6 Macro Benchmark I - LibVMI Examples ....................... 61
6.7 Macro Benchmark II - Kernel Integrity Check ............... 62
6.8 Security Considerations and Limitations .................... 63
List of Tables

3.1 Evolution of the Linux Kernel in LOC [Lö]. ............................................. 31
3.2 Comparing LibVMI to ITZ. ................................................................. 35

6.1 Microbenchmarks for translation functions (in milliseconds) ................... 56
6.2 Microbenchmarks for writing to memory (in milliseconds) ....................... 57
6.3 Microbenchmarks for reading from memory (in milliseconds) ................. 58
6.4 Microbenchmarks for printing from memory (in milliseconds) ................. 58
6.5 Code base size of our system. ............................................................... 60
6.6 Macrobenchmarks (in seconds) for the examples from LibVMI ................. 61
List of Figures

2.1 Native Hypervisor vs Hosted Hypervisor [Zil].......................... 7
2.2 Monitor Placement Options in VMI [JBZ+14].......................... 7
2.3 Ether’s Architecture [DRSL08]........................................ 12
2.4 The Concept of the Two Worlds........................................ 15
2.5 Flow of information in the Trustzone [KK14]........................ 16
2.6 VeriUI Architecture [LC14]........................................... 18
2.7 High-level Architecture of the TLR [SRSW14b]....................... 19
2.8 TZ-RKP Components [ANS+14]...................................... 22
2.9 Architecture of the LibVMI [XLXJ12]................................. 23
2.10 Sample from the LibVMI API........................................... 24

3.1 System Architecture.................................................. 32
3.2 Introspecting the Normal World Memory............................ 33
3.3 Sample of the ITZ Library API....................................... 34
3.4 ITZ Library API for Reading from Physical Addresses............. 36
3.5 ITZ Library API for Reading from Virtual Addresses............. 36
3.6 Sample of the ITZ Library API for Writing Functions.............. 37
3.7 Sample of the ITZ Library API for Translating Functions........ 37
3.8 ITZ Library API for Printing Functions............................ 37
3.9 Data Exchange using a Shared Memory Buffer ..................... 39

4.1 i.MX53 QSB Memory Layout........................................ 42
4.2 World Switch using Genode........................................... 43
4.3 Code to Read from a Physical Address.............................. 46
4.4 Code to Translate a Virtual Address into a Physical Address.... 47
4.5 Auxiliary Code to get a Virtual Address from Physical Address. 47
4.6 SHA256 on the TZ VMM.............................................. 48
Nomenclature

API  Application Programming Interface
DMA  Direct Memory Access
M4IF Multi-Master Multi-Memory Interface
MMU  Memory Management Unit
QSB  Quick Start Board
SGX  Security Guard Extensions
SMC  Secure Monitor Call
TCB  Trusted Computing Base
TEE  Trusted Execution Environment
TZ VMM TrustZone Virtual Machine Monitor
VMI  Virtual Machine Introspection
VMM  Virtual Machine Monitor
Chapter 1

Introduction

A Trusted Execution Environment (TEE) is a secure, integrity-protected, runtime environment, consisting of processing, memory and storage capabilities [EKA14]. A TEE provides isolation from the normal processing environment, where the operating system and its applications run.

ARM TrustZone is a secure extension of the Advanced RISC Machine (ARM) architecture, that allows software executed in ARM processors to be split in two environments: normal world and secure world [ARM]. The normal world or rich environment runs a common operating system (e.g., Android or Linux) and its applications. The secure world runs security services or, more generically, services that need to be isolated from the normal world.

These environments have independent memory address spaces and different privileges. While code running in the normal world cannot access the secure world address space or resources, code running in the secure world can access the normal world address space and resources [ARM, SRSW14a]. The isolation of the secure world from the normal world is important for example due to the fact that nowadays many of the commonly used protection mechanisms, such as anti-virus and intrusion detectors, are targeted by malware that is often able to disable them [Lea05, Gre].

1.1 Motivation

This thesis explores the ARM architecture to design a system that allows software in the secure world to detect malicious behavior in the normal world of a mobile device, e.g., a smartphone or a tablet. This system should run in the secure world – a TEE –, but how can code running in the secure world analyze the state of the normal world? One way of doing it is by using Virtual Machine Introspection (VMI), which is the approach of analyzing the state of a system from the outside, for instance from a secure environment, such as the secure world [GR03].
To achieve this, a few key components are needed. First, we need a VMI library that is compatible with the TrustZone, which is something that did not exist before this work. Our library – called ITZ (Introspection for TrustZone) – contains several functions for reading, translating and writing memory from the secure world, alongside with the means to communicate with the normal world and vice-versa. Second, we need a way to verify the system state with the contents of main memory, which can be done by introspection tools that use the library. These tools may serve to assure that the normal world of the current system has not been compromised, by verifying from time to time code and data stored in memory. Finally we want to measure the pros and cons of verifying the memory content on the secure world as a mean to analyze the normal world.

1.2 Topic Overview

During the development of this dissertation, several topics were studied in order to prepare for the implementation and challenges ahead. The results of that research are in detail in the next section, and cover the topics of virtual machine introspection, static and dynamic analysis, trusted execution environments, focusing on the ARM TrustZone, and finally an overview of some introspection libraries developed in the past few years.

The theme of virtual machine introspection is explored in the related work, focusing on the goals and advantages of using it, complementing with some examples. We also give a brief overview over two types of hypervisors, native and hosted, explain the difference between static and dynamic VMI, and show some of the work developed using both types of VMI. On the static and dynamic analysis topic we focus on the difference between those two approaches, and their different uses, complementing it with some work done using each of the techniques. Afterwards we study some trusted execution environments, and explore the particularities of each one, focusing primarily on the ARM TrustZone. We give in detail a description of its architecture and how the TrustZone works. To complement this, we explore some of its uses, focusing on mobile device security and kernel protection, and provide some examples. To finalize our research, we focus on some introspection libraries, in particular the LibVMI, which works as a basis to create our own introspection library. During this research project, the ITZ library was presented at an IT symposium called INForum 2017. This event took place in Aveiro, during the 2017 edition of Techdays.
1.3 Objectives

The goal of this dissertation is to show how ARM TrustZone can be used as a secure environment, in order to inspect the memory of a running system. The idea is to explore this technology and combine it with the concept of VMI. Overall the outcome is a library capable of performing secure introspection, similar to the introspection provided by libraries that currently exist for hypervisor-based virtualization solutions, such as LibVMI [Pay], but with stronger isolation. This library should provide us the means to run several existing introspection tools, as well as to create new ones. The same way ARM processors have been used to store data [CZG+15], this paper studies ARM TrustZone to support a secure environment for inspecting memory.

Besides the ITZ library design and implementation, this thesis presents a set of simple introspection tools based on ITZ and evaluation results.

1.4 Thesis Outline

The rest of this document is organized as follows. Section 2 presents an overview of the technology related to this work and methodologies that help achieving these goals. Section 3 presents a general work overview, which focus on presenting the Genode OS Framework, the architecture of the system and the library designed to perform introspection. Section 4 describes the technical details of our system, the implementation challenges and algorithms used, as well as the several components used in both worlds. Section 5 describes some small applications developed using our introspection library, as well as a more complex use case. Section 6 presents the results of the experimental evaluation study, and some insight into the different implementation attempts and limitations of our work. Finally, Chapter 7 concludes this document by summarizing our findings and discussing future work.
Chapter 2

Related Work

Over time there has been an increasing interest in software security. However no system is ever perfectly secure. So how can we find a way to inspecting software running on the system while at the same time assuring that the inspection is done in a secure environment? One way to do it is by using virtual machines. “The approach of inspecting a virtual machine from the outside for the purpose of analyzing the software running inside the VM is called virtual machine introspection (VMI)” [GR03]. That combined with ARM Trustzone should allow for a way to inspect running code and detect malicious behavior.

This section addresses these topics in order to get a better understanding of the related work done regarding our proposal. To do so we must first start by understanding the fundamentals of virtual machine introspection and dynamic analysis, and then proceed to explain the benefits of using a trusted execution environment, and why it is important to use it in our work. Section 2.1 defines what virtual machine introspection is and how it can be used, in both static and dynamic ways. Section 2.2 explores the problem of dynamic analysis, containing a description of some systems to give us some insight of its uses and benefits. Section 2.3 explains what a trusted execution environment is, focusing in particular on the ARM Trustzone, while still mentioning other existing trusted execution environments. Finally Section 2.4 will focus on introspection libraries, where we describe LibVMI, an introspection library that we used as basis to develop our own.

2.1 Virtual Machine Introspection

Nowadays there are many tools that can be used to detect and prevent potentially harmful attacks. But as tools get more sophisticated so do the attacks. The complexity of the tools might also require some sort of knowledge from the user. So the better way to create a more
secure system would be by inspecting the current state of the host, while maintaining the system performing the inspection isolated.

In order to achieve that we will be using virtual machines. One of the main goals of VMI is to assure that even in an untrustworthy OS, a secure policy is carried out while maintaining functionality [JBZ+14]. A virtual machine (VM) achieves strong isolation and confines processes running inside the VM. That makes it harder to compromise a system outside of the VM, even if the system itself has been attacked by malware [JWX07]. However this approach can only be considered secure if there are no vulnerabilities in the virtualization system, which is not always the case as Nguyen et al. refer in their paper, where they discovered 74 vulnerabilities in commonly used virtualization systems, such as Xen and VMWare [NRR+12].

Using the VM will allow for an inspection from the outside in order to analyze the software running inside, in a process called virtual machine introspection. VMI’s purpose is as the name says, to inspect a virtual machine from outside, which can be done without its knowledge. The advantages of using VMI are vast, but some of the more important can be summed up as [Zil]:

- Allowing for undetected monitoring and live analysis of the memory.
- No alteration of the target system.
- Providing more reliable data, since there is no data corruption through malware.

That being said the VMI can be used in many fields of application, but for this work security will be our main focus.

The existing mechanism that is commonly used to facilitate the VMI when constructing a new system is the virtual machine monitor (VMM). The VMM is the software responsible for the virtualization of the hardware of a single physical machine, which will then be divided in a logical way across separate virtual machines [GR03]. Using a virtual machine monitor comes with many benefits, such as allowing for direct hardware state inspection of the VM that a host is running on, monitoring both hardware and software level events and mediation of the interaction between the host software and the hardware, by performing both intrusion detection and controlling the hardware access. There are two types of hypervisor, commonly referred as type 1 or native hypervisor and type 2 or hosted hypervisor, as shown in Figure 2.1.

The type 1 hypervisor runs directly on the hardware of the host and can operate different OSs above it. Since it is independent from the OS it assures that problems occurring with a guest OS do not affect the other systems running on the hypervisor. The type 2 hypervisor is installed on an host OS and depends on it to perform its operations. Due to that any problem affecting the host OS affects the entire system, including all the guest OSs running on the hypervisor.
2.1.1 Static VMI

By further merging the capability with forensics memory-analysis techniques [DGPL11], VMI is gaining more visibility regarding security for virtualized systems [LKE15]. Static VMI, as the name suggests, involves no dynamic execution. Instead the introspection is done by using memory snapshots, without knowing the formal structure of the code ahead of time. This offers some limitations since there is little knowledge of the code entities’ behavior. Some work has already been done in this area, mainly using VM’s to detect malware and intrusions. Due to the strong isolation provided by the virtual machine it is possible to use it to get reliable information about a system even if it was compromised. On these systems, a monitor tracks the behavior of each guest OS and either detects or prevents policy violations. The placement of the monitor can be done in the hypervisor, in a sibling VM, in the guest itself, or in the hardware, as illustrated in Figure 2.2.

One of the main challenges when using VMI is that the hypervisor only has access to hardware-level abstractions, such as physical memory contents and hardware device operations.
When the goal of the VMI is security policy enforcement, we found ourselves with a semantic gap, since many desirable security policies are expressed in high-level, OS abstractions. On their paper, Jain et al. mentioned one good example of how the semantic gap creates challenges for introspection [JBZ+14]. Considering a hypervisor listing the processes running in the guest OS, we realize that it can only access hardware-level abstraction, such as the CPU registers and contents of memory pages. The hypervisor must identify specific regions of guest OS memory that include process descriptors, and interpret the raw bytes to reconstruct semantic information, such as the command line, user id, and scheduling priorities. Because of this semantic gap, there is a waste of VMI development effort, which requires some focus to be put in the reconstruction of low-level sources into high level semantic information [JBZ+14]. Jain et al. focused their work in memory introspection, where the hypervisor using the memory’s contents, makes assumptions about the guest behavior. To perform the reconstruction of the kernel data, in an attempt to solve the semantic gap problem, four techniques were explored [JBZ+14]:

- Learning and reconstruction: When performing the reconstruction, there are two phases, learning and searching. The learning phase is used to find a way of identifying and reconstructing the data structures inside the contents of kernel memory. This is usually done by extracting the signature of the data structures. The searching phase involves either doing a linear search through the kernel memory or examine data structure pointer, in order identify instances of the data structure.

- Code implanting: This technique involves code injection into the Guest OS, in order to transfer semantic information to the hypervisor.

- Process outgrafting: A technique used to prevent the monitoring of a VM from being compromised. This can be done by changing the monitoring process to a trusted VM, instead of keeping it in the VM being monitored. That way it is possible to obtain information from the kernel through introspection, without the risk of distortion.

- Kernel executable integrity: In order to make sure that the kernel is not modified, the Guest OS must have limited, or even non-existing, ability to alter its executable pages.

In conclusion, the authors noted that future VMI solutions should focus on assuring that inventive techniques and security properties care more about privacy and become more scalable.

Currently virtual machines combined with VMI, are being used for digital forensics, where they need to analyze the system as it is, without the risk of compromising the memory. Nance et al. [HN08], performed a study on the use of virtual machine introspection, regarding forensic
problems. These problems focused on the observation of volatile information without compromising it, and in the monitorization of compromised systems. It is possible to use the VMM to reconstruct the contents of a VM’s memory, since it has complete access to all of it. This allows for a forensics program to be able to use that information in order to locate processes of interest. Once located, the program might be able to reconstruct what was being done with this information. This approach allows for offline forensic analysis, preventing memory losses, and thus increasing the accuracy when reconstructing events. It also prevents the program from being detected by a potentially compromised system, which could also compromise the data obtained, leading to false information. By using the VMM to perform the analysis, it is possible to obtain a sequence of static snapshots of the memory, in an undetected way, but it is not perfect. It can suffer from a problem of consistency if the values of the locations not yet read change before being snapshoted, leading to an inconsistent reconstruction of memory contents.

2.1.2 Dynamic VMI

Dynamic VMI is the ability of performing introspection on a given system, during runtime, using virtual machines. Several works have been done using Virtual Machine Introspection to provide real-time, dynamic analysis of kernel-level malicious code. Neugschwandtne et al. presented dAnubis, a system for the dynamic analysis of malicious Windows device drivers.

Compromised computers running malware join botnets and participate in harmful activities such as spam, identity theft and distributed denial of service attacks. dAnubis helps understanding a device driver’s behavior and how it interacts with the operating system, other drivers and user land processes [NPCB10].

Their system is based on the QEMU emulator, which allows for the simulation of many processor architectures such as ARM, x86, Microblaze and many others. QEMU is able to emulate CPU instructions through dynamic binary translation, which allows it to be instrumented and monitored. Using this feature, the system observed events such as the execution of the malicious driver’s code, invocation of kernel functions, or access to the guest’s virtual hardware. The instrumentation of the emulator’s Memory Management Unit (MMU), allows for a better understanding of the manipulation done by rootkits at kernel level [NPCB10]. A memory management unit, also known as paged memory management unit (PMMU), is a computer hardware unit used to perform the translation of virtual memory addresses to physical addresses. Other uses include memory protection, cache control and bus arbitration.

Rootkits were originally designed to maintain root access after exploiting a system, while hiding the attacker’s presence [HB06]. While in the early days of the rootkits they used to run
entirely on the user space, they have evolved and are now able to perform kernel-level code attacks. The line of difference between a rootkit and malware has also become negligible, as malware nowadays is able to perform many techniques that were exclusive to rootkits, taking the form of bots, worms or Trojans.

To perform an attack on the kernel level, an attacker usually exploits some kind of weakness on the system. These exploits can take different forms, being one of the most common the buffer overflow, where a program that is writing to the buffer, overruns the buffer’s boundary and overwrites adjacent memory locations. Using the instrumentation of QEMU’s MMU, Neugschwandtne et al. detected and identified many commonly used rootkit techniques, and obtained reliable and precise information about which kernel routines were being attacked, providing a better understanding of the behavior of the malicious code. Taking into account the loss of semantic information caused the VMI approach, all exported symbols and data structure layouts from the Windows OS were extracted as a preliminary step.

dAnubis analysed more than 400 recent rootkit samples to reveal the techniques employed to subvert the Windows kernel and, in most cases, the nefarious goals attained with these techniques, proving to be an effective tool for security researchers and practitioners. However this systems showed some limitations concerning rootkit detection, dynamic analysis coverage and event attribution. Regarding rootkit detection, the system proved unable to analyse rootkits injected through kernel or device driver exploits, as there are already techniques that could be integrated to reliably detect the execution of injected code. The problem of return-oriented rootkits [HHF09] however is not handled, and no solution was provided. Return-oriented rootkits, focus on manipulating the call stack, by exploiting a bug, which allows for an attacker to execute arbitrary functions. Other limitation is the dynamic analysis coverage, in which the analysis environment can be detected, and if that happens the analysis can be disrupted if the malware refuses to run. Finally, this system, in order to attribute a write access to a monitored driver, takes the program counter of the instruction that carried out the manipulation and compares it with the codebase of the driver. However this fails if the attacker makes use of legitimate kernel functions in order to manipulate the memory [NPCB10], which can be countered by using secure control attribution techniques based on tainting.

Another malware analysis system worth mentioning is DRAKVUF by Lengyel et al., a system that introduced new techniques in order to improve scalability, fidelity and stealth [LMP+14]. The authors designed the system with minimal requirement of resources and fast deployment of virtual machines in mind, using copy-on-write techniques to help with the deployment [LMP+14]. Taking the assumption that the VMM is trustworthy, DRAVKUF is built on the open-source
Xen VMM. In order to speed up the deployment of analysis VMs, the system makes use of Xen’s native copy-on-write (CoW) memory interface along with the Linux’s copy-on-write disk capability to create full VM clones [LMP+14]. By doing so they assure scalability, since the allocation of hardware as additional resources is only done when needed. This also assures isolation of the system, because the cloned VMs do not interact with each other, and any exclusive resources previously allocated cannot be accessed. For this system Lengyel et al. used the breakpoint injection technique in which they insert a breakpoint instruction in points of interest in the code. With this technique, the authors can setup the CPU to issue a VMEXIT everytime a breakpoint is executed and use Xen to redirect those events to the control domain, causing a trap in the execution of the code being analysed by the VM [LMP+14]. They were the first one to use this to allow automatic tracing of the entire OS, and therefore assuring fidelity, since it gives a more reliable insight of the kernel and user-land code execution. When addressing the problem of maintaining the system stealthy, Lengyel et al. used automatic execution of samples instead of using agents, allowing the system to remain undetected on the monitored environment. They achieve that by using active VMI via breakpoint injection, which would cause an arbitrary process to start the sample. This way the authors also prevented the introduction of new code into the VM performing the analysis [LMP+14].

The experimental work done on this system used a wide scope of malware samples, with the objective of measuring the overhead and throughput, as well as assessing its effectiveness. The results proved that the system is indeed effective, being able to analyze modern malware while maximizing the number of concurrent analysis sessions, achieving on average an effective memory saving of 62.4% by using copy-on-write memory, with a standard deviation of 7.3%.

2.2 Dynamic Analysis

Dynamic analysis has become more and more popular in security analysis. When talking about dynamic analysis, we must first understand its differences from static analysis. Dynamic analysis is the analysis of the properties of a running program. In static analysis, a program’s source code is examined without executing it to derive properties that hold for all executions. However in dynamic analysis the properties we want to obtain are done by examining the program program [Bal99]. Since it is possible to monitor the code as it executes, allowing for more precise and accurate results, dynamic analyses is preferred in most occasions. This happens due to the fact that dynamic analysis enables the instrumentation of a program, allowing examination or recording certain aspects of its run-time state. This instrumentation can be manipulated in order to obtain the necessary information regarding a particular problem [Bal99].
Some tools have been developed with dynamic analysis as a focus in order to detect malicious behavior. That is the case of Ether, a transparent and external malware analyzer that is based on hardware virtualization extensions such as Intel VT [DRSL08]. The Intel VT extension, short for virtualization technology, has suffered some improvement in the past few years. Since their fourth generation of microarchitectures, Intel has managed to accelerate nested virtualization of VMMs, due to the inclusion of their virtual machine control structure shadowing technology [Int13]. Some of the virtualizations supported, at the moment, include:

- **CPU virtualization** - allows for the Intel CPU to be abstracted to a VM. It also allows for nested virtualization, as well as live migration of different generations of Intel CPUs.

- **Memory virtualization** - enables the memory to be monitored and support abstract isolation.

- **I/O virtualization** - aids in the assignment of VMs to virtual functions, as well as in the processing of offloaded multi-core packets to network adapters.

- **GPU virtualization** - enabled full or shared assignments of the GPU in a virtual machine. It also allow the virtual machine to use the video transcode accelerator engines.

There are two main types of dynamic malware analysis methods: in-the-box and out-of-the-box. When using the in-the-box approach, we install all anti-malware and debugging tools in the same OS as the malware. Despite being vulnerable to rootkits, that can modify the kernel and defeat the analysis, it is still considered efficient, if we take into account that it can use OS context information and directly call the kernel functions to study malware’s behaviors. Regarding the out-of-the-box approach, the installation of the tools is done outside the targeted OS environment, in a secure isolated execution environment [SSW+14]. Ether’s architecture as shown in Figure 2.3 is composed of two main components, the hypervisor and the userspace.

![Ether’s Architecture](image.png)

Figure 2.3: Ether’s Architecture [DRSL08].
Ether works by using the Xen hypervisor as base, which is an open source software that allows access to the lowest layer of the system. The hypervisor is the responsible for the detection of events on the target being analyzed, such as system call execution, instruction execution, memory writes, and context switches. The userspace acts as a controller regulating which processes and events in the guest should be monitored. It is also responsible for handling the logic needed to perform the semantic derivation of analyzed events, such as translations of system calls or displaying the content of the system call arguments [DRSL08]. With these components in mind, Ether uses the VT extension to perform the malware analyzes while still maintaining transparency.

The monitorization of the instructions execution is done by executing one instruction at a time, setting a trap flag to the next instruction, which triggers a debug exception, allowing Ether to control which exceptions reach the analysis target and therefore maintaining it undetected. The memory writes are monitored using shadow page tables and privilege over the guest in handling page faults. Every attempt to perform a memory write on the analysis target, causes a page fault induced by Ether, which will then trap the fault, preventing it from reaching the guest [DRSL08]. The system call execution is monitored using a special register from the modern x86 processors, which allows Ether to cause a page fault at a chosen address. Therefore each fault that is triggered at the chosen address alerts Ether that a system call was done at the analysis target. Regarding context switches, Ether uses VMI every time there is one in order to obtain the name of the next process, and if it corresponds to the one stated by Ether, analysis is enabled. Since everytime there is a context switch to change address space updates the page directory, it is certain that Ether will detect all guest context switches [DRSL08].

This analyzer however is not without limitations, caused by the Intel VT itself, which may allow for its detection by the hypervisor. This can happen since the Intel VT flushes the translation lookaside buffer on every VMExit, and the paging mode must be turned on before entering VMX Root Mode. However after evaluating the tool Dinanburg et al. concluded that Ether does not induce any unconditionally detectable side-effects by completely residing outside of the target OS environment. As a result, malware cannot detect the presence of Ether, maintaining its transparency and defeating the evading obfuscation tools that existed at the time [DRSL08].

Nitro [PSE11] is another system that monitors system calls within the virtual machine, with the help of virtualization extensions. Like Ether, Nitro also manipulates the guest state, but with a difference. When managing the SYSENTER instruction, which is responsible for executing a fast call to a level 0 system procedure or routine, Nitro changes the the 32-bit segment selector
for the privilege level 0 code segment. This will cause the load of an invalid value to the CS register, causing a protection fault, which leads to a VMEXIT. The advantage of Nitro’s approach over Ether is in the number of page faults triggered, which are significantly less than Ether’s, resulting in a better performance. However Ether’s page permission based monitoring offers more flexibility, allowing not only to trap system calls, but also other high-level events [LKE15].

2.3 Trusted Execution Environments

A trusted execution environment (TEE) has been defined as a secure, integrity-protected processing environment, consisting of processing, memory and storage capabilities [EKA14]. It provides isolation from the normal processing environment, where the operating system and its applications run. As such, we can describe the TEE as a way to protect the operating system against attacks, such as exploits, to the system software. This is achieved due to a layer in the TEE that keeps the sensitive data separated from the rest, assuring that it cannot be used by a compromised OS. TrustZone an extension of ARM processors that can virtually separate hardware into two domains referred to as worlds. In this manner it is similar to a hypervisor, but it differs in the way that both worlds potentially have full access to all hardware while it is still possible to restrict access to certain resources for one of the worlds. This higher level of security provided by the TEE has been explored by some embedded hardware technologies such as ARM, described in Section 2.3.1, Intel SGX or Intel vPro, described in Section 2.3.2. However the main focus of this project will be the ARM Trustzone and its uses and advantages, which are described in Section 2.3.3.

2.3.1 ARM Trustzone

The ARM Trustzone is a security extension of the Advanced RISC Machine (ARM) architecture, that allows for an application to be split into two worlds, the normal world and the secure world. As we can see in Figure 2.4, when the software is running in the secure world, there is a separate view from the one running on the normal world. This way, it is possible to keep critical functions and cryptographic credentials hidden from the normal world [Gen16].

In these modes there are independent memory address spaces and different privileges. While code running in the normal world cannot access the secure world address space, code running in the secure world can access the normal world by using as monitor [SRSW14b]. The ARM processor also possesses two ways of handling interruptions, the normal interrupt request (IRQ) and the fast interrupt request (FIQ). The FIQ has higher priority over the IRQ, and as such
whenever a FIQ is being handled all interruptions are disabled. In the Trustzone the FIQ is used by the secure world, while the IRQ by the normal, guaranteeing that the secure world maintains control over the normal one and not the other way around. There are different processor modes in the ARM processors, depending on the architecture version. These modes are the following:

- **User Mode**: The unprivileged mode for normal usage.
- **FIQ Mode**: Triggered whenever a fast interrupt request is accepted.
- **IRQ Mode**: Triggered whenever a normal interrupt request is accepted.
- **Supervisor Mode**: Entered when there is a reset of the CPU or a supervisor call instruction is executed.
- **Abort Mode**: Entered whenever there is an attempt to access an invalid instruction or data memory, causing an abort.
- **Undefined Mode**: Entered whenever there is an undefined instruction exception.
- **System Mode**: Only supported from ARMv4 and above. This mode can only be entered by executing an instruction that explicitly writes to the mode bits of the Current Processor Status Register.
- **Monitor Mode**: Only supported by the Trustzone, and is entered upon world switching. It is the mode responsible for the execution of the secure monitor.

When running TrustZone the processor is divided into two virtual processors, one trusted and one untrusted. If the processor has several cores one could be dedicated to run as trusted while the remaining cores run as untrusted. Each virtual processor have their own memory management unit and, unless one or more cores have been dedicated to the secure world, the...
full capability of the physical processor. Both processors can also run in the first seven modes described above. The eighth mode, the monitor mode, which runs the secure monitor, is only available to the trusted virtual processor [KK14].

The monitor is one of the most important parts of the Trustzone, as it will be responsible for coordinating the accesses to the registers, as well as making sure that the untrusted processor does not access any secure zones. A special bit called the NS bit, is used as an indicator of the world the processor is currently running, which can be used to control the accessibility of the devices to the different worlds. Since only one security mode can be accessed at a time, there is a special instruction called Security Monitor Call (SMC), that allows the processor to perform a world switch. When the monitor call is executed by the CPU, it triggers the secure monitor, which will then perform the context switch into the secure world and allow for data exchange between worlds [SRSW14b].

The monitor keeps a log of all the registers running and upon a call to change world it loads the previously stored registers of the corresponding world. Whenever an application needs to access trusted functions, an SMC instruction must be issued to enter the secure world. Since this can only be done in supervisor mode, the supervision call (SVC) must be done first, except in some cases where the secure world does not have a user mode. The same process needs to be done when returning to the normal world. Figure 2.5 displays how this process works, showing the flow of the data within different privilege levels.

![Figure 2.5: Flow of information in the Trustzone [KK14].](image)

### 2.3.2 Other TEE’s

While the main focus of this project is the ARM Trustzone extension, it is worth noting that there have been other companies that have been exploring the trusted execution technology in order to improve the security of their systems.

**Intel** One of these companies is Intel, that under the name vPro possess a large collection of computer hardware technologies, including Hyperthreading, Turbo Boost 2.0, VT-x, trusted execution technology (TXT), and Intel Active Management Technology. The TXT extensions is
one of the results of merging trusted computing and virtualization. The virtualization support on
the hardware comes from a Trusted Platform Module, which adds trusted and sealed storage and
guarantees platform attestation. The static root of trust for measurement (SRTM) is improved,
since the creation of protected VMs is done dynamically with the help of a dynamic root of
trust for measurement (DRTM). The TXT allows for the platform to initialize a measured
environment anytime after the initial boot. This can happen anytime during runtime, putting
the root of trust into its hardware in a dynamic way [GDB08].

They also started delivering software guard extensions (SGX) in their sixth generation Intel
Core microprocessors based on the Skylake microarchitecture. Software guard extensions can
be described as an hardware isolation mechanism, that allows the exchange of streams of infor-
mation, over a communications link at the same time, in the form of a single complex signal,
between multiple isolated programs. This is done without relying on trusted software. Intel’s
SGX was designed as an extension of the Intel architecture, with the goal of assuring integrity
and privacy to security-sensitive computation in a potentially compromised computer [CD].

GlobalPlatform GlobalPlatform, in their own words, is a non-profit, member driven asso-
ciation which defines and develops specifications to facilitate the secure deployment and man-
agement of multiple applications on secure chip technology [Glo]. It provides standards for the
service providers to be able to develop and deploy services across different devices and channels.
One of their goals is to achieve a TEE standardization, which would allow for a more rapid
technological growth by enabling TEE applications to run in different TEE. This means that
a TEE application would only need to be developed once, and then re-used across any TEE.
This standardization would solve the problem of interoperable TEEs [Det16]. The TEE API, as
specified by GlobalPlatform consists has two components, the Client API and the Internal API.
The Client API describes functions and definitions that allow the exchange of data between
the Rich Execution Environment (REE) and the trusted application (TA). The Internal API
describes functions responsible for the encryption, signing and hashing, as well as functions to
help with the storage of sensitive data and encryption keys. In a trusted application, it is the
Internal API that enables customized implementations of encryption algorithms, by providing
arithmetic functions [KK14].

2.3.3 Uses of the Trustzone

Now that we managed to get a better understanding about the Trustzone, we will dwell further
into the topic and present some of its uses. Through this section we will be presenting a few
systems regarding mobile security and services, and one regarding kernel protection, all of them
designed with the ARM Trustzone in mind.

**Mobile Device Security** As mobile apps start to handle security-sensitive data, smartphones become an attractive target for attacks [SRSW14b]. This is mostly due to the fact that smartphones are used nowadays to execute a vast amount of sensitive apps, such as accessing bank accounts, executing payments, remote authentication for a large amount of services (e.g. facebook, twitter, instagram, rdp and many others) and so on. With this in mind Dongtau Liu et al. proposed VeriUI, an attested login mechanism for mobile devices. This system provides a secure hardware-based environment, using the ARM Trustzone, for password inputs and transmissions, that helps preventing phishing attacks. VeriUI works by providing to the user a secure web browser, where he can safely login. This safety is guaranteed by the system’s login attestation service, which makes use of the isolation provided by the ARM Trustzone. It is in the Trustzone that the service handles all the password related security, being responsible for the generation of the login attestation and output of the password data.

![Figure 2.6: VeriUI Architecture [LC14].](image)

The VeriUI architecture, as shown in Figure 2.6, has a secure OS inside the secure world which will run a service called SecureWebKit responsible for handling the sensitive input. To submit a request to the SecureWebKit, an app passes a message to a TrustZone driver in the rich operating system, which switches to the Secure World and forwards the message to the secure operating system [LC14]. This way the SecureWebKit manages to isolate itself from any calls from the normal world therefore providing sensitive data a secure and trustworthy UI. However this system comes with assumptions that may become serious safety problems. For instance this system offers no safety assurances regarding the rich OS which is running on the Normal World, assuming it’s always secure. If it was compromised, it could result in DDoS attacks, or worse,
since the UI of the secure world could be spoofed by an app running in the Normal World and trick the user into inserting his password and therefore compromising it.

The ARM Trustzone has also been used to develop a Trusted Language Runtime (TLR) for Mobile Applications, which is a system with the purpose of protecting the confidentiality and integrity of .NET mobile applications from OS security breaches [SRSW14b]. This system was designed assuming the existence of external third parties and relying on the correctness of cryptographic primitives and algorithms. The TLR is split in two execution environments, using the Trustzone’s worlds to map them, as we can see in Figure 2.7. On the normal world runs the untrusted, where the OS and most of the applications run. On the secure world runs the trusted, where the TLR code and security sensitive application components run.

![Figure 2.7: High-level Architecture of the TLR](image)

These environments act on the same policies of isolation as the Trustzone itself, with a secure communication channel between them. In the trusted world, TLR provides a language runtime based on .NET Micro Framework. The code of the application running in the TLR can only perform computations, with no access to peripherals, which are all managed by the untrusted OS. The TLR has four main components [SRSW14b]:

- **Trustbox** - A trustbox is an isolated runtime environment that assures that the integrity and confidentiality of code and data is protected. Prevents the OS from tampering or inspecting the code running.

- **Trustlet** - A trustlet is a class within an application designated to run inside a trustbox. The trustlet specifies an interface that defines what data can cross the boundary between the trustbox and the untrusted world.

- **Platform identity** - A unique cryptographic platform identity, provided by any system that uses TLR, is used for platform authentication and for protection of any trusted code and data deployed on the platform (using encryption). The systems uses a public key pair
as the platform ID, and the TLR ensures that the private key is never revealed to any external component.

- **Seal/Unseal data** - The seal primitive encrypts data and binds it to a particular trustlet and platform identity. The unseal primitive yields the data contained in the sealed envelope only if this operation is performed inside a trustbox, and by the trustlet and platform ID originally specified upon seal.

Communication between the TLR-domain and the OS domain is handled by the TLR’s Trust-Zone layer, which is responsible for handling the context switches, the data exchange between worlds and the interrupts. The world switching is done by a secure monitor responsible for detecting the direction of the transition, which will prompt a call to the world descriptor maintained by the Trustzone driver, saving and loading the state accordingly. Finally the processor mode bit is changed.

As referred before, there is data exchange between worlds, which is handled by the secure monitor executing in the secure world, in order to assure access to memory in both worlds. The interrupts must be handled carefully, as it is possible for an interrupt to trigger when on the secure world, since the normal world contains all the I/O devices, and therefore whenever such a situation occurs the secure monitor must do a world switch and handle control to the OS. The TrustZone layer keeps track of the state of all ongoing TLR runtime calls. For each call, the TrustZone drivers maintain a descriptor that contains the call ID, input parameters, output parameters, and the current status of the call. To assure that this state is accessible in both world, the descriptor table is replicated in both, and synchronized when there is a world transition [SRSW14b].

To evaluate the performance of the TLR, Santos et. al, compared TLR applications with the standard .NET versions. Regarding the trustlet code execution, results showed that .NET runtime slightly outperformed the TLR with 57% of the methods executing on average $4.27 \times$ faster, whereas 43% of the methods executed on average $2.34 \times$ faster in the TLR. The primitive’s performance was also evaluated from which was that concluded that the execution time varies depending on the input parameters. From a security point of view the attack surface of the TLR is limited the SMC instruction exposed to the OS and the managed code interface exposed to the trustlets. The SMC interface is relatively narrow, which limits the exposure of code vulnerabilities. However the real problem lies in the managed code interface, which offers a larger attack surface. If a bug in the TLR runtime is found by an attacker, it can be exploited by injecting carefully crafted code sequences in the trustlet code [SRSW14b]. On other hand physical attacks are possible, but were disregarded mainly due to their complexity considering
the ARM architecture. Overall the authors concluded that the TLR protects the integrity and confidentiality of application code and data within the trusted environment [SRSW14b].

Yalew et. al presented three services based on the TrustZone. T2Droid is a service used to securely detect intrusions in an Android device, by leveraging the use of dynamic analysis to perform the detection of malware. This detection mechanism works by using the traces of Android API function calls and kernel system calls performed by an application, in order to verify if its malicious or not. In order to assure security, the authors perform the detection inside the secure world [YMHC17b]. DroidPosture is a posture assessment service for Android that reports posture information for external services. It is used to securely detect intrusions by evaluating the security status of the OS and applications of the mobile device it is running [YMJHC17]. Finally, TruApp is a software authentication service that provides assurance of the authenticity and integrity of applications running on mobile devices. It relies on watermarking and hashing to verify the integrity of the applications [YMM+17].

**Kernel Protection** The OS kernel still poses a real security threat if compromised, despite all the advances made in system security. Most attacker seek to exploit the kernel in order to obtain sensitive data, escalate privileges, disrupt the system or simply take over the machine entirely. From this need to protect the OS kernel, some research has been done about the best way to do it.

One of the proposals regarding the ARM processor is the TrustZone-based Real-time Kernel Protection (TZ-RKP). It is a innovative system that uses the ARM TrustZone secure world to provide real-time protection of the OS kernel [ANS+14]. What this system does is use a secure monitor located in the secure world, that has full control of the kernel’s memory management. This way it is possible for it to analyze all the critical events before they are executed, and decide if they pose a security threat or not, using a system that is both isolated and effective. The TZ-RKP prevents any unauthorized privileged code from being run on the system and also prevents any kernel data from being directly accessible by the user’s processes.

To provide such effective kernel protection the TZ-RKP relies on two features. The first feature, event-driven monitoring, is enforced by depriving the kernel from its ability to control certain critical functions. Hence, it is forced to route requests to perform these functions through TZ-RKP. The removed system control instructions are the ones that allow the normal world to control security critical system state, such as defining the location of memory translation tables and exception handlers. Therefore, the only way to execute these instructions is through emulating them from the secure world. The second feature provided by TZ-RKP is memory protection, which ensures that the target kernel cannot be modified. It also prevents kernel
code injection and return-to-user attacks [ANS+14].

![TZ-RKP Components](image)

Figure 2.8: TZ-RKP Components [ANS+14].

According to the authors, any processor that supports the ARM Trustzone can run the TZ-RKP. This system’s architecture is pretty simple, as seen in Figure 2.8. The kernel runs in the normal world while the TZ-RKP runs in the secure world. Whenever the kernel needs to execute it will perform a request to the TZ-RKP, asking to emulate the needed control instructions that change the system state and update the OS memory translation tables. The way this system handles the emulation of the control instructions is by removing them from the normal world and replacing them with hooks, which allows them to be emulated. Regarding the OS memory translation tables, the TZ-RKP traps updates to the memory translation tables so that it can always keep an up-to-date information about the virtual-to-physical mapping of the normal world memory and the access permission of each memory page [ANS+14].

Through experimental evaluation, it was concluded that the TZ-RKP presented a low overhead regarding application load times, power consumption and device boot-up time. It had however a great variation of the number of cycles regarding the switches between the normal and secure world, due to the way trustzone memory protection is implemented on each processor, which requires further examination of the system being used. Nevertheless the TZ-RKP provides real world systems an effective security solution.

2.4 Introspection Libraries

The definition for virtual machine introspection was first introduced in 2003 by Garfinkel and Rosenblum [GR03]. It is focused on security and it was proposed to address the dichotomy between existing intrusion detection systems. As such it is of no surprise that on the years that followed its introduction, several libraries were created with focus on VMI. This section introduces three of those libraries, starting with the LibVMI, which is the one our work is based on. The remaining libraries are IntroLib, that focus on tracing user-level library calls made by malware, and Virtual Introspection for Xen, a set of tools made by Xen with focus on the
introspection for forensic purposes.

### 2.4.1 LibVMI

LibVMI [Pay] is a C library with Python bindings. It is used for VMI, and makes monitoring the low-level details of a running virtual machine easier by viewing its memory, trapping on hardware events, and accessing the vCPU registers. All these functionalities work with VMs running on QEMU, Xen or KVM. It also works with static snapshots that have been saved to a file.

Figure 2.9 shows the LibVMI architecture. As we can see, the LibVMI sits between monitor applications and the VMM specific low level semantic library. Monitor applications are responsible for assuring that the applications are executed and perform according to what is expected within their scope. LibVMI also includes service routings regarding memory, disk, network and CPU register accesses.

![Architecture of the LibVMI](image)

Figure 2.9: Architecture of the LibVMI [XLXJ12].

Overall the LibVMI API can be divided in two categories, the init and destroy management API and the utility API. Regarding the init and destroy API, there is a function called vmi_init, which is responsible for access initialization to a virtual machine with a given name. Due to its overhead, it should only be called once on a virtual machine, and eventually should end with a call to vmi_destroy. This utility API contains the API’s for memory accesses, disk accesses, CPU registers accesses and others. LibVMI also possesses a driver interface, used to abstract common functions of the VMM and Guest OS. To initialize the LibVMI, we must first obtain an instance of the VMI and initialize it with zero values. Then it selects the VMM driver, and loads the Guest OS configuration, along with its memory size and layout, completing the initialization.
Once this procedure is concluded, the applications are ready to use the utility API.

Figure 2.10 shows a sample of the LibVMI API. As we can see there are several functions available, that range from reads and write into memory, to more complex functions that deal with events. Other important functions include those that read the current CPU registers, pause the VM, and resume the VM. There are also a series of functions for managing the various LibVMI caches.

```c
status_t vmi_init (vmi_instance_t *vmi, uint32_t flags, char *name)
status_t vmi_destroy (vmi_instance_t vmi)
addr_t vmi_translate_kv2p (vmi_instance_t vmi, addr_t vaddr)
addr_t vmi_translate_uv2p (vmi_instance_t vmi, addr_t vaddr, int pid)
addr_t vmi_translate_ksym2v (vmi_instance_t vmi, char *symbol)
addr_t vmi_pid_to_dtb (vmi_instance_t vmi, int pid)
size_t vmi_read_ksym (vmi_instance_t vmi, char *sym, void *buf, size_t count)
size_t vmi_read_va (vmi_instance_t vmi, addr_t vaddr, int pid, void *buf, size_t count)
size_t vmi_read_pa (vmi_instance_t vmi, addr_t paddr, void *buf, size_t count)
size_t vmi_write_ksym (vmi_instance_t vmi, char *sym, void *buf, size_t count)
size_t vmi_write_va (vmi_instance_t vmi, addr_t vaddr, int pid, void *buf, size_t count)
size_t vmi_write_pa (vmi_instance_t vmi, addr_t paddr, void *buf, size_t count)
void vmi_print_hex (unsigned char *data, unsigned long length)
unsigned long vmi_get_memsize (vmi_instance_t vmi)
status_t vmi_get_vcpureg (vmi_instance_t vmi, reg_t *value, registers_t reg, unsigned long vcpu)
status_t vmi_pause_vm (vmi_instance_t vmi)
status_t vmi_resume_vm (vmi_instance_t vmi)
void vmi_v2pcache_flush (vmi_instance_t vmi)
status_t vmi_register_event (vmi_instance_t vmi, vmi_event_t *event)
status_t vmi_clear_event (vmi_instance_t vmi, vmi_event_t *event)
vmi_event_t* vmi_get_reg_event (vmi_instance_t vmi, registers_t reg)
```

Figure 2.10: Sample from the LibVMI API.

### 2.4.2 Virtual Introspection for Xen (VIX)

Xen, an open source project, created the Virtual Introspection for Xen, a suite of tools focused on digital forensics [HN08]. Although it was developed by Xen, these tools can be applied to other virtualization platforms by implementing the details of memory access, which are different. For other virtualization platforms, the VMM enforces the separation between virtual machines. Although it enables virtual introspection, the full access to the resources of all the virtual machines that it manages, is itself a risk, meaning that a compromised VMM could mean the compromise of all the virtual machines on the system.

This suite of tools allows live analysis of an unprivileged Xen VM, known as DomU, from a privileged Dom0 VM, without the need of modifications. VIX contains a set of tools to mimic Unix command line utilities, along with a library of common functions. To use VIX, first the
VM is paused, then there is a process of data acquisition in order to perform the requested functions. Once the data acquisition is finished the VM state is resumed. By pausing the VM, it is assured that its state does not change during the data acquisition. The Xen Control library allows DomU systems to be operated on by a Dom0 system, giving it control over the the pauses of the DomU VM’s and mapping the memory pages of a target VM into the Dom0 address space.

To access a memory address in a Linux VM, the appropriate physical frame in memory needs to be found. That is done by consulting the page tables to determine the corresponding physical frame number. Due to the virtualization there is also the need to translate the physical frame numbers from the perspective of the VM into the physical frame numbers for the underlying physical hardware. This process is handled via a call to the Xen Control library, through the operation that maps a given physical machine frame number into Dom0 memory space. When this process is complete, the Dom0 application can perform the virtual introspection on the mapped page.

2.4.3 IntroLib

Deng et. al. presented IntroLib, a practical tool that traces user-level library calls made by malware with low overhead and high transparency [DXZJ12]. Their main goal is to detect user space malware, and as such they want to avoid malware’s anti-analysis logic such as the detection of emulation or library API hooking. The IntroLib is based on a hypervisor that utilizes hardware virtualization to elevate its transparency to malware.

The key aspects of the design of the IntroLib rely on:

• Interception of control-flow transitions between malware and library functions

• Identifying the memory layout the malware process

• Logging

• Improving transparency to malware

To intercept the control-flow transitions, the authors set a shadow page table to work as a barrier in memory. Shadow page tables are used by the KVM hypervisor to keep track of the state in which the guest assumes its page tables should be. Normally the hypervisor only uses one shadow page table per guest, but IntroLib maintains two mutually exclusive pages for each guest. That way they want to trigger events when there is a control-flow transition between the malware and the library functions, that would other way be captured by the hypervisor.
Identifying the memory layout the malware process is needed to assure the separation between the malware and the library code. Since malware can load libraries during runtime, data pages can become either malware code or library code pages. To identify the data pages, the author came up with lazy identification. With lazy identification, an area of code pages is not identified until an instruction-fetch from one of its pages happens. So when the mapping of the area happens, all the pages are assumed as data pages and set as non-executable in the two shadow page tables. That way, when the instruction-fetch from one of the pages happens a page fault is triggered and captured by the hypervisor.

The source and destination addresses from the control-flow transitions are read from the Last Branch Record (LBR) stack, which is a circular stack that stores information about the recent branches taken by the processor. From the memory layout, the IntroLib will use the source and destination addresses to locate the corresponding code areas. If the code area is associated with a library, it will parse the library file to obtain the functions. If the address is the entry address of a library function, then the transition is a library call. The caller address, the called library name, function name and values of arguments are all logged and for each call it logs, IntroLib will read and record its return address from stack.

To be more transparent to malware, IntroLib avoids timing attacks by manipulating the time seen by the guest, so that the existing time discrepancy caused by the use of the library can not be detected by the malware. Another technique used to increase transparency is the masking of the use of the LBR, which is achieved by using a shadow LBR stack.

Summary

This chapter described the related work regarding introspection and different kinds of analysis that can be done on a system. We started by providing an overview over virtual machine introspection, going into details on its advantages and disadvantages, as well as into the two types of hypervisors used: native and hosted.

The following parts introduce the concepts of static and dynamic VMI, with the first being more focused on forensics and the later more related to our work, which focuses on the analysis of a running system. We explored several techniques used to solve the semantic gap when dealing with the reconstruction of kernel data, and examined several systems that relied on instrumentation of instructions to perform an analysis. Due to the importance of dynamic analysis, we delved further into the subject, analyzing several systems designed for different virtualization platforms and technologies.
We then describe two hardware-based isolation technologies which can be applied to perform a secure introspection. ARM TrustZone and Intel SGX are security extensions to processors which allow the use of one secure hardware-managed mode for processing sensitive data. We also give a brief overview over Global Platform, which is dedicated to create a standard for secure chip technologies to facilitate the secure deployment and management of multiple applications.

Due to our choice of using ARM TrustZone as our hardware-based isolation technology, we decided to focus on the uses of that technology, by exploring systems designed for mobile device security and kernel protection. The later inspired the use case designed to check for modifications to the kernel in our system.

We end this chapter by discussing three introspection libraries designed with different analysis purposes, from which we picked LibVMI as the basis of our work. The next chapter will introduce the design details of our system, mainly our introspection library.
Chapter 3

ITZ Library Design

This chapter describes the design of our system, mainly the ITZ library, as well as the tools used during development. In order to securely introspect the memory, our system will leverage the security primitives supported by ARM TrustZone. This technology enforces hardware-level isolation between domains with different execution privileges.

In this chapter, we start by presenting a few use cases for the library (Section 3.1) and then describing the framework used to design our system (Section 3.2). Afterwards we describe the assumptions and threat model taken into account when designing ITZ (Section 3.3). We then describe the architecture of our system by presenting how the library interacts with the different components, as well as how the ITZ API is structured along with the process of obtaining contents from memory (Sections 3.4 and 3.5, respectively). The interactions with the normal world should be done on the secure world in order to assure that the data has not been compromised. We then conclude this chapter by discussing how the interaction between worlds is done (Section 3.6).

3.1 Use Cases

This section presents briefly three use cases for ITZ for the reader to better grasp what it is useful for.

A company adopted BYOD (Bring Your Own Device) and lets its employees use their own smartphones to access an internal database (running in some private cloud) using an app provided by the company. The app runs in the normal world, as any other app. The database is critical so it is not reasonable to allow access from arbitrary devices. Therefore, when the app is executed it starts by calling a service that runs in the secure world, installed by the company, which was implemented using ITZ. That service uses VMI to check the security status of the
normal world. If it considers there are no security issues, it returns to the app a certificate that
the app sends to the company’s cloud. The database only returns internal data if the certificate
is valid.

In the second scenario, an organization that runs a critical business (a bank, the military, a
3-letter agency) has its own tablets for employees to do their job in the field. The tablets need to
have Internet connection so the organization is concerned about their security status. Therefore,
periodically (e.g., every hour) a service implemented using ITZ is triggered to access the status
of the normal world. This triggering can be done using a hardware clock that generates an
interruption [YMHC17a]. Then, the service checks the status and if the level of risk is high it
puts the device in a locked status, that can only be unlocked by taking it to the organization’s
facilities.

In the third use case, a company is concerned about the security of the apps from online
markets that it recommends employees to use in their devices (e.g., the email client), with good
reasons for that [ZWZJ12, SCT12]. Therefore, it implements a service based on ITZ to do
introspection and analyze these apps. For that purpose, its engineers use a set of test devices
that run the service in the secure world. These engineers install these apps and use the service
to understand if there are security issues.

The benefit of ITZ in the three use cases is to simplify the implementation of the service
that, with ITZ, do not have to implement introspection from scratch.

3.2 Genode OS Framework

In order to provide more insight into our work, we will first start by discussing the framework
used. The choice of the Genode OS framework came from the easiness to use, as well as from
the several resources available for new developers [Ste].

Nowadays kernels, such as the Linux kernel, are complex and contain several components of
software ranging from information storage to resource management and control of user processes.
Those components have high functional requirements causing a typical kernel to have millions
of lines of code [KH07]. That factor and the different range of hardware available has caused
the Linux kernel to grow quickly over the years, which can be seen in Table 3.1.

With this large amount of code, it is hard to avoid security leaks and bugs, which can lead
to a compromised system. That is where the Genode OS framework comes in. By providing
the tools to create our custom microkernel it is possible to strip the system of all non-needed
components, shrinking the size of the kernel. But Genode goes even further, decomposing the
system policy by imposing a strict organizational structure onto each part of the system.
<table>
<thead>
<tr>
<th>Linux Kernel Version</th>
<th>Lines of Code (LOC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>150 867</td>
</tr>
<tr>
<td>2.0.1</td>
<td>632 940</td>
</tr>
<tr>
<td>2.2.12</td>
<td>1 671 028</td>
</tr>
<tr>
<td>2.6.7</td>
<td>5 004 394</td>
</tr>
<tr>
<td>3.0</td>
<td>11 914 002</td>
</tr>
<tr>
<td>4.0.1</td>
<td>15 525 472</td>
</tr>
<tr>
<td>4.13</td>
<td>19 784 513</td>
</tr>
</tbody>
</table>

Table 3.1: Evolution of the Linux Kernel in LOC [Lö].

Genode Microkernel   Developing a microkernel from scratch to support the ARM TrustZone and the necessary requirements to develop our system would be a very long and complex task. This would require expertise of low-level details of the platform, as well as a good amount of knowledge of machine language. Due to this we chose to adapt the Genode microkernel, creating our own custom version of it. The codebase of the microkernel is of more or less 20,000 lines of code and it is compatible with our development platform. This TrustZone compatible microkernel is designated base-hw microkernel, and provides several components useful for the development of our work, from which the most important is the TrustZone Virtual Machine Monitor (TZ VMM). The TZ VMM acts as a secure world manager and is responsible for leveraging the Multi-Master Multi-Memory Interface (M4IF), which is the component responsible for the protection of the memory. The microkernel also includes a client application responsible for managing a rich OS running in the normal world, which in our case is a paravirtualized Linux kernel. A paravirtualized Linux kernel is aware that is virtualized, and has functions to make calls to an hypervisor. In our case, it allows the kernel to use the SMC call to perform a world switch. Due to the simplicity of the basic microkernel, we needed to extend it, by porting some necessary libraries, such as libc, and enabling a shared memory buffer for communication between worlds. This extension also includes support for our introspection library which had to be designed taking into account the features and limitations provided by the microkernel.

3.3 Assumptions and Threat Model

In order to justify our design and implementation, we have to describe the assumptions and threat model considered for our system. As mentioned before, our work aims at providing a functional library for introspecting the normal world of a device from a secure execution environment.

We assume that the hardware is correct, i.e., that all TrustZone security features supported by the processor are correctly implemented and cannot be compromised or circumvented by an attacker. We assume that the normal world runs an untrusted kernel and is used by untrusted
users, which however do not have access to the secure world resources and configuration. We also assume to have the source code of the Linux kernel.

We accept that an attacker may trigger a SMC call (a call to the secure world), and attempt to pass fake data onto the secure world. The attacker may do this call repeatedly to cause a local denial-of-service.

We are primarily concerned with potential memory modifications resulting from software exploits to normal world applications. In order to prevent attacks caused by external exploits on the system, the introspection of the memory is done from the secure world. The introspection does not rely on any information passed explicitly by the normal world, accessing the memory directly.

The Trusted Computing Base (TCB) [Nat83] of our system comprises the hardware platform (ARM processor with TrustZone extension and other chips), the secure world microkernel, the ITZ library, and the introspection tool(s) (that run in the secure world).

### 3.4 Architecture

The proposed architecture has its components divided among the two worlds as shown in Figure 3.1. The ITZ library is placed in the secure world. The figure shows also an introspection tool (there could be more), that uses the library functions to do some job based on VMI.

![Figure 3.1: System Architecture.](image)

In the normal world there is a Linux kernel (or another OS) that is extended with a module to support the shared memory buffer necessary for the two worlds to exchange data. This is done using direct memory access (DMA), and passing the address and size through certain registers.
This memory buffer is only used by the secure world to send data to the normal world; whenever the secure world needs information from the normal world, it accesses the memory directly, since otherwise the memory could be compromised.

In the secure world there is a microkernel that provides basic resource management functions (memory, CPU, I/O). On top of that microkernel, there is a virtual-machine monitor (VMM) that runs in user mode. This VMM mainly manages the shared memory buffer and communication with the normal world.

3.5 ITZ Library

The goal is to design a library called ITZ capable of performing introspection on the normal world, in order to obtain the content stored in memory (non-persistent or persistent). Our library is inspired on LibVMI [Pay], a VMI library for VMs working on QEMU, Xen or KVM. LibVMI is a C library with Python bindings. We first tried to port LibVMI to run in the secure world, but this was unfeasible without running the Linux kernel and several libraries in that environment. This would lead to a large TCB and make the TEE more prone to vulnerabilities.

The normal world of the device is responsible for maintaining the Linux kernel and the applications running. We consider this world insecure and easily compromised, and therefore it cannot inspect itself. Taking this into account, we adapted the idea of VMI, which is a technique used to inspect and analyze the code running on a given virtual machine. As we can see in Figure 3.2, the way we do the introspection is by using our library running in the secure world. As we can have the normal world compromised, the memory access is always done directly from the secure world. This is possible due to the properties of the TrustZone, which prevent code running in the normal world from accessing the secure world address space, but allow code running in the secure world to access the normal world address space.

![Figure 3.2: Introspecting the Normal World Memory.](image)
ITZ provides an API divided in several classes of functions. It provides functions to read and write from both virtual addresses and physical addresses, as well as from the symbol table used by the kernel. These functions access directly the memory of the normal world from the safety of the secure environment.

When using memory addresses in a normal application or in the kernel we are dealing with virtual memory addresses that are translated in hardware via the Memory Management Unit (MMU) to physical memory addresses. Therefore, the library provides also translation functions to perform the translation from virtual addresses to physical addresses and vice-versa.

Additionally we also support print functions responsible for printing out the hexadecimal and ASCII of a chunk of bytes located in a given address. We can describe these functions as memory dumpers, that dump the content of a given address to the terminal or a file. Their output is similar to the output obtained with Unix’s od command line tool. The od command dumps a file in octal, decimal, and other formats.

Figure 3.3 shows a sample of the API of the ITZ library, with examples for read, write, print and other functions.

```c
//Destroy instance (free memory and close handles)
status_t vmi_destroy(Vm * _vm);

//Read 8 bits from a given physical address
status_t vmi_read_8_pa(Vm *_vm, uint64_t paddr, uint8_t * value);

//Write 8 bits to memory, given a virtual address
status_t vmi_write_8_va(Vm *_vm, uint64_t vaddr, int32_t pid, uint8_t * value);

//Translate virtual address to physical address
uint64_t vmi_translate_uv2p(Vm *_vm, uint64_t vaddr);

//Print hex and ascii versions of a chunk of bytes from a kernel symbol
void vmi_print_hex_ksym(Vm * _vm, char * sym, size_t length);

// Get memory size of the normal world (i.e., the maximum physical address that ITZ can access)
uint64_t vmi_get_memsize(Vm * _vm);

//Get the memory offset associated with the given offset_name
uint64_t vmi_get_offset(Vm * _vm, char * offset_name);
```

Figure 3.3: Sample of the ITZ Library API.

3.5.1 Comparing ITZ to LibVMI

We now compare the functions of our library with those of LibVMI. Our library has an interface similar to LibVMI’s to reduce the effort necessary to port tools based on LibVMI to work with
ITZ. Table 3.2 shows that we included all the functions we considered relevant and some stubs to make the API similar to LibVMI’s. We did not include some Windows-only or LibVMI specific functions. The event and single step functions available in LibVMI were not implemented in ITZ due to limitations of the environment, which does not allow for interruptions of the normal world (e.g., system calls) to be handled by the secure world. Even if this was possible, it would severely impact the performance of the device due to constant world-switches. One possible workaround would be to insert breakpoints in the software being monitored to trigger a SMC call and perform a world switch. However, this approach would be vulnerable to malicious software that could remove these instructions, besides the overhead of changing worlds.

<table>
<thead>
<tr>
<th>LibVMI functions</th>
<th>ITZ functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization functions</td>
<td>Stubs implemented (not used)</td>
</tr>
<tr>
<td>Destroy functions</td>
<td>Stubs implemented (not used)</td>
</tr>
<tr>
<td>Translation function</td>
<td>Implemented (excluding Windows)</td>
</tr>
<tr>
<td>Read functions</td>
<td>Implemented (excluding Windows)</td>
</tr>
<tr>
<td>Write functions</td>
<td>Implemented (excluding Windows)</td>
</tr>
<tr>
<td>Print functions</td>
<td>Implemented</td>
</tr>
<tr>
<td>Get and Set functions</td>
<td>Partially implemented (some not used)</td>
</tr>
<tr>
<td>Pause and Resume functions</td>
<td>Stubs implemented (not used)</td>
</tr>
<tr>
<td>Cache functions</td>
<td>Partially implemented (some not used)</td>
</tr>
<tr>
<td>Event functions</td>
<td>Not implemented</td>
</tr>
<tr>
<td>Single Step functions</td>
<td>Not implemented</td>
</tr>
</tbody>
</table>

Table 3.2: Comparing LibVMI to ITZ.

3.5.2 Main Functions

Read from Memory Our library provides functions to read from both virtual addresses and physical addresses, as well as from the symbol table used by the kernel. These functions access directly the memory of the normal world from the safety of the secure environment. Our functions make use of some of the modules available on Genode to facilitate the memory access.

In Genode, the RAM service provides access to physical memory. The RAM-session client can allocate memory blocks in the form of dataspace objects. A dataspace represents a contiguous physical address space region with an arbitrary size. These objects can be used to communicate with other processes and make the dataspace’s physical-memory region accessible from these processes. Due to these proprieties it is possible to use our read functions to read directly the values in the RAM.

Another important component is the memory mapped input/output (I/O) service that provides a dataspace abstraction for non-memory parts of the physical address space, such as memory-mapped I/O regions or BIOS areas. In contrast to a memory block that is used for storing information, where its physical location in memory is not important, a non-memory object has a special semantic attached to its location within the physical address space.
Read count bytes from memory located at the physical address paddr and stores the output in buf.

```c
size_t vmi_read_pa(Vm *vm, uint64_t paddr, void *buf, size_t count);
```

Read X bits from a given physical address

```c
status_t vmi_read_8_pa(Vm *vm, uint64_t paddr, uint8_t *value);
status_t vmi_read_16_pa(Vm *vm, uint64_t paddr, uint16_t *value);
status_t vmi_read_32_pa(Vm *vm, uint64_t paddr, uint32_t *value);
status_t vmi_read_64_pa(Vm *vm, uint64_t paddr, uint64_t *value);
```

Figure 3.4: ITZ Library API for Reading from Physical Addresses.

In Figure 3.4 we can see the available API functions for reading from physical addresses, while in Figure 3.5 we have the API functions for the virtual addresses. These last ones differ from the first ones, because they require every virtual address to be translated into its corresponding physical one before accessing its content.

Write count bytes from memory located at the virtual address vaddr and stores the output in buf.

```c
size_t vmi_read_addr_va(Vm *vm, uint64_t vaddr, int32_t pid, void *buf, size_t count);
```

Read X bits from a given virtual address

```c
status_t vmi_read_8_va(Vm *vm, uint64_t vaddr, int32_t pid, uint8_t *value);
status_t vmi_read_16_va(Vm *vm, uint64_t vaddr, int32_t pid, uint16_t *value);
status_t vmi_read_32_va(Vm *vm, uint64_t vaddr, int32_t pid, uint32_t *value);
status_t vmi_read_64_va(Vm *vm, uint64_t vaddr, int32_t pid, uint64_t *value);
```

Figure 3.5: ITZ Library API for Reading from Virtual Addresses.

**Write to Memory**  Similarly to the reading functions, the writes will also use the dataspace objects of the RAM session, but instead of obtaining the content stored, they overwrite them. The handling of the virtual addresses is also similar, requiring translation to physical addresses before executing any write function. In Figure 3.6 we can see the available API functions for writing.

**Translation Functions**  When using memory addresses in a normal application or in the kernel we are dealing with virtual memory addresses that are translated in hardware via the Memory Management Unit (MMU) to physical memory addresses. With that in mind we added some auxiliary functions to help accessing this type of memory from the secure world. These helper functions are used to decode register values of the normal world that potentially contain virtual addresses to physical ones. This allows us to perform the translation from a virtual address to a physical address and vice-versa. There is an additional function to translate the kernel symbols, which is done by using the System.map file generated upon compiling the Linux
//Write X bits to memory, given a virtual address.
status_t vmi_write_8_va(Vm *_vm, uint64_t vaddr, int32_t pid, uint8_t * value);
status_t vmi_write_16_va(Vm *_vm, uint64_t vaddr, int32_t pid, uint16_t * value);
status_t vmi_write_32_va(Vm *_vm, uint64_t vaddr, int32_t pid, uint32_t * value);
status_t vmi_write_64_va(Vm *_vm, uint64_t vaddr, int32_t pid, uint64_t * value);

//Write count bytes to memory located at the physical address paddr from buf.
size_t vmi_write_pa(Vm *_vm, uint64_t paddr, void * buf, size_t count);

//Write X bits to memory, given a physical address.
status_t vmi_write_8_pa(Vm *_vm, uint64_t paddr, uint8_t * value);
status_t vmi_write_16_pa(Vm *_vm, uint64_t paddr, uint16_t * value);
status_t vmi_write_32_pa(Vm *_vm, uint64_t paddr, uint32_t * value);
status_t vmi_write_64_pa(Vm *_vm, uint64_t paddr, uint64_t * value);

Figure 3.6: Sample of the ITZ Library API for Writing Functions.

kernel running on the normal world. This file represents the symbol table used by the kernel, along with their corresponding virtual addresses in memory. To assure the reliability of the file, we obtain it upon compiling the kernel and feed it directly into our library in the secure world.

In Figure 3.7 we can see the available API functions for translating.

//Translate kernel symbol to virtual address
uint64_t vmi_translate_ksym2v(Vm *_vm, const char * symbol);
//Translate physical address to virtual address
uint64_t vmi_translate_p2uv(Vm *_vm, uint64_t phy_addr);
//Translate virtual address to physical address
uint64_t vmi_translate_uv2p(Vm *_vm, uint64_t vaddr);

Figure 3.7: Sample of the ITZ Library API for Translating Functions.

**Hexadecimal and ASCII Printing** These functions are responsible for printing out the hexadecimal and ASCII of a chunk of bytes located in a given address. We can describe these functions as memory dumpers, that dump the content of a given address to the terminal. Additionally they can also be used to dump the content into a file.

Their output is similar to the output obtained with `od` on the Unix command line. The `od` command dumps a file in octal, decimal, and other formats. With these functions we get the hexadecimal with the additional ASCII information on the right side of the display. In Figure 3.8 we can see the available API functions for printing.

//Prints out the hex and ascii version of a chunk of bytes from virtual address
void vmi_print_hex_va(Vm *_vm, uint64_t vaddr, int32_t pid, size_t length);
//Prints out the hex and ascii version of a chunk of bytes from physical address
void vmi_print_hex_pa(Vm *_vm, uint64_t paddr, size_t length);

Figure 3.8: ITZ Library API for Printing Functions.
3.6 Interaction Between Worlds

Although both the library and introspection tools run in the secure world, the usual way for introspection to be started is by an application from the normal world to call the secure world. In order to support that, we have developed a way to send arguments from the normal world to the secure world, which allows us to control a memory region to perform communication between worlds.

Whenever the normal world requests to switch worlds, the CPU state is saved in order to be restored later. The component responsible for handling and restoring the state of the normal world is the TZ VMM. This component saves the processor state of the normal world, allowing for the secure kernel to access it. As such it is possible, from the secure world, to access CPU registers, the stack pointer and the random-access memory (RAM) pointer. This last, is a special pointer used to indicate the top of the normal world’s RAM. When control is given back to the normal world, TZ VMM restores the previously saved CPU state.

Using the modified TZ VMM, we have designed a way to send the desired arguments through worlds, by relying on the use of the CPU registers. In the normal world, whenever a request to switch worlds is issued, the OS can push the registers to the stack, and then write the arguments onto them. Afterwards on the secure world, by using the saved state of the normal world’s processor, it is possible to extract the data that was pushed. In order to send data back into the normal world, the secure world is allowed to modify the saved CPU state of the normal world, by replacing the values stored in the registers. The normal world OS, after regaining control, can then access them and extract the new data. Once this operation is complete, the register values are restored, and the CPU state is resumed to what it was before the world switch.

This method does not allow sending large amounts of data in either direction, but only some bytes. However, these bytes are enough to indicate the memory address and the size of the shared memory buffer, which is then used to pass arbitrary amounts of data. Specifically we use register $r2$ for the address and $r3$ for the size. To solve the limitations, the idea was the creation of a shared memory buffer, which could be accessed in both worlds.
In the normal world OS, the shared memory buffer is created, and it needs to send over the necessary information for access onto the TZ VMM. As such, the physical address corresponding to the start of the shared memory and the size of the allocated memory region are sent over to the secure world through the CPU registers. With this information it is possible in the secure world to point to the address corresponding to the starting position of the shared memory buffer. It is worth mentioning, that this memory buffer should only be used for non-sensitive data, because its contents can be compromised. It can also be used for testing purposes, since it is a controlled area of memory for which we know its address and size.

Figure 3.9 illustrates how the data is exchanged through the shared memory buffer using Genode and some of the created functions. As we can see, it all starts with the mapping of the memory in the secure world, followed by the virtual machine initialization. On the normal world, the buffer is created with a given size, and afterwards the address and size are passed to the secure world upon making an SMC call, through the CPU registers. The address and size of
the buffer are then retrieved by the TZ VMM. To write to the secure world, the normal world OS, simply copies into the shared buffer the content, by providing its address and size of the content. Then the SMC call is triggered passing this information to the TZ VMM. A similar process happens in the secure world to write into memory, for the normal world to read.

**Summary**

In this chapter we described the design of system, focusing on ITZ, our introspection library for the ARM TrustZone. We gave an overview of the Genode OS Framework, which is the framework used to create our own modified microkernel, that we have running in the secure world and runs our library. We then explained the assumptions and threat model in which our work is based. Afterwards we presented the ITZ’s architecture by describing the several components that are part of the normal and secure worlds of our system. We explained the design choices, and the reasons behind the need for building a new library from scratch based on LibVMI. Finally we give an overview over the ITZ API, focusing on the several classes of function provided, ending this chapter with an explanation on the shared memory strategy adopted for communication between worlds. Next we present Chapter 4, which describes the implementation details behind the implemented ITZ library.
Chapter 4

Implementation

This chapter describes a prototype implementation of ITZ, a TrustZone compatible library for introspection. We implemented a prototype of ITZ for the Freescale NXP i.MX53 Quick Start Board (QSB) and some of the implementation details described in this chapter are specific to this development board. Next, we describe our system configuration (Section 4.1) and the implementation challenges for our prototype (Section 4.2). Then we continue this chapter by presenting the secure world components, providing details on the introspection functions provided by the ITZ library and the hashing functions used in our use case application, as well as a comparison between our implementation and LibVMI (Section 4.3). Finally we finish by providing a brief overview over the different implementation attempts that were made prior to deciding upon the one here presented (Section 4.4).

4.1 System Configuration

The i.MX53 QSB has a mechanism to assure that memory regions are protected to prevent unauthorized access from the normal world. This works as part of the DDR memory controller, through the Multi-Master Multi-Memory Interface (M4IF). The M4IF enables the masking of DDR RAM resources and guarantees that a configurable range of memory is protected and used exclusively by the secure world.

The i.MX53 QSB has 1GB of RAM split into two memory banks, RAM 0 and RAM 1. Each memory bank has 512MB of memory that can be configured and used. For our work, and due to the limitations of the M4IF, that only allows for up to 256MB of memory to be protected in each bank, we choose to protect the initial 256MB of the first memory bank.

This process is done at boot time, and assures that 256MB of memory are reserved for our secure world, while the rest of the bank is used for the normal world OS. In Figure 4.1 we can
see the memory layout used in our device. Despite the fact that we only use the first memory bank, we have not noticed any limitations to our system. However it is possible to make the necessary changes to use both memory banks, thus leaving us with 512MB of protected memory, but it would require more extensive modification on the base-hw micro-kernel running and on the TZ VMM.

![Memory Layout Diagram](image_url)

Figure 4.1: i.MX53 QSB Memory Layout.

In the normal world, we run a custom version of the Linux kernel. It is based on the kernel version 2.6.35.3 and we have modified it to use it alongside the i.MX53 QSB. The OS kernel has some additional system calls that allow to trigger a world switch when needed. Moreover, an additional kernel module has been added to create a shared memory buffer using DMA, allowing for data exchange. There is an additional module that allows obtaining the Linux offsets necessary for introspection purpose, such as the offset of the symbol table. Another implemented module is based on an existing one for LibVMI. It is used for obtaining the linux offsets necessary for the introspection, such as the pid or name offsets. Additional information on how to configure Genode and setup the workspace can be found in Appendix A and B respectively.

4.1.1 World Switch

The kernel running in the secure world should be able to answer a world-switch request from the operating system in the normal world. In order to do so, the secure kernel should be configured. To use the monitor mode, the kernel needs to trap and handle the world-switch requests, which is done by using the Secure Configuration Register (SCR). This special register allows the CPU to trigger an exception in monitor mode.

The TZ VMM works as a gatekeeper to both worlds, and since it is an important security component, the entries received by the normal world to access the monitor are controlled. This way, the normal world is only able to request a world switch by triggering an interruption or
by calling the SMC instruction. The secure world however is able to enter the monitor mode directly, by writing to the Current Program Status Register (CPSR), and then accessing the SCR, where the NS bit can be changed.

Triggering the world-switch from the normal world can be a security issue, due to possibility of the operating system being compromised, which can lead to a potential attack.

![World Switch using Genode](image)

Figure 4.2: World Switch using Genode.

The microkernel we are using in our work, supports a basic world-switch mechanism, where the TZ VMM is not directly responsible for switching from the secure world to the normal world. That way the change of the NS bit is manipulated by the secure world, allowing for the assembler routine to be simpler. This routine is responsible for saving the current state the current world’s CPU and restoring the state of the world it wants to switch into.

Since we are using a simple switching mechanism, the routine only saves the current CPU state as the normal world’s state and restores the secure kernel context. The reverse operations are executed to switch from secure world to normal world, with the particularity of being executed outside the monitor, which means that the kernel context must always be saved and stored in a secure memory region. Only afterwards the previously saved normal world state can be restored and the world switch triggered, by changing the NS bit in the SCR to the value 1. In Figure 4.2 we can see how the world switch is executed.

### 4.1.2 Normal World Components

As explained before, our microkernel has a TrustZone Virtual Machine Monitor responsible for handling a normal world rich operating system. The normal world runs a paravirtualized Linux kernel modified to support data exchange between worlds, as well as allowing for world switching. In this operating system, we will have normal applications running, which will replicate an average user environment, in order to provide normal working conditions.
This Linux kernel, version 2.6.35.3, is a modified version of the original kernel, which was adapted to be compatible with the i.MX53 QSB board. We modified the Linux kernel used in the normal world in order to be able to work with the Genode. The main changes focused on adding system calls to trigger the world switch. In addition to this, some kernel modules have been developed to support the creation of the shared memory buffer. Other thing worth mentioning is the lack of several libraries and compilers on the kernel, in order to maintain its size as small as possible. Thus all the applications that run in the kernel, are compiled for ARM using an external laptop, and any necessary libraries were manually added.

The applications running in the normal world can perform a system call to trigger the world switch. Alongside this, the applications can also write into a specific memory address, and pass it to the secure world along with its size. As far as we are aware it is possible to add any Linux library to work on this kernel, but the bigger it is, the longer it will take to boot.

4.2 Implementation Challenges

The implementation of the ITZ library and its integration with a hardware board with the TrustZone extension has shown several implementation challenges inherent to the development of the features and components described in our architecture:

- Configuration of the monitor mode in the secure world, to assure that the library can be used.

- Guaranteeing secure memory regions and memory management.

- The scope of the work itself, which has no middleware and requires low-level programming knowledge, as well a specific knowledge of how the Genode works.

There was also an interest in keeping a small TCB, i.e., low code in the secure world. By achieving a small TCB, we simplify the code, which in turn decreases the chance of design flaws, code flaws and unnecessary services that can be used by attackers to compromise the system, and therefore compromise the secure introspection we aim to achieve. This required fine-tuning Genode and other software to reduce its footprint. While maintaining a small TCB we also want our kernel to be flexible enough to feature memory management, threading and interrupt handling.
4.3 Secure World Components

The secure world is our trusted execution environment, which runs the modified microkernel and the TZ VMM. It provides isolation from the normal world, which makes it the ideal place to perform the introspection. This section will focus on the components that run in the secure world, mainly our introspection library. We will start by introducing the library, focusing on the classes of functions provided, and illustrating it with some code samples. These code samples are functions from ITZ, and are used in our applications, to perform the introspection of the normal world memory.

4.3.1 ITZ Library

As stated in Section 3, the ITZ library contains several classes of functions that were inspired on LibVMI. Some of these functions allow us to perform introspection of the memory, by enabling the TZ VMM to read and write into specific addresses in memory, and can be combined with the support functions, such as those for translation and printing of contents. In the paragraphs below, we will get into more detail, by providing code samples of some of the available functions and explaining their functionality. We will focus on the functions that are used for memory manipulation, while also mentioning some of the other available functions. These functions have been designed to work within the secure world, and to have direct access to the memory of the normal world.

Read and Write Functions

When using memory addresses in a normal world application, or in the kernel, we are dealing with virtual memory addresses that are translated in hardware via the MMU to physical memory addresses. If we target a memory region that does not relate to a valid translation entry in the translation table of the process, we will get a translation fault. However, we are not able to access the virtual addresses directly, and as such the translation to a physical address is always required when requesting for a read function. The same problem happens whenever we want to read the content of a kernel symbol, so the solution is to request a translation from the symbol to a valid address.

On other hand, the reads from physical memory can be done directly, without the need of a translation, provided that the given address is valid, which is verified by our read functions. Similarly to the read functions, the corresponding write functions also require the translation beforehand, through the same process described above.

In Figure 4.3 we can see a sample of the code used to read from a physical address in the secure world, and verify that by pushing the obtained physical address to the top of the stack.
we are able to read its content. However, we are not able to access the virtual addresses directly, and as such the translation to a physical address is always required when requesting for a read function. The same problem happens whenever we want to read the content of a kernel symbol, so the solution is to request a translation from the symbol to a valid address.

```c
size_t vmi_read_pa (Vm *vm, uint64_t paddr, void *buf, size_t count){
    void *buf;
    Genode::size_t _buf_size;
    Ram * ram;
    //The buf_base is initialized with the physical address to read
    Genode::addr_t buf_base = paddr;
    //The _buf_size is initialized with the number of bites we want to read
    _buf_size = count;
    //To read the _buf_size bites we get the address corresponding to the end of the size
    Genode::addr_t buf_top = buf_base + _buf_size;
    ram = _vm->ram();
    Genode::addr_t ram_top = ram->base() + ram->size();
    //Check if any bounds are broken
    bool buf_err;
    buf_err = buf_top <= buf_base;
    buf_err |= buf_base < ram->base();
    buf_err |= buf_top >= ram_top;
    if (buf_err) {
        PERR("Illegal block buffer constraints");
    }
    //Read the content in the ram at the given address into the _buf and prints it
    Genode::addr_t buf_off = buf_base - ram->base();
    _buf = (void *)(ram->local() + buf_off);
    ...
}
```

Figure 4.3: Code to Read from a Physical Address.

**Translate and Print Functions** In Figure 4.4 we can see the code for translating a normal world virtual address into a secure world physical address. As mentioned before, when in the normal world, applications make use of virtual memory addresses. So in order to obtain the corresponding physical address of virtual address we access the MMU, which will go through the map until it finds the pair (virtual address, physical address) and gets the one we requested.

A similar process happens when we want to obtain the secure world’s virtual address of a physical address. The main difference is that it requires a call to an internal service, which will generate the virtual address using the code in Figure 4.5. There is also another function that translates a kernel symbol to its corresponding address, by searching a list of (virtual address, kernel symbol) pairs and returning the address. The print functions are very straightforward functions, which take any virtual or physical address and print out the corresponding hexadec-
/**
 * Performs the translation from a user virtual address to a physical address.
 * @param[in] *vm Genode instance
 * @param[in] vaddr Desired virtual address to translate
 * @return phys_addr_SW Physical address
 */
uint64_t vmi_translate_uv2p(Vm *vm, uint64_t vaddr){
    Genode::addr_t phys_addr_SW = _vm->va_to_pa(vaddr);
    Genode::printf("Corresponding Physical Address: %lu\n", phys_addr_SW);
    return phys_addr_SW;
};

Figure 4.4: Code to Translate a Virtual Address into a Physical Address.

class RAM{
...
    Genode::addr_t va(Genode::addr_t phys) {
        if ((phys < _base) || (phys > (_base + _size)))
            throw Invalid_addr();
        return _local + (phys - _base);
    }
...
}

Figure 4.5: Auxiliary Code to get a Virtual Address from Physical Address.

4.3.2 Other Functions

The functions mentioned above are those that we considered more important when developing our library. However, there are other classes of functions offered by LibVMI and by our own library. These functions exist to support and provide extra functionality to the introspection library, and as such we decided to only implement some of them, while creating stubs for the remaining ones in order to provide a more complete library.

The get and set functions have been implemented partially, to provide some useful data that the LibVMI offers, e.g. the offsets of the kernel running in the normal world. For the cache functions, we modified some of the functions to support our work, such as maintaining a log of the used functions, accessed memory addresses, and the hashes obtaining from specific memory regions.

To complement our library, and to be used as part of our use case application, we decided to support four hashing functions.

Instead of developing our own versions, we decided that it was better to use well-known
} else if(Genode::strcmp(str3, hashname)==0) {

    // Initalizes the context with SHA256
    SHA256_CTX c;
    unsigned char out[SHA256_DIGEST_LENGTH];
    SHA256_Init(&c);

    // Add the message data
    for (i=0; i<51; i+=1) {
        vmi_read_addr_va(_vm, pt[i], 0, buf, 8);
        SHA256_Update(&c, buf, 8);
    }

    // Finalize the context to create the signature
    SHA256_Final(out, &c);

    // Save and print the digest as one long hex value
    printf("kernel section hash value: ");
    for (i=0; i<SHA256_DIGEST_LENGTH; i++)
        printf("%x ", out[i]);
    printf("\n");

...
4.4 Discussion on Different Implementation Attempts

We started with the goal of porting LibVMI to an emulated version of the TrustZone using Quick Emulator (QEMU), which is an open-source hosted hypervisor used for hardware virtualization, available for free [B+07]. The lack of expertise in using QEMU and associated complexity, lead to a failed attempt to port LibVMI using the emulator. However the idea to port LibVMI remained, and we moved from a software supported platform to real hardware, and started development using the i.MX53 QSB. After studying the possibilities for ARM TrustZone regarding this board, we decided on the Genode, as it was easy to access and provided a fair degree of documentation. After exploring the boot process for the ARM platform and studying the TrustZone features, we were able to add a new system call to our normal world OS, which would be our trigger to the world switch. This enabled us to have access to a secure microkernel.

With this as basis we attempted once again to port LibVMI. The first challenge was the lack of libraries available, which lead us to an investigation on how to add them. The results of this are available in Appendix B. However even after finding a reliable way to port third party libraries, LibVMI proved impossible due to the extent of requirements as well as its interconnection with Xen and KVM. Thus we concluded that the difficulty to achieve such task was out of the scope of this dissertation, and it would also go against our goal to maintain a small TCB. As such we decided to develop our own library, using LibVMI as basis for the design of our API, and Genode’s base-hw microkernel as support platform to develop our functions.

Summary

We implemented a prototype of our ITZ library for a TrustZone-aware development board with the concern of keeping a small Trusted Computing Base (TCB) and maintaining an API as similar to LibVMI as possible, in a way to allow an easy port of existing application. We described our system configuration, focusing on our microkernel running in the secure world. It is responsible for memory management, thread execution, and context switch operations between worlds. In addition, we have the TZ VMM which can manage a paravirtualized Linux kernel running in the normal world. In the normal world we have a new system call called SMC which is responsible for preparing the system for a world switch and allocating a shared memory region to communicate between worlds. The ITZ library runs on top of the secure world kernel and supports operations such as reading, writing, translating and printing of virtual and physical addresses, as well as symbols. Additionally it also provides some of the support functions offered by the LibVMI, and implements several core hashing functions, such as MD5, SHA1, SHA256.
and SHA512, which are later used as part of our use case application. In the next chapter, we present several applications that make use of the ITZ library.
Chapter 5

ITZ Tools

This chapter describes a set tools designed using our TrustZone compatible library for introspection. We were successful in porting some of the existing applications for LibVMI with minimum effort, due to the nature of our own library, which provides most of the needed functions. Below, we describe the implementation of our own use case application, which makes use of several functions of our library (Section 5.1), as well as our implementation of the demos based on the ones from LibVMI (Section 5.2).

5.1 Kernel Integrity Check

As use case, we wanted to perform a kernel check, using the majority of the functions available in our library. This tool was inspired on an existing tool that used LibVMI [Zha], that we adapted to use ITZ instead. This was not exactly a port, as some parts of the code were restructured. However, these changes required little effort, mainly due to our goal of maintaining a similar API to LibVMI. The idea behind the existing tool is simply to generate an MD5 hash for a given kernel boundary. Since there are parts of the kernel that remain unchanged during runtime, the generated hash for those boundaries is expected to remain the same, and any modification could be a sign of malware [ZJ12].

The tool needs to know some kernel addresses. Therefore, we first need to obtain the `System.map` file from the normal world Linux kernel. This file is a symbol table used by the kernel. With this we have a way to associate kernel symbol names to their corresponding virtual addresses in memory. To assure that the file is not compromised, we obtain it right after compiling the kernel, and feed it directly into our tool in the secure world.

This tool starts by using the translation functions to obtain the addresses corresponding to the kernel boundaries. Then it goes through the kernel addresses between those boundaries.
These addresses are virtual addresses from the kernel, so we use our library to read them, by first doing the translation from virtual to physical addresses. Afterwards we read their content into an hash context, that is updated every time a new address is read. The hashes can be chosen from our available implementations.

Finally, when all the addresses are processed, the output is the corresponding hash. To verify if there has been any change in the kernel addresses, we simply need to compare an initial hash with the new one. If they are not the same then the kernel has been compromised. If all is verified, a certificate is generated which can be sent to a third party in order to assure that the results are not tampered by returning them to the normal world.

5.2 Tools Based on the LibVMI Examples

To establish some comparison to LibVMI, we decided to recreate some of the examples provided in their GitHub repository [Lib]. From these examples, we were able to confirm the simplicity of porting code from LibVMI to ITZ, while at the same time assuring that the results were those expected when using our functions.

**Dump Memory**  This example does a simple memory dump. It obtains the size of the physical addresses, then goes through them, writing the obtained content into a file. It starts by opening the file for writing, and then we use a library function to obtain the maximum physical address. Afterwards the tool reads the content of each address using the library function to read physical addresses, and writes them to a file, until it reaches the last address.

The file can be either stored in the secure world memory or in a memory card (e.g., if the space in the secure world memory is not enough). We tested this functionality with a microSD card inserted in the SD/MMC card slot of the device.

**Map Address**  The map address example aims to get an hexadecimal print of the requested address. In order to achieve that, it receives the address to map as input, which will then be processed to print the corresponding content by calling the function to print from virtual addresses. Before this process there is a check to verify the validity of the address, and if it returns true the function to print the content is executed.

As we can see by observing the Figures 5.1 and 5.2, there is little difference in using LibVMI or the ITZ. The main differences are the lack of need to initialize the library using ITZ, as well as a difference in the inputs for the used functions, due to our own implementation. Other than that, they are identical and produce the same output.
/* this is the address to map */
char *addr_str = argv[2];
addr_t addr = (addr_t) strtoul(addr_str, NULL, 16);

/* initialize the libvmi library */
if (VMI_FAILURE == vmi_init_complete(&vmi, name, VMI_INIT_DOMAINNAME, NULL,
                                     VMI_CONFIG_GLOBAL_FILE_ENTRY, NULL, NULL)) {
    printf("Failed to init LibVMI library.\n");
    goto error_exit;
}

/* get the symbol's memory page */
if (VMI_FAILURE == vmi_read_va(vmi, addr, 0, PAGE_SIZE, memory, NULL)) {
    printf("failed to map memory.\n");
    goto error_exit;
}
vmi_print_hex(memory, PAGE_SIZE);

Figure 5.1: Map Address Example for LibVMI.

/* this is the address to map */
char *addr_str = data;
addr_t addr = (addr_t) strtoul(addr_str, NULL, 16);

/* get the symbol's memory page */
if (VMI_FAILURE == vmi_read_va(_vm, addr, 0, PAGE_SIZE, memory, NULL)) {
    printf("failed to map memory.\n");
    goto error_exit;
}
vmi_print_hex(addr, PAGE_SIZE);

Figure 5.2: Map Address Example for ITZ.

**Map Symbol** The map symbol example is similar to the one we just described, but instead of receiving an address it receives a symbol. Each symbol corresponds to an address in our symbol table, which we obtain through the `System.map` file. To map the content of a given symbol, we first verify the validity of the symbol, by getting the corresponding virtual address. Once that is done the process is similar to the example above, obtaining the contents for the given symbol and printing them afterwards. By observing the Figures 5.3 and 5.4, we see that they present similar differences to those exposed on the example above.
Summary

In this chapter we present several applications that make use of our introspection library. We start the chapter by describing the design and implementation of our use case application, explaining how we make use of our library to create an application capable of validating the contents of portions of the kernel.

Afterwards, we show a few more applications, that were developed with basis on the existing LibVMI examples. This applications show us the easiness of porting existing LibVMI applications to use our library. Finally we mention some partial implementations, which were not successful due to limitation of our work environment. In the next chapter we present the experimental evaluation made using our introspection library.
Chapter 6

Evaluation

In order to evaluate our system we have decided to measure and study different aspects of this implementation. In this chapter we start by describing the methodology used throughout our experiments (Section 6.1), then we evaluate the overhead of the context-switch mechanism (Section 6.2). Following that we analyze the performance of the ITZ library functions and compare some of them with normal world counterparts (Section 6.3). Afterwards we analyze the performance the hashing functions (Section 6.4), and the differences on the trusted computing base size (Section 6.5). Finally, we discuss the macro benchmarks of our examples (Section 6.6) and our use case application (Section 6.7), concluding with a discussion over some of the security considerations and limitations of our work (Section 6.8).

6.1 Methodology

Our evaluation testbed consisted of an i.MX53 Quick Start Board, featuring a 1 GHz ARM Cortex-A8 Processor, and 1 GB of RAM memory. The board executed our system from a 4GB mini SD card, split into two partitions, the first was flashed with our modified versions of Genode and the Linux kernel, and the second one works as a file system. For each experiment, we report a mean of 30 runs, with the exception of the context-switch overhead and micro benchmarks in which we report a mean of 1000 runs. To measure the execution time of a specific operation executed in the development board, either on the normal or secure worlds, we implemented two functions which leverage the gettimeofday system call to start and end a timer.

6.2 Context-Switch Overhead

Before analyzing the performance overhead of using the library in the secure world, we first evaluate the overhead of switching from the normal world Linux kernel to the secure world run
time. This allows us to better understand how much of the overhead measured in the section above is caused by the context-switch. Since the value is expected to be very small, we have decided to perform 1000 runs, and then calculate the mean and standard deviation.

To evaluate the context-switch overhead, we measured the execution time of a simple call for the world switch, starting the timer right before the normal world executes the call, returning immediately and stopping the timer once the normal world OS is back in control. We measured an average of 0.08 milliseconds, with a standard deviation of 0.03 milliseconds. By observing these results we can say that the time spent with a world switch is negligible in most situations.

6.3 Micro Benchmarks of the Library Functions

In order to evaluate our library functions in the secure world, we decide to measure the execution time of the functions that deal with memory manipulation. To establish a baseline for our different functions, we measure the execution time of corresponding versions in the normal world, and compare these times with the execution time of processing the functions using the secure world. By comparing these execution times we can conclude if there is a significant overhead caused by context-switching between normal and secure world, which is intrinsic in ITZ.

With that in mind, we focused on the functions responsible for reading, writing, translating and printing, and performed 1000 runs for each function. The execution flow for the testing the library is as follows: the test starts with the normal world running a tool which fills the shared memory buffer. Then we obtain the physical address and the size of the buffer, and put them on registers \( r2 \) and \( r3 \) respectively. Right before the system call to perform the world switch, we start the timer. On the secure world, we execute the library functions providing the obtained address and size from the registers. Upon returning the value, the timer stops, and we verify its content. From the values obtained we measure the average execution time and standard deviation. On the paragraphs below we have the tables for each class of functions and an analysis of the values.

<table>
<thead>
<tr>
<th>Kernel Symbol</th>
<th>Normal World</th>
<th>Secure World</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translate Physical Address to Virtual</td>
<td>0.085 ± 0.002</td>
<td>0.09 ± 0.002</td>
</tr>
<tr>
<td>Translate Virtual Address to Physical</td>
<td>0.09 ± 0.002</td>
<td>0.085 ± 0.002</td>
</tr>
</tbody>
</table>

Table 6.1: Microbenchmarks for translation functions (in milliseconds)

Table 6.1 shows the times for the translation function available in our library. To measure
the times and maintain consistency, we used the physical address for our shared buffer, and translated it to the corresponding secure world virtual address. From that we obtained an average of 33.22 milliseconds with a standard deviation of 0.22. We then used the obtained address to perform the reverse operation and obtained an average of 37.90 milliseconds with a standard deviation of 0.24.

Regarding the translation of a kernel symbol, we chose the symbol `lookup_processor_type` with the corresponding virtual address of `c00081bc`, according to the `System.map` file. The average of this translation is 36.79 milliseconds with a standard deviation of 0.22. The value of the kernel symbol translation is not higher, due to the fact that we save the pair (symbol, address) of the `System.map` file inside the secure world. That way by handling the file before using the library, allows us to save execution time, since the symbol and corresponding values are cached inside the secure world.

<table>
<thead>
<tr>
<th>Execution Time</th>
<th>Write 8 bits</th>
<th>Write 16 bits</th>
<th>Write 32 bits</th>
<th>Write 64 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Secure World)</td>
<td>34.02 ± 0.25</td>
<td>34.45 ± 0.23</td>
<td>34.70 ± 0.24</td>
<td>35.04 ± 0.23</td>
</tr>
<tr>
<td>(Normal World)</td>
<td>0.049 ± 0.001</td>
<td>0.050 ± 0.001</td>
<td>0.051 ± 0.001</td>
<td>0.051 ± 0.001</td>
</tr>
</tbody>
</table>

Table 6.2: Microbenchmarks for writing to memory (in milliseconds)

In Table 6.2, we have the times for writing different sizes of data into memory. The chosen sizes are according to our library functions, that provide writes for 8, 16, 32 and 64 bits, as well as a custom size. To measure these functions, we decided to write characters from a string onto the physical address of the shared buffer. That way we measured only the time of copying the data to the memory, without the need to perform any additional tasks, such as address translation, which is needed when writing to a virtual address. The buffer had 128 bytes allocated to receive data, which was more than enough for our tests, which required a maximum of 8 bytes.

As we can see by the values obtained, the cost of writing more bytes does not increase by much the overhead of writing from the secure world. Since writing 1 byte took an average of 34.02 milliseconds with a standard deviation of 0.25 and writing 8 bytes took an average of 35.04 milliseconds with a standard deviation of 0.23, we can say that most of the cost of the execution is in locating the given memory position, rather than copying the data.

In Table 6.3, we can observe the times measured when performing reads from memory. These read functions allow the user to read 8, 16, 32 or 64 bits from a given address. There is also an additional function for a custom amount of bits. Similarly to what was done with the writing functions, we prepared the shared buffer by filling it from the normal world with a string of 128 bytes. Since we have the physical address for the shared buffer and want to avoid additional
translations, we perform the reads on that address.

As we can observe, there is a bigger difference in the average time of each function, than there was for the writing functions. This can be explained by the fact that the read function print the obtained value to the terminal, which does not happen on the writes. Due to that we see an approximately 47% increase from reading 8 bytes, when compared to the value of reading 1 byte.

<table>
<thead>
<tr>
<th></th>
<th>Read 8 bits</th>
<th>Read 16 bits</th>
<th>Read 32 bits</th>
<th>Read 64 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Execution Time</strong></td>
<td><strong>(Secure World)</strong></td>
<td>37.74 ± 0.25</td>
<td>40.25 ± 0.25</td>
<td>45.24 ± 0.29</td>
</tr>
<tr>
<td><strong>Execution Time</strong></td>
<td><strong>(Normal World)</strong></td>
<td>0.048 ± 0.001</td>
<td>0.052 ± 0.001</td>
<td>0.058 ± 0.002</td>
</tr>
</tbody>
</table>

Table 6.3: Microbenchmarks for reading from memory (in milliseconds)

In Table 6.4 we have the average time for printing the hexadecimal and ascii values of a chunk of bytes, obtained from virtual and physical addresses, as well as kernel symbols. These values were obtained once again by using the shared buffer to store 128 bytes of data, which was then printed on the terminal upon calling each function. As expected the values for printing from a virtual address were higher than from a physical address, due to the fact that there is the need for an additional translation of the address.

Regarding the values for printing from a kernel symbol, these were also according to what was expected, due to the translation of the kernel symbol into the corresponding virtual address, and then that address into the physical address.

In general when comparing the results of both worlds, we can observe lower times in the normal world. This can be explained by the lack of a context switch, and due to the fact that the similar functions were tested at a kernel level in the normal world. The print functions are the most costly due to the writing to the terminal, which causes a big overhead. Overall we can conclude that despite these differences there is not a significant cost of using the library in the secure world for most of the functions, and the security advantages provided by the TrustZone overcome the overhead.

In Table 6.4 we have the average time for printing the hexadecimal and ascii values of a chunk of bytes, obtained from virtual and physical addresses, as well as kernel symbols. These values were obtained once again by using the shared buffer to store 128 bytes of data, which was then printed on the terminal upon calling each function. As expected the values for printing from a virtual address were higher than from a physical address, due to the fact that there is the need for an additional translation of the address.

Regarding the values for printing from a kernel symbol, these were also according to what was expected, due to the translation of the kernel symbol into the corresponding virtual address, and then that address into the physical address.

In general when comparing the results of both worlds, we can observe lower times in the normal world. This can be explained by the lack of a context switch, and due to the fact that the similar functions were tested at a kernel level in the normal world. The print functions are the most costly due to the writing to the terminal, which causes a big overhead. Overall we can conclude that despite these differences there is not a significant cost of using the library in the secure world for most of the functions, and the security advantages provided by the TrustZone overcome the overhead.

In Table 6.4 we have the average time for printing the hexadecimal and ascii values of a chunk of bytes, obtained from virtual and physical addresses, as well as kernel symbols. These values were obtained once again by using the shared buffer to store 128 bytes of data, which was then printed on the terminal upon calling each function. As expected the values for printing from a virtual address were higher than from a physical address, due to the fact that there is the need for an additional translation of the address.

Regarding the values for printing from a kernel symbol, these were also according to what was expected, due to the translation of the kernel symbol into the corresponding virtual address, and then that address into the physical address.

In general when comparing the results of both worlds, we can observe lower times in the normal world. This can be explained by the lack of a context switch, and due to the fact that the similar functions were tested at a kernel level in the normal world. The print functions are the most costly due to the writing to the terminal, which causes a big overhead. Overall we can conclude that despite these differences there is not a significant cost of using the library in the secure world for most of the functions, and the security advantages provided by the TrustZone overcome the overhead.

58
6.4 Microbenchmarks of the Hash Functions

This section evaluates the performance of the hashing functions used in the secure world, which can be used to assess data integrity. In order to measure this we have used a test string with the content "Hello from the normal world", that is obtained from the normal world, and executed on it the different implemented hash functions in order to compare the execution times. To have a basis for comparison we have used the same string and functions in the normal world. With this we want to check if is there any significant overhead caused by using the secure world.

The execution flow for the secure world case is as follows: we start to measure the execution time as soon as the string is allocated in the normal world. Our normal world application then triggers the SMC system call to proceed with the world-switch request. On the secure world the string is read and then processed with an hash function, returning to the normal world when it finishes. Upon returning to the normal world the measuring of the execution time stops. The execution flow in the normal world is more straightforward, using an application to execute the hash function on the string without switching worlds. The measuring is done the same way.

![Comparison of Hashing Functions - Normal World vs Secure World.](image)

Figure 6.1 shows the execution times of all the hashing functions for the given string. By analyzing this graph, which depicts the execution time in milliseconds in the Y-axis and the corresponding hash in the X-axis, we can observe a constant penalty for the secure world execution flow when compared to the same functions executed in the normal world. This overhead is associated to the implementation of the microkernel’s memory manager, scheduler and compiler.
optimization.

6.5 Trusted Computing Base Size

One of the requirements for our work discussed in this dissertation was developing a secure introspection library while maintaining a small trusted computing base. In order to evaluate the size of the TCB, we counted the number of lines of code present in our systems’ components and compared it to other similar kernels and LibVMI. To count the lines of code (loc) we have used cloc [AlD], an application that does that task for many programming languages, including all types of lines: blank lines, comment lines, and physical lines of source code.

<table>
<thead>
<tr>
<th>World</th>
<th>Component</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal world</td>
<td>Linux kernel</td>
<td>9132 K</td>
</tr>
<tr>
<td>Secure world</td>
<td>Microkernel</td>
<td>20 K</td>
</tr>
<tr>
<td>Secure world</td>
<td>Genode OS framework</td>
<td>10 K</td>
</tr>
<tr>
<td>Secure world</td>
<td>TZ VMM</td>
<td>3 K</td>
</tr>
<tr>
<td>Secure world</td>
<td>ITZ Library</td>
<td>2 K</td>
</tr>
</tbody>
</table>

Table 6.5: Code base size of our system.

Table 6.5 shows the lines of code of the several components of our system. The first component is the Linux kernel, which was used as the normal world kernel. This is provided as baseline, as the Linux kernel is not part of the TCB. This kernel comprises a large and complex code base of more than 9 million loc, mainly due to the necessary additional libraries such as libc in order to run applications. Nevertheless, this is still much smaller than the current Linux kernel (4.15.9) that has 20323 Kloc.

The second and third components, are the micro kernel and the Genode OS Framework that we have running in the secure world. In order to avoid high complexity kernels, Genode provides a micro kernel with 20 Kloc, alongside with some modules that can be compiled with it. From those, we use the Genode OS Framework, that has a small code base of 10 Kloc and provides some tools to handle the world switch.

The fourth component is the TZ VMM, which is based on the version distributed by Genode, which includes the VMM implementation, the secure memory manager and the necessary code to boot the normal world Linux kernel. With the additional code to support the shared memory buffer as well as the use of our library and a file system, we managed to have a code base size of 2.5 Kloc.

Finally the fifth component is the ITZ library that has 2 Kloc, and contains several classes of functions, such as reading, writing, translating and printing from memory.

Adding the values in the table, we have a TCB of 35 Kloc that is 2 orders of magnitude lower
than an OS like Linux. ITZ is much smaller (2 Kloc) than LibVMI that has 20 Kloc, although LibVMI supports more OSs and a few different hypervisors.

### 6.6 Macro Benchmark I - LibVMI Examples

To test our library using more practical cases, we decided to look into the examples provided by the LibVMI. One of our main objectives of measuring these cases, is to study the impact that the use of the secure world has in the normal world, since every time an application runs in the secure world, the normal world is in a paused state. From the several examples available we decided to adapt the memory dump, and the mapping of addresses and symbols, to support our library functions. With this we aim to have both a way of comparing our library to LibVMI, as well as a way to measure the portability of existing applications. Each of these examples was executed 30 times, measuring the mean and standard deviation of the executions in milliseconds.

<table>
<thead>
<tr>
<th>Execution Time</th>
<th>Dump Memory</th>
<th>Map Address</th>
<th>Map Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>186.34</td>
<td>2.10 ± 0.003</td>
<td>2.11 ± 0.003</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.6: Macrobenchmarks (in seconds) for the examples from LibVMI

As we can see in Table 6.6, the most costly example is as expected the memory dump, with 186 seconds of execution time. This was already expected due to the fact that we are dumping the whole normal world memory onto a file and saving it to the file system located in the micro SD card.

The memory dump is done using our library functions to obtain the initial address and size of the RAM of the normal world. Then we proceed by dumping all the RAM data into a file, which ends up with a size of 268.4 MB, which can be used to perform further memory analysis.

The map address and map symbol examples, offer similar execution times, with an average in seconds of 2.10 and 2.11 accordingly. This can be explained due to their similarity of both implementation and execution. The way they were implemented was according to the LibVMI example, where the former receives an address to map, while the other receives a symbol. The execution process is also similar, with the first one using the function to print the hex from virtual addresses, provided by our library, which requires the translation of the virtual address to physical to obtain the corresponding memory page. On other hand the map symbol prints the hex from kernel symbol function, which requires two additional function, the first being the translation of kernel symbol to virtual address, and then from virtual address to physical. Only then it is possible to print the corresponding hexadecimal.
6.7 Macro Benchmark II - Kernel Integrity Check

In order to perform an evaluation of our work in a more complex scenario, we used an application to perform a kernel check, using our introspection library, as mentioned in greater detail in the section above. For the purpose of our evaluation, we have selected 15, 30 and 50 kernel addresses from a read only data section. We then ran our application with them for a total of 30 runs. Figure 6.2 shows the execution times for the different hashes on those addresses.

![Figure 6.2: Macro Benchmarks for Kernel Integrity Check](meta-chart.com)

As we can observe the difference in the execution time for the different hashes is not very significant. If we increase our application to include more addresses, the time increases accordingly resulting in a larger overhead if we wanted to include all the 40K addresses that are inside the kernel boundaries. By comparing the obtained hash with a previously obtained one, it is possible to verify if there have been any modification to the read only data section, which would mean that the system had been compromised. Overall we can conclude that this application is viable to use if we want to verify the integrity of segments of the kernel memory, such as parts of the read only data section. Also these results show that the most viable and secure hash to use would be SHA256, which shows an acceptable increase of the execution time when compared to SHA1, which has already been compromised.
6.8 Security Considerations and Limitations

The attack surface of our work is mainly composed by the SMC instruction used in the normal world OS to perform the world switch and the shared memory buffer also allocated by the normal world. The applications to perform the world switch in the normal world are comprised of few lines of code, and as such the probability of vulnerabilities is relatively narrow. The main concern on these applications is the possibility of abusing the SMC instruction to perform a loop of world switches, resulting in a denial of service by locking the normal world OS.

The other security concern is related to the shared memory region, which allows to read and write data between worlds. However due to the fact that it is limited to that, the only possible attack is writing wrong information into the secure world through the shared memory region. This is not exactly an issue because we assume the possibility of the normal world OS faking information, by accessing the data directly in the secure world. There are also hashing functions in the secure world that can be used to verify the veracity of the data provided by the normal world, by hashing both the received and the introspected and comparing the hashes. Due to the support of secure memory provided by the ARM TrustZone, we can safely say that the memory allocated for the secure world is protected and therefore cannot be accessed by the normal world in any way.

The normal world is not protected, so it can be compromised. For instance an user that creates an application to request the secure world for introspection information of a given process, can have its application compromised. Since the normal world OS is vulnerable an attacker can spoof the world switch and return fake information the user. As such we can only consider secure information obtained in the secure world and by the library in the secure world. The solution to this would be create such application inside the secure world, and schedule the secure world to do the world switch automatically. Since the time period could be random, the normal world could never guess at which time the switch would take place and therefore could not spoof it. Regarding limitations, it is worth noting that in its current state, we do not provide support for Android or dynamic tracing. Finally any kind of system on chip attacks to the ARM architecture are out of the scope of this paper and are not considered.

We assume that the hardware is correctly implemented and the TrustZone mechanisms cannot be circumvented.
Summary

In this chapter we evaluate the performance of our implementation and we give a theoretical analysis and discussion on the initial implementation attempts as well as some general security considerations and limitations of the system. We start the chapter by describing the evaluation methodology used throughout. We then proceed to measuring the overhead of the context-switching mechanism.

Afterwards we do an analysis on the performance of the different introspection functions in our library, as well as the different hash functions. We proceed with the evaluation of our achieved trusted computing base size, finishing with an analysis on the use and development of an application to detect for modifications on section of the kernel.

Finally we do a discussion on different implementation attempts before discussing some security considerations and limitations of the system. The next chapter finishes this dissertation by presenting the conclusions and guidance for future work.
Chapter 7

Conclusions

Malicious code is the root of many existing security problems [SBY+08]. The capability of inspecting code running on a system grants the possibility to detect and prevent any malicious code from doing harm. By using introspection combined with the ARM TrustZone it is possible to achieve an inspection of the code running on the system from the outside, thus allowing for the system to be able to detect attacks.

In this dissertation we introduce ITZ, a library capable of introspection that leverages the use of the ARM TrustZone. Our library provides clear isolation from a possibly compromised normal world OS and guarantees that sensitive data is only handled in the trusted execution environment. By doing so the inspection of memory on the normal world of a device allows for more reliable results. Through experimental evaluation of our library we observe that our work adds a small overhead caused by running in a secure environment, when compared to a system that runs in the normal world, with a normal sized kernel. The only exception are the hashing functions used, that showed a more significant overhead, when using larger amounts of data.

We were also able to maintain our goal of a small trusted computing base, by stripping the system of all the unnecessary libraries, keeping only the essential, and therefore achieving a more reduced attack surface. This also affects boot time, which we were able to maintain short, due to the small size of the image we obtain after compiling. Ultimately we achieved the goal of creating a functional library that can be used by new developers, in a way that eases up the extraction of run-time data for analysis.

7.1 Future Work

In the future, the goal is to improve even more the library in order to supported additional functions. This would leave us with a more complete library capable of being used to create
several other applications, such as analyzing and registering events. In addition to this we want to create a dynamic analysis tool, that leverages both the use of our library and the ARM TrustZone, resulting in a secure analysis of the running processes. Finally we would like to expand the support of our library, by not limiting it to the use of the Genode, which can only run in a limited amount of boards. In doing so we would be providing an introspection tool, that would leverage a secure environment and could be used across a wider range of devices that support the ARM architecture.
Bibliography


Appendix A

Genode Configuration

Genode provides a way to select the components we want to use in the form of generic building blocks. Through its hierarchic and recursive structure a parent has full control over the way its children interact with each other and with the parent. At boot time, the init component is started by the core, assigns all physical resources and controls the execution of all other components. This policy is defined by a configuration file in XML form, which declares the number of children, their relationships and the assignment of resources. This section contains the Genode configuration file which runs the ITZ library. As it can be seen, between lines 23 and 47, there is the configuration for the file system using the SD card, in which the partition 2 is set as a writable second extended file system (ext2fs).

1  <config verbose="yes">
2    <parent-provides>
3        <service name="ROM"/>
4        <service name="RAM"/>
5        <service name="IRQ"/>
6        <service name="IO_MEM"/>
7        <service name="IO_PORT"/>
8        <service name="CAP"/>
9        <service name="PD"/>
10       <service name="RM"/>
11       <service name="CPU"/>
12       <service name="LOG"/>
13       <service name="SIGNAL"/>
14       <service name="VM"/>
15    </parent-provides>
16    <default-route>
17        <any-service><any-child/><parent/></any-service>
<start name="timer">
  <resource name="RAM" quantum="3M"/>
  <provides><service name="Timer"/></provides>
</start>

<start name="sd_card_drv">
  <resource name="RAM" quantum="20M"/>
  <provides><service name="Block"/></provides>
  <config file="ext2.raw" block_size="512"/>
</start>

<start name="part_blk">
  <resource name="RAM" quantum="10M"/>
  <provides><service name="Block"/></provides>
  <!-- route part_blk to the ata_driver -->
  <route>
    <service name="Block"><child name="sd_card_drv" /></service>
    <any-service><parent/><any-child/></any-service>
  </route>
  <config>
    <policy label="tz_vmm" partition="2"/>
    <policy label="rump_fs" partition="2"/>
  </config>
</start>

<start name="rump_fs">
  <resource name="RAM" quantum="8M"/>
  <provides><service name="File_system"/></provides>
  <route>
    <any-service><child name="part_blk" /> <any-child/> <parent/></any-service>
  </route>
  <config fs="ext2fs"><policy label="" root="/" writeable="yes"/></config>
</start>

<start name="tz_vmm">
  <resource name="RAM" quantum="10M"/>
  <route>
    <any-service><child name="part_blk" /> <any-child/> <parent/></any-service>
  </route>
  <config>
    ...
  </config>
</start>
<libc stdout="/dev/log" stderr="/dev/log">
    <vfs>
        <dir name="dev">
            <log/>
        </dir>
        <jitterentropy name="random"/>
    </dir>
    <fs/>
</vfs>
</libc>
</config> }
Appendix B

Workspace Setup

Here we have some of the required steps to perform the setup of the work environment necessary to implement our project.

B.1 SMC call as Linux System Call for ARM Architecture

Step 1:
Add a system call number

path : /arch/arm/include/asm/unistd.h
e.g. :
#define __NR_my_smc (__NR_SYSCALL_BASE+377 )

Step 2:
Make an entry to calls.S with path given below

path: /arch/arm/kernel/calls.S
e.g. :
CALL(sys_my_smc) (the position has to be same as the system call number i.e., 377)

Note: This file will be automatically added to entry-common.S as the file entry-common.S includes it as follows.

#include "calls.S"

Step 3:
1. Create a directory in the kernel source directory.

# mkdir my_smc
2. Change into this directory

# cd my_smc

3. Create a my_smc.c file in this folder and add the definition of the system call to it as given below (you can use any text editor)

   # gedit my_smc.c

   Add the following code:

   #include<linux/kernel.h>
   #include<linux/syscalls.h>
   #include <asm/system.h>
   #include <linux/linkage.h>

   asmlinkage long sys_my_smc(void)
   {
       asm volatile("mov r0, #3  \
                     "dsb  \
                     "dmb  \
                     "smc  #0    \
                     "\n                   ::= "r0");

       return 0;
   }

4. Create a Makefile in the my_smc folder and add the given line to it.

   # gedit Makefile

   Add the following line to it:

   obj-y := my_smc.o

   This is to ensure that the my_smc.c file is compiled and included in the kernel source code.

5. Add the hello directory to the kernel’s Makefile and change back into the linux folder and open Makefile

   # gedit Makefile

   Goto line which says:

   core-y := usr/

   and add the hello folder.
core-y := usr/ my_smc/

Step 4: Register the function with the prototype.

path: /include/linux/syscall.h

Add the following line:

`asmlinkage long sys_my_smc();`

## B.2 Steps to port third party software inside TZ VMM

1. In

   ```
   $(BUILDDIR)/etc/build.conf uncommented this line
   REPOSITORIES += $(GENODE_DIR)/repos/libports
   ```

2. Execute the following

   ```
   cd $(GENODE_DIR)
   ./tool/ports/prepare_port libc
   ```

3. Inside

   ```
   $(BUILDDIR)/repos/os/src/server/tz_vmm/spec/imx53
   ```

   add the following to target.inc

   ```
   LIBS += libc
   ```

4. Add the header files you want such as stdio.h to main.cc inside

   ```
   $(BUILDDIR)/repos/os/src/server/tz_vmm/spec/imx53
   ```

5. Add the dynamic linker to the files (XXX.lib.so - It can be found in your build-directory under bin/) to be loaded inside the tz_vmm.run run script at line below.

   ```
   set boot_modules { core ld.lib.so timer init tz_vmm linux }
   ```

## B.3 Steps to manually compile Linux

1. Install ARM toolchain to cross compile Linux.


   ```
   # sudo mkdir /opt/CodeSourcery/Sourcery_G++_Lite
   ```
# sudo cp -R arm-2011.03/* /opt/CodeSourcery/Sourcery_G++_Lite
# rm -r arm-2011.03
# rm arm-2011.03-41-arm-none-linux-gnueabi.src.tar
# gedit ~/.bashrc

The last command in the script will open your .bashrc file. You can add the following text to the bottom of the file to make it easy to access CodeSourcery in the future.

export PATH=/opt/CodeSourcery/Sourcery_G++_Lite/bin:$PATH

2. Download the Linux kernel (linux-imx53_tz) modified by Genode Labs here
   https://github.com/skalk/linux.git (branch imx53-tz)

   # make CROSS_COMPILE=/opt/CodeSourcery/Sourcery_G++_Lite/bin/arm-none-linux-gnueabi-
   # ARCH=arm imx5_android_tz_defconfig

   # make CROSS_COMPILE=/opt/CodeSourcery/Sourcery_G++_Lite/bin/arm-none-linux-gnueabi-
   # ARCH=arm -j4 uImage

3. From Genode Build Directory run the following command to copy and use the compiled Linux uImage

   # dd if=<path_to_linux_dir>/arch/arm/boot/uImage of=bin/linux bs=64 skip=1

   In case of this error or similar

   /tmp/cc0tlN1r.s:61: Error: selected processor does not support ARM mode 'smc #0',
   scripts/Makefile.build:230: recipe for target 'drivers/video/mxc/mxc_ipuv3_fb.o'

   The solution is to add

   asm(".arch_extension sec

   before the call to the asm volatile method.

B.4 Steps to manually compile Genode

1. To build Genode version 16.08, first install the required packages and Genode tool-chain on your machine. Please note that some of the packages may be old. Packages:

   libsdl-dev, tcl, expect, autoconf2.64, autogen, bison, flex, g++, git, gperf, libxml2-utils, subversion, xsltproc, u-boot-tools, texinfo, libncurses5-dev

   To install Genode tool-chain (http://genode.org/download/tool-chain) run the following
commands:

```bash
# cd <genode-dir>
# tool/tool_chain arm

2. Build Genode:
Download Genode here https://github.com/genodelabs/genode/tree/16.08

# cd <genode-dir>
# ./tool/create_builddir hw_imx53_qsb_tz
# cd build/hw_imx53_qsb_tz/
# echo "SPECS += trustzone" >> etc/specs.conf
# echo "RUN_OPT += --include image/uboot" >> etc/build.conf
# make run/tz_vmm

Note: The Genode build system never touches the source tree but generates object files, libraries, and programs in a dedicated build directory (in this case the build directory will be hw_imx53_qsb_tz).

3. Create uImage manually if necessary

# /usr/local/genode-gcc/bin/genode-arm-objcopy -O binary \ var/run/tz_vmm/image.elf
var/run/tz_vmm/image.bin

# mkimage -A arm -O linux -T kernel -C none -a 0x70010000 \ -e 0x70010000 -d
var/run/tz_vmm/image.bin var/run/tz_vmm/uImage

4. Connect i.mx53 QSB to host machine via USB-to-serial cable (USB RS232 Cable):

# sudo picocom -b 115200 -r -l /dev/ttyUSB0

5. Use the uImage created above to boot the board using u-boot commands:

# ext2load mmc 0 addr uImage-file-path
# bootm addr

Example

# ext2load mmc 0:1 0x71000000 /uImage
# bootm 0x71000000