Operating System Support for Run-Time Security with a Trusted Execution Environment

- Usage Control and Trusted Storage for Linux-based Systems -

by

Javier González

Ph.D Thesis

IT University of Copenhagen

Advisor: Philippe Bonnet

Submitted: January 31, 2015

Contents

Abstract

Software services have become an integral part of our daily life. Cyberattacks have thus become a problem of increasing importance not only for the IT industry, but for society at large. A way to contain cyberattacks is to guarantee the integrity of IT systems at run-time. Put differently, it is safe to assume that any complex software is compromised. The problem is then to monitor and contain it when it executes in order to protect sensitive data and other sensitive assets. To really have an impact, any solution to this problem should be integrated in commodity operating systems.

In this thesis we introduce run-time security primitives that enable a number of trusted services in the context of Linux. These primitives mediate any action involving sensitive data or sensitive asset in order to guarantee their integrity and confidentiality. We introduce a general mechanism to protect sensitive assets at run-time that we denote split-enforcement, and provide an implementation for ARM-powered devices using ARM TrustZone security extensions. We design, build and evaluate a prototype Trusted Cell that provides trusted services. We also present the first generic TrustZone driver in the Linux operating system. We are in the process of making this driver part of the mainline Linux kernel.

Preface

The journey that has led to this writing has been long; at times exciting and at many others difficult. However, as in any other long term commitment, it is the process of walking the new path that is worth the effort. There are many people to thank for their time, knowledge, and above all support during these years. I can only hope that I have expressed my gratitude enough to them, before they can identify themselves here.

Thanks to Philippe Bonnet, my advisor, for taking care of me during these years: Thanks for not babysitting. Thanks for letting me hit the wall hard enough to know the feeling without getting hurt. Thanks for sharing your knowledge. But above all, thanks for helping me find something that I love doing.

Thanks to INTERACT for financing my Ph.D. Also, thanks to Inger Vibeke Dorph for handling it on the IT University side: Paperwork would have killed me long ago had it not been for you.

Thanks to my Ph.D and office colleagues, who have helped in all possible ways. Matias Bjørlig, Jonathan Fürst, Aslak Johansen, Joel Granados, Niv Dayan, Jesper Wendel Devantier, Mohammed Aljarrah, and Michal Moučka: Thanks for the coffees, the discussions, the comments, and the critics. But above all, thanks for taking the time for explaining what I could not understand alone.

Thanks to the SMIS group for hosting my stay abroad at INRIA Rocquencourt, France. Special thanks to Luc Bouganim for making it possible: This stay has indeed been one of the most productive periods of my Ph.D, and it is all thanks to you. Also, thanks to Philippe, Henriette, Anna, Clara, and Oscar for opening a home for me during my stay in France: I did feel home, and I will never forget your kindness.

Thanks to Xilinx for taking me on an internship. Special thanks to Dave Beal and Steve McNeil. Also, thanks to the rest of the security team at Albuquerque: Thanks for giving me the chance to learn and contribute on equal terms.

Thanks to all the people collaborating in the publications that have made this thesis possible. Michael Hölz and Peter Riedl from the Upper Austria University of Applied Sciences: Thanks for the great work. Also, thanks to their advisor, Rene Mayrhofer, for endorsing this collaboration.

Thanks to the IT University of Copenhagen for being such a fantastic place to work. To all

the people that make ITU what it is: Thanks. Special thanks to Freja Krab Koed Eriksen and Christina Rasmussen for supporting us, Ph.D students, in the best possible way: We all owe you, I hope you know it.

Thanks to family and friends for all the rest. Naming all of you would require twice as many pages as the ones already written: You know who you are. Thanks for the phone calls, the music, the bars, the concerts, the smiles, the beers, the dinners, the skype calls, the games, the movies, the trips, the messages, the coffees. Thanks for the discussions, the wine, the laughs, the all-nighters (the good ones), the good habits, and the bad ones... Thanks for always being there and making me feel alive.

Special thanks to my mum and dad, who have always truly supported me, even when that meant moving away from them: Gracias de corazón, por todo. Os quiero.

Finally, thanks to the love of my life, Mia, who, besides all the rest, has made short evenings worth a life: Tak for, at du altid minder mig om hvad, der er vigtigst. Jeg elsker dig.

Chapter 1

Introduction

1.1 Context

Cybercrime has a growing impact on all levels of society. According to a recent survey by the Ponemon Institute [156], the average cost of cybercrime for U.S. retail stores has more than doubled since 2013, reaching an annual average of $8.6M *per company* in 2014. In 2014 alone[1], *reported* cyberattacks have targeted well-known companies such as Amazon[2], Sony[3], or Apple[4]. The implications of these attacks are now social and political, not just economical. The 2014 hack to Apple's iCloud[5] targeting intimate photographies of celebrities quickly became *vox populi* and, while it did trigger a discussion about privacy and the security of cloud services, it very quickly became an issue involving the media and the public opinion. The Sony Pictures hack in late 2014 did not stop with the publication of a number of unreleased films and internal Sony sensitive emails, it increased the political tension between North Korea - who has been made the primary suspect for the attack by the FBI -, and the USA, especially after the follow-up Denial of Service (DoS) attack that left North Korea without Internet access for several days[6]. Political figures and media in both countries have even considered these cyberattacks *acts of war*. The full consequences of these series of cyberattacks have not yet been cleared.

With the adoption of personal devices, such as smart phones, tablets or set top boxes, cybercrime raises concerns ranging from creepiness [260], with applications leaking personal sensitive data, to the loss of privacy [116], or the fear for surveillance after Edward Snowden's revelations on massive surveillance programs in the USA [142, 138]. In order to address these concerns, it is necessary to introduce some control over the flow of sensitive data in personal

[1] http://www.itgovernance.co.uk/blog/list-of-the-hacks-and-breaches-in-2014/

[2] http://thehackernews.com/2014/12/password-hacking-data-breach.html

[3] http://www.theverge.com/2014/12/8/7352581/sony-pictures-hacked-storystream

[4] http://en.wikipedia.org/wiki/2014_celebrity_photo_hack

[5] http://www.theguardian.com/world/2014/sep/01/jennifer-lawrence-rihanna-star-victims-hacked-nude-pictures

[6] http://www.nytimes.com/2014/12/23/world/asia/attack-is-suspected-as-north-korean-internet-collapses.html?_r=0

devices, but also in the cloud [215]. However, cybercriminals and surveillance states exploit any vulnerability that allows them to bypass such control. Prove of it are the discovery of *backdoors* introduced in popular mobile operating systems such as iOS [299], or continuous stream of new revelations about existing surveillance programs[7].

In the preface of Gary McGraw's and John Viega's *Building Secure Software* [280], Bruce Schneier writes: *"We wouldn't have to spend so much time, money, and effort on network security if we didn't have such bad software security"*. As McGraw himself writes: *"Almost all modern systems share a common Achille's heel in the form of software"* [146]. Leading figures in the IT industry such as Microsoft's Bill Gates[8], Oracle's Larry Ellison[9], or Google's Larry Page[10] have publicly recognized security as a transversal problem in software systems. It can be argued that security experts and CEOs could use their position to exaggerate the security problem to their own benefit. But, to which extend do these claims resemble the truth? Are we surrounded by a generation of individuals that suffer from software security paranoia? Or is it in reality an issue that the IT industry is struggling with?

So far the enthusiasm for new opportunities has thwarted security concerns in the software industry. As a result, software is now permeating our daily lives. Quoting Marc Andreessen, *"software is eating the world"* [14]. However, changing the world trough software comes at the price of increased complexity. There are two negative consequences associated with complex software: (i) it introduces vulnerabilities, and (ii) it is more likely that undocumented behavior goes undetected. In terms of vulnerabilities, Gary McGraw has studied in depth what he calls *The Trinity of Trouble* [146, 202]: complexity, connectivity, and extensibility. His argument is that complex software that interacts with distributed systems in different ways (i.e., innovative services) necessarily contains vulnerabilities (*bugs*). Using lines of code (LOC) as a metric, his research points at 5 to 50 bugs being introduced per thousand lines of code (KLOC). To put this in perspective, in 2011, the Linux kernel reached 15 million LOC[11]. In terms of hidden behavior, the more complex a piece of software is, the more difficult it is to evaluate its behavior under all possible circumstances. As a consequence, undocumented behavior can be intentionally or unintentionally introduced, and remain undetected for years, or decades. In general, as software grows in complexity, the chances of it exhibiting unspecified behavior increase.

From a security perspective, unspecified behavior is synonym with indeterminism, which is the corner stone for software attacks. A third party that discovers a piece of software exhibiting unspecified behavior can exploit it for different reasons and motivations: to report the vulnerability, steal sensitive data, or compromise system assets. The fact that the two last options exist is the basis for cybercrime.

[7] http://www.theverge.com/2015/1/17/7629721/nsa-is-pwning-everyone-and-having-a-chuckle-about-it

[8] http://archive.wired.com/techbiz/media/news/2002/01/49826

[9] http://www.forbes.com/sites/oracle/2014/09/30/larry-ellison-on-oracles-cloud-strategy-comprehensive-cutting-edge-secure/

[10] http://blog.ted.com/2014/03/19/computing-is-still-too-clunky-charlie-rose-and-larry-page-in-conversation/

[11] http://arstechnica.com/business/2012/04/linux-kernel-in-2011-15-million-total-lines-of-code-and-microsoft-is-a-top-contributor/

1.2 Problem

An important question for the IT industry is then: *How do we handle the security of complex software without killing innovation?* We can find three answers to this question (which are not mutually exclusive): (i) by teaching adequate programming skills and ethics to future developers at what can be called *education-time*, (ii) by improving software development tools in order to detect vulnerabilities and abusive behavior at *compilation-time*, or (iii) by defining a security perimeter that allows to protect sensitive data and monitor the execution of complex services to satisfy a number of privacy-, integrity-, security-, etc. oriented policies at *run-time*.

At *education-time*, computer science educators should teach future generations of software developers to write more secure code [280], apply privacy-oriented techniques [86], and convey with programming ethical codes[12],[13]. In terms of ethics, the NIS education programs in Europe, which made public their roadmap in late 2014 [109], are a good example of how to impact IT studies from a social, legal, and ethical perspective. In terms of code quality, it is a reality that today not all programmers have a background in computer science. Higher abstractions hiding complexity (e.g., user space - kernel space separation, object-oriented programming, software libraries) have made it possible for more and more people to implement complex services writing simple code, probably without understanding the full implications that their code have in the system. There is no doubt that this has an invaluable positive effect on multidisciplinary innovation (e.g., liberal arts, architecture) and research (e.g., medicine, mathematics), but it also means that vulnerabilities introduced in these abstractions affects a larger number of services and devices. The consequences of vulnerable software are bigger than ever before.

At *compilation-time*, some have pointed to the need to improve programming tools such as editors, languages, or compilers in order to make them both (i) more available to programmers, and (ii) more strict about the code they take as an input. In [235], Marcus Ranum makes an interesting reflection about how these tools in Unix environments have become an anachronism in today's development processes, the trade-offs between features and bugs in powerful (but more error-prone) programming languages such as C, and how it altogether affects the quality of the code that is produced today. This is what in developer's jargon responds to the *it's not a bug, it's a feature* [279]. While education is able to positively impact the code that will be generated in the future, the truth is that legacy code - which might contain unspecified behavior -, still plays a major role in our systems today, and the cost of substituting it is unbearable [286]. Bugs such as *Heartbleed*[14], *Shellshock* [287], or the latest *git* bug[15] are good examples of this problem.

- Heartbleed (CVE-2014-0160), described by Bruce Schneier as a *"catastrophic bug"*, is a vulnerability in openSSL that allowed an attacker to carry out a buffer over-read[16] and

[12] `http://www.acm.org/about/se-code`

[13] `http://www.ibm.com/developerworks/rational/library/may06/pollice/`

[14] `http://heartbleed.com`

[15] `http://arstechnica.com/security/2014/12/critical-git-bug-allows-malicious-code-execution-on-client-machines/`

[16] `http://cwe.mitre.org/data/definitions/126.html`

exposed data in motion. The bug was introduced in OpenSSL in December 2011 and has been out in the wild since OpenSSL release 1.0.1 on 14th of March 2012. OpenSSL 1.0.1g released on 7th of April 2014 fixes the bug.

- Shellshock (CVE-2014-6271) is a bug in *bash* that allowed an attacker to execute arbitrary shell commands on a *bash* console. This could result on an attacker gaining control of a remote machine. The bug was introduced with Bash v1.03 in 1989[17] and was publicly disclosed on 24th of September 2014.

- The latest git bug (CVE-2014-9390) is a bug in git that allowed a remote attacker to change git's configuration file (*.git/config*), thus allowing the execution of arbitrary code. As in the case of Shellshock, this could result on an attacker gaining control of a remote machine. This bug was disclosed on 18th of December 2014.

At *run-time*, different approaches have targeted the definition of a security perimeter to protect *Sensitive Assets*. We define sensitive assets as any software or hardware component that manages sensitive information (e.g., personal photographies, archived mails, or encryption keys) in any way. Examples of sensitive assets include webcam and microphone devices, input devices such as keyboards or touch screens, or secondary storage devices where sensitive data at rest resides. Also, this security perimeter has been used to monitor the actions taken by a device in order to enable different kinds of integrity guarantees. Defining this security perimeter has been the focus of a large number of software and hardware solutions. In general, protecting sensitive assets and monitoring a system at run-time is a dynamic problem, based on protection from well defined attack vectors. These include software and hardware techniques that allow to steal or compromise sensitive assets. Examples of attack vectors go from simple brute-force techniques or social engineering to software attacks targeting code bugs, or complex physical attacks that modify the hardware properties of a device.

In this thesis, we focus on the third option: protecting a device at run-time. We denote this *Run-Time Security*. While we hope that future developers will produce honest, bug-free code, we assume that services will, for the time being, exhibit unspecified behavior. Thus, in the short- and mid-term we need run-time security primitives that allow to protect sensitive assets that users identify from being leaked, exploited, or misused. Our hypothesis is that it is possible to define a hardware-protected security perimeter to which the main operating system and applications do not have access to. Such security perimeter defines an area where sensitive assets reside. Our research question is then: *Can we define a set of run-time security primitives that enable innovative services to make use of sensitive assets, while guaranteeing their integrity and confidentiality? What is more, can we add support for these run-time security primitives to commodity operating systems?*

1.3 Approach

Our approach is to provide *Run-Time Security Primitives* that allow innovative services to make use of sensitive assets. One of our main objectives is adoption, therefore providing

[17] `http://thread.gmane.org/gmane.comp.shells.bash.bugs/22418`

support for these primitives in commodity operating system is a strong requirement that we impose. We focus on Linux-based systems given Linux's wide adoption, and the fact that it is open source.

Our first step is to find a piece of existing secure hardware that can define a security perimeter to divide a device in two hardware areas: a *Trusted Area*, where the hardware components dealing with sensitive assets belong to, and an *Untrusted Area* composed by the rest of the hardware components. Depending on the secure hardware capabilities, this division can be more or less dynamic at run-time. In this way, two different software execution environments emerge, a *Trusted Execution Environment (TEE)* where software components using resources from the trusted area execute, and a counterpart for the untrusted area that we denote *Rich Execution Environment (REE)*.

Having such an environment, we can then provide a series of *Trusted Services* that guarantee the integrity and confidentiality of sensitive assets in a running environment. Trusted services are made available to services in the untrusted area through run-time security primitives. In this way, innovative services can make use of sensitive assets without compromising them. The challenge is then to define how the trusted and untrusted areas should integrate in order to: (i) minimize overhead for innovative services, and (ii) maximize the protection of the sensitive assets used by trusted services. Put differently, we need to expose trusted services to the interfaces that are already known to innovative services in such a way that this does not negatively impact the process of developing innovative services. The extent and adoption of these interfaces, the level of integration between innovative and trusted services, and the security guarantees defining the trusted area are then the three variables that we have available to design and implement our approach.

While existing approaches have already been defined, they are not well suited for either current software, or current security threats. On the one hand, solutions that offer a high-level of tamper-resistance can protect sensitive assets against large classes of intricate attacks, but they restrict the scope of the applications they can support. On the other hand, solutions that integrate with existing software cannot face the security threats that respond to today's cyberattacks.

In order to build run-time security primitives and their correspondent trusted services we follow an **experimental approach**. We design, implement, and evaluate our solution. Note that, while trusted services are the visible objective, our research centers on providing support for these services in commodity operating systems, since it is where the state of the art in run-time security has its main shortcomings. In terms of performance, we are interested in evaluating the impact that run-time security has on a running system. This evaluation will be entirely experimental. In terms of resilience to attacks, we will provide an analytical study of the security and scope of our solution both in relation to the proposed concept and our actual implementation of it. We will additionally provide an evaluation with regards to the requirements that we have established. All in all, we intend to push the state of the art in run-time security and make our research available by (i) contributing to open source projects such as the Linux kernel, (ii) using standard interfaces and methodologies, and (iii) documenting and publishing our research to maximize its adoption.

1.4 Contribution

Our contribution is threefold. First, we provide an exhaustive analysis of the state of the art in run-time security. Here, we analyze the different types of secure hardware that can enable a TEE, we present the theoretical work targeting different aspects related to run-time security (e.g., memory protection mechanisms, usage control models), and finally we cover concrete solutions to run-time security in different research communities (e.g., security community, virtualization community, systems community). Altogether, this first analysis represents an updated look at the run-time security landscape from boot protection mechanisms to run-time policy enforcement.

Second, we argue for the concrete hardware and software combination we choose for our solution. We use ARM TrustZone security extensions for the definition of the security perimeter, and Open Virtualization to leverage these security extensions and provide a TEE. Here, we first analyze Open Virtualization and present our contributions to it. Then, we discuss how a TEE can be generalized, and report on the design and implementation of a generic TrustZone driver for the Linux kernel. This driver represents the first proposal to provide TrustZone support in the Linux operating system. At the time of this writing, we have submitted a first set of patches with the generic TrustZone driver to the *Linux Kernel Mailing List (LKML)* with the intention of contributing our work to the mainline Linux kernel.

Third, we report on the design and implementation of a *Trusted Cell*, i.e., a distributed framework that leverages the capabilities of a given TEE to provide *Trusted Services*. Using this framework, we implement two trusted services that we consider pivotal to run-time security: (i) a trusted storage solution that ensures the confidentiality and integrity of sensitive data while reusing legacy infrastructure to minimize the *Trusted Computing Base (TCB)*; and (ii) a reference monitor that allows to enforce *Usage Control* policies targeting a wide range of *Sensitive Assets* in a device (e.g., sensitive data, memory, peripherals). Combining these two trusted services we can relax the trust assumptions that have shaped the security community in the last years. Both the trusted cell and the two trusted services make use of the generic TrustZone driver presented above to leverage the TEE. We also provide an exhaustive security analysis and evaluation of the overall architecture, as well as the design and implementation of each component.

This thesis is partially based on the three publications and two technical reports listed below. We are also working on two other publications based on contributions that we present in this thesis. We will refer to them when we describe those contributions.

- J. González and P. Bonnet. Towards an open framework leveraging a trusted execution environment. In *Cyberspace Safety and Security*. Springer, 2013
- J. González, M. Hölzl, P. Riedl, P. Bonnet, and R. Mayrhofer. A practical hardware-assisted approach to customize trusted boot for mobile devices. In *Information Security*, pages 542–554. Springer, 2014
- P. Bonnet, J. González, and J. A. Granados. A distributed architecture for sharing ecological data sets with access and usage control guarantees. 2014

- J. González and P. Bonnet. Versatile endpoint storage security with trusted integrity modules. ISSN 1600–6100 ISBN 978-87-7949311-7, IT University of Copenhagen, February 2014
- J. González and P. Bonnet. Tee-based trusted storage. ISSN 1600–6100 ISBN 978-87-7949-310-0, IT-Universitetet i København, February 2014

Two extra workshop publications define where our focus for future research lays.

- M. Bjørling, M. Wei, J. Madsen, J. González, S. Swanson, and P. Bonnet. Appnvm: Software-defined, application-driven ssd. In *NVMW Workshop*, 2015
- M. Bjørling, J. Madsen, J. González, and P. Bonnet. Linux kernel abstractions for open-channel solid state drives. In *NVMW Workshop*, 2015

1.5 Thesis Structure

This thesis is organized in three main parts. Part I is dedicated to the state of the art in run-time security; Part II is dedicated to our contribution to it; Part III is dedicated to evaluating our contribution.

In Part I, we cover the state of the art in run-time security. In Chapter 2 we look at *Trusted Execution Environments (TEEs)* and the different secure hardware approaches that can leverage it. In Chapter 3 we look into theoretical and practical approaches to run-time security. Here, we will review work in very different communities that have solve specific problems relating to security in general: from the conceptual definition of usage control and its formal model, to platform-specific memory protection mechanisms used in the virtualization community. In Chapter 4, we cover the different boot protection mechanisms that can provide a root of trust, as required for run-time security to exist. Finally, in Chapter 5, we discuss about the state of the art in run-time security and how different works relate to each other. We use this discussion to pre-motivate our contribution in Part II.

In Part II, we present our contribution. In Chapter 6 we introduce the conceptual contribution of this thesis. Here, we present our hypothesis, discuss the design space, and finally argue for a concrete design path. In Chapter 7, we report on the design an implementation of our operating system support proposal for a TEE based on the ARM TrustZone security extensions. In Chapter 8, we use this operating system support to provide *Trusted Services*. In Chapter 9, we present a novel boot protection mechanism that leverages the framework formed by a TEE, operating system support, and trusted services. These four chapters are the core contribution of this thesis.

In Part III we evaluate our contributions. In Chapter 10 we analyze the security of our proposals in terms of the trust assumptions we have made and the attacks that they can protect against. We also provide an analytical analysis in terms of requirement compliance and architecture. In Chapter 11, we provide a complete experimental evaluation of our approaches focusing on performance impact.

Finally, in Chapter 12, we conclude this thesis with a summary of our contributions, a general evaluation of our work, and a roadmap for future work.

A glossary is available at the end of this thesis. We invite the reader to consult it at any time. We point to the terms we are using persistently throughout the whole thesis in order to help the reader understanding our explanations. If in doubt, please consult it.

Part I

State of the Art

In Part I, we cover the state of the art in the three areas that define run-time security. In Chapter 2, we look at *Trusted Execution Environments (TEEs)* and the different secure hardware approaches that support them. In Chapter 3.3 we look into theoretical and practical approaches to run-time security. Here, we review work in different research communities that have solve specific problems that relate to security in general: from the conceptual definition of usage control and its formal model, to platform-specific memory protection mechanisms used in the virtualization community. In Chapter 4, we cover the different boot protection mechanisms that provide the root of trust that is a prerequisite for run-time security. Finally, in Chapter 5, we summarize our view on run-time security based on the state of the art exposed in the preceding chapters. We use this discussion to motivate our contribution in Part II.

A glossary is available at the end of this thesis. We invite the reader to consult it at any time. We point to the terms we are using persistently throughout the whole thesis in order to help the reader understanding our explanations. If in doubt, please consult it[18].

[18]Note that if this thesis is read in PDF format, all terms link to its entry in the Glossary (12.2.3).

Chapter 2

Trusted Execution Environments

What is a Trusted Execution Environment? Before we answer this question, we need to define execution environments in general. At a high level of abstraction, an execution environment is the software layer running on top of a hardware layer. Both hardware and software layers are combined to fom a *Device*. In this thesis, we focus on a class of device that contains two execution environments that are physically separated. One environment contains the main OS and applications, the other environment contains trusted software components. We thus have a physical separation between the *Trusted Area* and the *Untrusted Area*. The trusted area is not intrinsically trusted; no software executing in it, and no hardware attached to it offers more guarantees than an equivalent outside of the security perimeter. However, since the trusted area is separated by hardware from OS and applications, its isolation is guaranteed. Everything outside the trusted area is untrusted. Each area features a different execution environment. This means that a device has two different software stacks. We denote the execution environment in the trusted area *Trusted Execution Environment (TEE)*, and the one in the untrusted area *Rich Execution Environment (REE)*. Indeterministic software in the REE cannot affect software running in the TEE. We depict this definition in Figure 2.1.

The term TEE was first used by the Global Platform consortium[1] to define *"a secure area that resides in the main processor of a smart phone (or any mobile device) and ensures that sensitive data is stored, processed and protected in a trusted environment."*. Also, Trustonic, the joint venture between ARM, Gemalto, and Giesecke & Devrient, has defined a TEE as *"a secure area that resides in the application processor of an electronic device"*[2]. For us, a TEE is the software stack in the trusted area, as depicted in Figure 2.1; it is not specific to a type of devices (e.g., smart phones), it does not uniquely deal with sensitive data, and it is not necessarily defined by an area in a processor.

In this Chapter we will look at the technology that can provide a TEE. In this way, we will describe the different alternatives that define the state of the art in hardware security. From isolated cryptographic co-processors to security extensions targeting the CPU or the

[1] `http://www.globalplatform.org`
[2] `http://www.trustonic.com/products-services/trusted-execution-environment`

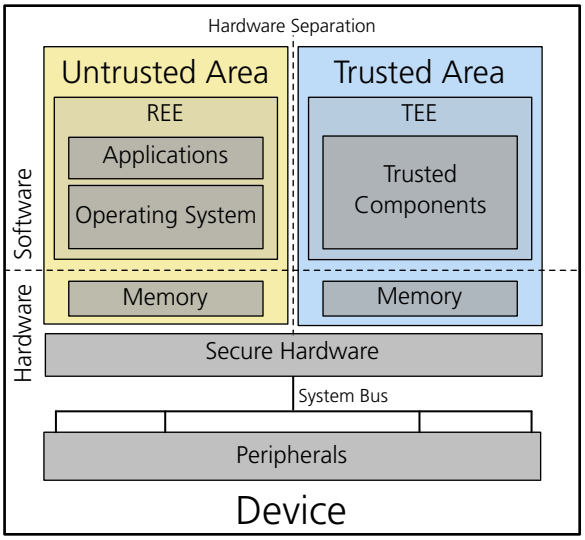


Figure 2.1: *Trusted Execution Environment (TEE)* definition. In order to support a TEE, a device needs to define a security perimeter separated by hardware from the main OS and applications, where only trusted code executes. We refer to this security perimeter as *Trusted Area*. The trusted area is represented on the right side of the figure (blue), where trusted components execute in a Trusted Execution Environment (TEE). Everything outside the trusted area conforms the *Untrusted Area*, where OS and applications execute in a *Rich Execution Environment (REE)*. The untrusted area is represented on the left side of the figure (yellow). Peripherals connected to the system bus are can belong to either of the two areas, or both of them. This depends on the specific technology.

system bus, these technologies share a common ground: the definition of a trusted and an untrusted area. How each specific technology determines the interplay between these two areas, as supported by the hardware underneath them, will ultimately define the TEE for that specific technology. Thus, determining the support that each technology provides for run-time security, and utterly the *Trusted Services* that are going to be supported. We will retake this discussion at the end of this chapter, once we have covered the different approaches to hardware security that can today be found in the market. We present these technologies in an increasing order of integration of the trusted and untrusted areas.

2.1 Smart Cards

A smart card is an embedded integrated circuit card with a CPU, memory, and at least one peripheral interface to communicate with a host device. Given the attack model under which they are designed [179, 180], smart cards are normally considered to have a high level of tamper-resistance. There are two international standards for smart cards: one for contactless smart chips (ISO/IEC 144430), and another for contact smart chips (ISO/IEC 7816).

The trusted area defined by a smart card is very primitive: every component in the smart

card belongs to the trusted area; every component not in the smart card belongs to the untrusted area (Figure 2.2). Put differently, the separation between the trusted and the untrusted area in smart cards is sharp. Since there is a complete physical isolation of the trusted area, smart cards can be very small, which also contributes to achieving higher levels of tamper-resistance. This comes however at the cost of having reduced RAM and a narrow hardware communication interface. The combination of their low computational power, narrow interfaces, and the isolated trusted area they define makes that trusted services leveraged by a smart card are very secure, but very limited in scope.

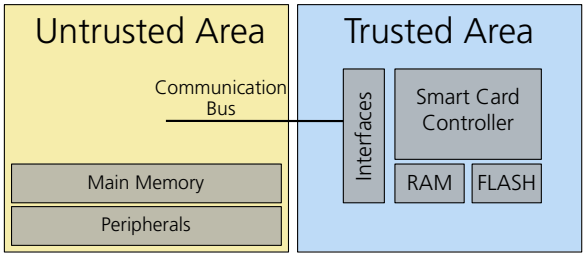


Figure 2.2: Smart Card TEE definition. The trusted area corresponds to the Smart Card chip, where all its components are located. Everything interacting with the smart card is considered untrusted. Examples of smart card peripheral interfaces include USB and BlueTooth. Main memory and peripherals belong exclusively to the untrusted area. The smart card is indeed a peripheral.

Typically, smart cards are used to power credit cards and enable money payments. Some work, leaded by the SMIS[3] group at INRIA, France, have focused on implementing a secure database inside of a smart card [55] as a way to protect and process sensitive data. Their work include PicoDBMS [231], GhostDB [9], MILo-DB [11], and secure personal data servers (PDS) [5]. Other relevant work outside SMIS include TrustedDB [31]. All these approaches conform the research with regards to hardware-assisted data protection using smart cards. Here, the state of the art is mainly divided between centralized solutions (e.g., TrustedDB [31]), and decentralized solutions, many of them also discussed in [208]. Independently from the approach, the smart card plays a fundamental role in each design given its high level of tamper-resistance [179, 198].

So high are the expectations put into the security that smart cards can leverage that some work, specially in form of patents, see the smart card as the medium to materialize the digital passport [244], and other digital personal identification mechanisms [238, 183]. Other emerging use cases for smart cards are also discussed in [258].

Commercial secure processors implemented in the form of smart cards include ARM SecurCore[4], Atmel TwinAVR[5], and Texas Instruments GemCore[6].

[3] `http://www-smis.inria.fr`

[4] `http://www.arm.com/products/processors/securcore/`

[5] `http://www.atmel.com/products/microcontrollers/`

[6] Not much information about TI GemCore is public, but based on driver maintenance and public announces this family of processors seems to be operative and maintained.

2.1.1 Secure Element

A secure element (SE) is a special variant of a smart card, which is usually shipped as an embedded integrated circuit in mobile devices together with Near field Communication (NFC) [195]. It is already integrated in a multitude of mobile devices (e.g., Samsung Galaxy S3, S4, Galaxy Nexus, HTC One X). Furthermore, a secure element can also be added to any device with a microSD or an Universal Integrated Circuit Card (UICC).

The main features of a SE are:

- **Data protection** against unauthorized access and tampering.
- **Execution of program code** in form of small applications (applets) directly on the chip. These small applications (applets) can perform operations on data and keys directly on the card without leaving the TEE.
- **Hardware supported execution of cryptographic-operations** (e.g., RSA, AES, SHA, etc.) for encryption, decryption and hashing of data without significant run-time overhead [147]. Since the SE provides tamper-resistant storage, the SE is able to protect stored data against unauthorized access and tampering. Access to this data is only possible over a standardized interface, controlled by the OS using Application Package Data Units (APDU) [234]. This characteristic makes the secure element a good candidate for storing private keys. Still, attacks such as power-analysis [177], or differential fault-analysis [177] should be considered.

2.2 Trusted Platform Module (TPM)

The Trusted Platform Module (TPM) is a specification for a secure crypto co-processor defined by the Trusted Computing Group (TCG)[7], and made an international standard (ISO/IEC 11889)[8] in 2009. The TCG defines the TPM as a generator storage device and protector of symmetric keys [274], but in general, it is used to store state, keys, passwords, and certificates.

At the time of this writing TPM 2.0 specification is still relatively new[9], and coexists with TPM 1.2. For the purpose of this thesis the differences between TPM 1.2 and 2.0 and irrelevant; we are only interested in the general architecture and components forming a TPM, so that we can compare it to other secure hardware approaches. Concrete interface, commands, and other specification differences have still not been addressed in a publication, however, the topic has been discussed in different oral presentations [278, 120, 288, 61].

Architecturally, TPM is formed by several components; some are required and some are optional. This depends on the specification version. The most relevant components are:

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

[8] `http://www.iso.org/iso/catalogue_detail.htm?csnumber=50970`

[9] TPM specification version 2.0 was published for public review on March 13, 2014

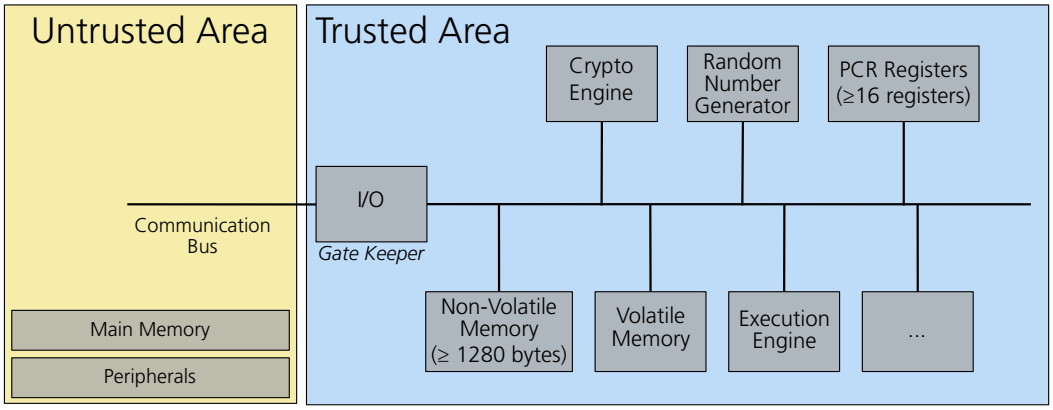


Figure 2.3: Trusted Platform Module TEE definition. The trusted area corresponds to the TPM chip. Everything outside the TPM is considered untrusted. Main memory and peripherals belong exclusively to the untrusted area. The TPM is indeed a peripheral.

- **I/O component** is a gatekeeper mechanism that manages the information flow in the communication bus. It enforces access control policies to the TPM, performs encoding and decoding of commands passed through internal and external buses, and routes them to the appropriate component.
- **Execution Engine** is the component where the TPM components execute. The execution engine is a vital component since it is the one defining TPM's TEE. This guarantees that operations are properly segregated and allows to define shielded locations. The TCG defies a shielded location as an area where data is protected against interference from the outside exposure.
- **Platform Configuration Register (PCR)** is a 160-bit storage location for discrete integrity measurements. There are a minimum of 16 PCR registers. All PCR registers are shielded- locations and are inside of the TPM. The decision of whether a PCR contains a standard measurement or if the PCR is available for general use is deferred to the platform's implementation of the specification.
- **Crypto Engine** implements cryptographic operations within the TPM. In TPM 1.2 the specification requires a SHA-1 engine, a RSA engine and a HMAC engine. It also requires key sizes of 512, 1024, 2048 bits. TPM 2.0 is more flexible and does not specify algorithms or key sizes in order to prevent early deprecation[10].
- **Volatile Memory** is used to cache state and sensitive data inside of the TPM so that it is protected against interference from outside exposure.
- **Non-Volatile Memory (NVRAM)** is used to store persistent identity and state associated with the TPM. The specifications require at least 20 bytes of NVRAM; most implementation however follow the recommendation of providing 1280 bytes.

[10]The US Department of Commerce has considered SHA1 as deprecated from 2011 to 2013, and disallowed after 2013 (NIST SP800-131A) [34]; Microsoft is not accepting SHA1 certificates after 2016; Google is penalizing sites using SHA1 certificates expiring during 2016. TPM 1.2 requiring SHA1 results in it being inevitably subject to expiration.

- **Random Number Generator (RNG)** is the source of randomness in the TPM. The TPM uses these random values for nonces, key generation, and randomness in signatures. The RNG is also used to hold shielded locations.

Regardless of the implemented components, from the TPM perspective, everything behind the I/O component is trusted. The rest of the system is untrusted. In this way, the definition of the TEE leveraged by a TPM resembles that of a smart card. However, since the TPM embeds more components in its trusted area, the interface with the untrusted area is broader. Figure 2.3 depicts these components and the area to which they belong.

In terms of capabilities, TPM provides more functionality than generation and storage of cryptographic keys:

- **Remote attestation** creates an unforgeable summary of the hardware, boot, and host OS configuration of a computer, allowing a third party (such as a digital music store) to verify that the software has not been changed. In Section 4 we will see how this capability is used for protecting the boot process in *trusted boot*.
- **Binding** allows to encrypt data using the TPM's endorsement key (i.e., a unique RSA key put in the chip during its production) or another trusted key generated by the TPM. The asymmetric keys used for binding can be migratable or non-migratable storage keys. If non-migratable storage keys are used, the encrypted data is bound to a specific platform.
- **Sealing** allows to encrypt data and associate the used encryption key with a specific TPM state in the PCR registers. In this way, data can only be decrypted if the TPM is in the exact same state. The state can be associated with one or several components in the boot sequence, the integrity of a binary file, or a TPM internal state. It can be seeing as an extension to *binding*.

Since TPM is a co-processor, it needs to be attached to a host system. If a physical co-processor is used (i.e., external TPM), a physical interface has to be used to attach it to the host communication bus. Today, I^2C [254] and SPI [184] are the most popular interfaces supported by external TPMs (Table 2.1). Low Pin Count (LPC) is also supported by some vendors, however ad-hoc connectors are necessary to match vendor-specific pin layouts.

While an external TPM is an easy way to add TPM functionality to an existing design, the fact that the TPM is attached does that its level of tamper resistance, and specially its level of tamper evidence is weakened. Example attacks to external TPMs include targeting external, visible pins [292, 181] and casing [269].

It is also possible to connect a TPM logically, as long as the TPM is built using components that are already available in the host system [130]. In this case, the TPM's components (Figure 2.3) need to be implemented by parts in the host that can fulfill TPM's requirements. If the host has a FPGA, isolation requirements such as the ones established for the TPM's execution environment are easier to meet without sacrificing a physical processor. The main issue with this approach comes from the fact that a TPM requires secure non-volatile

I^2C	SPI
2-wire serial communication protocol	4-wire serial communication interface
Standard protocol	Established protocol, but no standard
Speed (max. throughput) up to 3.4 Mbps	No throughput limitation (no max. clock speed)
Support for multiple devices on same bus	Arbitrary choice of message size, content, and purpose
Ensures that data is received by the slave device	Very low power

Table 2.1: Principal characteristics of the I^2C and SPI protocols.

memory. Thus, if the host does not count on tamper-resistant storage, TPM requirements cannot be met. Such TPM implementation is commonly referred to as a *soft TPM*. This is a recurring problem in mobile devices, where there have been attempts to build a Mobile TPM (MTPM, or MPM) using ARM TrustZone [92, 290] without using a tamper-resistant unit for providing secure non-volatile memory; as we will see in Section 2.4, TrustZone is not tamper resistant.

2.3 Intel Security Extensions

2.3.1 Intel TXT

Intel Trusted Execution Technology (Intel TXT, formerly referred to as LaGrande Technology) is a set of hardware security extensions to the Xeon processor family with the objective of providing a *dynamic root of trust for measurement* (DRTM), so that a system can be attested for integrity at run-time. Intel TXT uses a specific set of CPU instructions - Safer Mode Extensions (SMX) [161] - which implement: (i) measured launch of the Measured Launched Environment (MLE), (ii) a set of mechanisms to ensure such measurement is protected and stored in a secure location, and (iii) a set of protection mechanisms that allow the MLE to control attempts to modify itself [163]. Intel TXT depends on the TPM's PCR registers [274] to store its integrity measurements; thus the TPM is the root of trust for Intel TXT (Figure 2.3). In a way, Intel TXT completes the purpose of the TPM. In the TPM specifications [274, 276] the TCG presents the procedures by which the TPM seals measurements in the PCRs. How the PCRs are updated is also covered in the TPM specification. However, no procedures are specified to ensure the quality of the measurements in themselves. Indeed, most of the TPM literature avoid this discussion, assuming that the measurements are ready for sealing. While DRTM is the base for Intel TXT, it is not exclusively used by it; DRTM is also the mechanism supporting other technologies such as AMD Secure Virtual Machine (SVM) security extensions [7], which is used in works such as OSLO [171].

To put Intel TXT into context, the TCG defines *static root of trust for measurement* (SRTM) as a fixed or immutable piece of trusted code in the BIOS that is executed before the boot

process starts, in such a way that every piece of code in the boot process is measured by it (the trusted code) before it is executed. In this way, SRTM aims at measuring the integrity of sensitive fragments of code before they are executed. While in theory every component in the boot process can be measured - with the consequent performance cost -, SRTM only guarantees that the boot system is in a know state, not providing mechanisms for run-time integrity checks. TrustedGrub[11] is an example implementation of SRTM. DRTM approaches this limitation, and extends SRTMS in such way that run-time measurements are supported; DRTM allows to load a measured environment without requiring a power down and reset.

At boot-time, i.e., during BIOS load, a SMX instruction (GETSEC) is issued to ensure that only two code modules are loaded into memory: the MLE, and the authenticated code module (ACM)[12]. After this, measurements are stored in the PCR registers. Thus, the resilience of Intel TXT depends the PCR registers not being modifiable, but only extensible. In this way, the hash of the concatenation of the current PCR value and the new measurement is stored in the PCRs. In order to provide a chain of trust, this process begins during BIOS load as mentioned above. Since TPM is orthogonal to Intel TXT, it has support for both SRTM and DRTM: PCRs [0 to 15] are used for SRTM, and PCRs [17 to 20] are used for DRTM when in *locality 4* [276] (SMX) [253]. A complete description of the DRTM launch sequence, measurement management, and TPM usage detailing how different attack vectors are faced (e.g., DMA protection via IOMMU disabling) can be found in Intel's TXT Software Development Guide [163].

One of the main applications of Intel TXT is ensuring a chain of trust at boot-time. An example is *Trusted boot* - and one of its implementations, (tboot)[13]. In Chapter 4 we will look in more detail into boot protection mechanisms.

2.3.2 Intel SGX

Intel Software Guard Extensions (Intel SGX) is a set of instructions and mechanisms for memory accesses to be incorporated in future Intel processors [159, 162, 157]. These extensions allow an application to instantiate a protected container, referred to as an *enclave*. An enclave is defined as a protected area in the application's address space which cannot be altered by code outside the enclave, not even by higher privileged code (e.g., kernel, virtual machine monitors, BIOS). In other words, enclaves are orthogonal to x86 protection rings [161]. SGX also prevents that code and data in one enclave are accessed from other enclaves (i.e., inter-enclave isolation). Figure 2.4 depicts an enclave within an application's virtual address space.

If we look at the extensions independently, Intel SGX provides the following protection mechanisms [204, 8]:

- **Memory protection.** Since an enclave is part of the application address space (user space), it is necessary to protect it against the many software and hardware attacks

[11] `http://sourceforge.net/projects/trustedgrub/`
[12] ACM is also referred to as AC (authenticated code) module.
[13] `http://sourceforge.net/projects/tboot/`

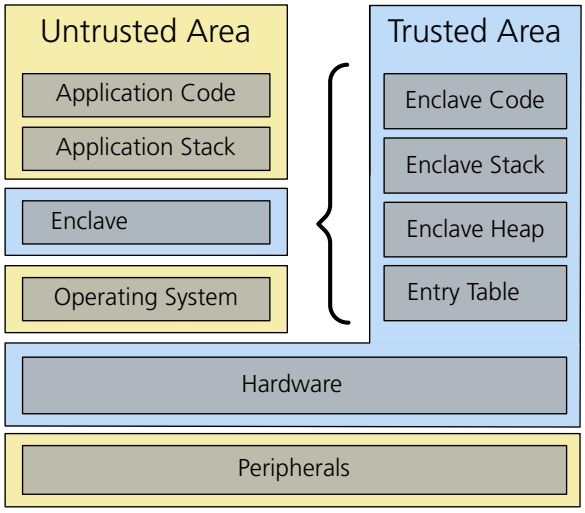


Figure 2.4: Intel SGX's Enclave Architecture [204]. The trusted area is formed by the memory enclave, the rest belongs to the untrusted area. Note that peripherals are only accessed from the untrusted area since SGX does not extend to the system bus.

against main memory (e.g., DRAM) [56]. For this purpose, SGX uses the Enclave Page Cache (EPC), which is a piece of cryptographically protected memory with a page size of 4KB. In order to encrypt main memory, guarantee its integrity, and protect the communication with the processor, a hardware unit, i.e., the Memory Encryption Engine (MEE), is used. For the processor to track the contents of the EPC, the security attributes for each page of the EPC are stored in an implementation-dependent micro-architecture structure called Enclave Page Cache Map (EPCM).

Enclaves are created using the $ECREATE$ instruction, which converts a free EPC into a SGX Enclave Control Structure (SECS) and initializes it in protected memory. From that moment, pages can be added using $EADD$. Even though pages are allocated transparently by the OS, they must be mapped to the EPC in physical memory. Each EPC page is tracked in hardware in terms of its type, the enclave to which it is mapped, the virtual address within the enclave, and its permissions.

Just as main memory, the EPC is a limited resource. Therefore, SGX enables the OS to virtualize the EPC by paging its contents to other storage. The specific mechanisms used to protect the virtualized EPC can be found in [158, 204, 39].

- **Attestation.** SGX's attestation mechanisms are supported directly by the CPU [8]. The idea is the same as in TPM: a remote entity is able to verify the integrity of the trusted area and the TEE. In SGX, the remote entity can cryptographically verify an enclave and create a secure channel for sharing secrets with it. In this way, the remote party can load trusted code in an enclave, and verify that is has been properly instantiated in the target platform afterwards. This is done in a per-enclave basis.

  SGX supports intra- and inter-platform enclave attestation. This allows enclaves to securely communicate with both other enclaves and remote entities. SGX provides different mechanisms for *local attestation* and *remote attestation*.

Ultimately, the processor manufacturer (e.g., Intel) is the root of trust for attestation. i.e., Root of Trust of Measurement (Figure 2.4).

- **Sealing.** SGX guarantees that when an enclave is instantiated, the hardware provides protections (confidentiality and integrity) to its data, when it is maintained within the boundary of the enclave. It also guarantees that when the enclave process exits, the enclave will be destroyed and any data that is secured within the enclave will be *zeroized*. If the data is meant to be reused later, the enclave must make special arrangements to store the data outside the enclave. Persistent sealing keys are accessed through *EGETKEY* [158]. Sealing keys can be used to enforce sealing policies such as *Sealing to the Enclave Identity*, and *Sealing to the Sealing Identity*. [8].

Although Intel SGX is fairly new at the time of this writing, some work exploring use cases has already been done either on simulators or on early prototypes, and presented in form of demos at some conferences and venues [40]. In [143], Hoekstra et al. (Intel Corporation) used SGX to implement: (i) a one-time password (OTP) generator for second factor verification without the need of RSA SecureID®; (ii) a secure enterprise rights management (ERS) client-server architecture; and (iii) a secure video-conferencing application. Even though no performance evaluation or security analysis are presented[14], it is at least possible to see the way Intel envisions the use of SGX across cloud services as a way to load trusted services in remote, untrusted environments. In [39], Baumann et al. propose Haven, i.e., a framework to implement shielded execution of unmodified Windows applications using SGX's enclaves as their TEE. We expect to see much more work targeting cloud services using Intel SGX, specially because of its light-weight remote attestation [8], and the possibility to remotely load trusted code in the TEE defined by the SGX security extensions, even when the remote system is untrusted.

2.4 ARM TrustZone

ARM defines TrustZone as a hardware-supported system-wide approach to security which is integrated in high-performance processors such as Cortex-A9, Cortex-A15, and Cortex-A12 [29]. Today, TrustZone is implemented in most ARM modern processor cores including the ARM1176, Cortex-A5/A7/A8/A9/A15, and the newest ARMv8 64-bit Cortex-A53 and -A57. TrustZone relies on the so-called NS bit, an extension of the AMBA3 AXI system bus to separate the execution between a *secure world* and a *non-secure world* (AWPROT[1] and ARPROT[1] [20]). The NS bit distinguishes those instructions stemming from the secure world and those stemming from the non-secure world. Access to the NS bit is protected by a gatekeeper mechanism referred to as the secure monitor, which is triggered by the System Monitor Call (SMC). Any communication between the two worlds is managed by the secure monitor. Interrupts can also be configured in the secure world to be managed by the secure monitor. From a system perspective, the OS distinguishes between user space, kernel space

[14]Neither on the complementary SGX publications [204, 8]. In [56] however, the properties and attack model for designing a secure container are described in the context of Intel processors and their security extensions.

and secure space. Authorized software runs in secure space, without interference from user or kernel space. Secure space can only be accessed from kernel space.

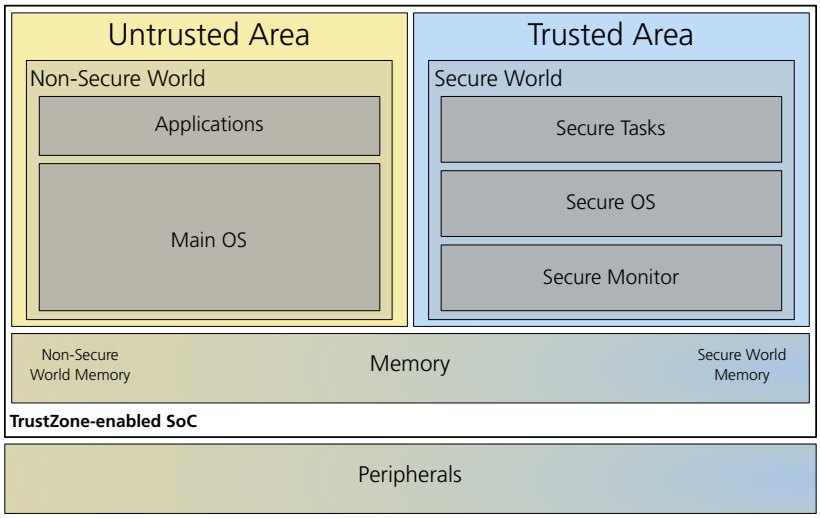


Figure 2.5: TrustZone TEE definition. The trusted is formed by the chip and the part of the system protected by the NS bit. The rest of the system corresponds to the untrusted area. In TrustZone's nomenclature the secure world represents the TEE, and the non-secure world the REE. Since TrustZone extends the NS bit to the system bus, peripherals can be secured.

TrustZone's most powerful feature is that it gives the ability to secure any peripheral connected to the system bus (e.g., interrupt controllers, timers, and user I/O devices) in a way that they are only visible from the secure world. The AMBA3 specification includes a low gate-count low-bandwidth peripheral bus known as the Advanced Peripheral Bus (APB), which is attached to the AXI system bus using the AXI-to-APB bridge. This is an important part in the TrustZone design, since the APB bus does not carry the NS bit so that AMBA2 peripherals are supported in TrustZone-enabled devices. The AXI-to-APB bridge hardware is then responsible for managing the security of the APB peripherals. Peripherals can also be shared between worlds, with peripherals secured *on-the-fly* with accesses from the secure world having always highest priority. To support this, the TrustZone Protection Controller (TZPC) virtual peripheral guarantees that when a peripheral is set as secure it is exclusively accessible from the secure world. The TZPC is attached to the AXI-to-APB bridge and can be configured to control access to APB peripherals. This enables secure interactions between users and programs running in the secure world (secure tasks), and between secure tasks and the peripherals of their choice. Since TrustZone-enabled processors boot always in secure mode, secure code executes always before the non-secure world bootloader (e.g., u-boot) is even loaded in memory. This allows to define a security perimeter formed by code, memory and peripherals from the moment the device is powered on. This security perimeter defines TrustZone's trusted area, as depicted in Figure 2.5.

Today, TrustZone is marketed as a technology for payment protection technology and digital

rights management (DRM)[15]. Global Platform's APIs[16] [66, 67, 68, 70, 71, 69, 72, 65], and their definition of a TEE follow this paradigm. Even the most recent commercial TrustZone software stack - Nvidia TLK[17], and Linaro OP-TEE[18] are founded on the principle that with TrustZone, security-sensitive tasks are offloaded to a short duty cycled secure world. Put differently, most TrustZone TEEs are designed for the case where secure world is called from non-secure world in order to handle short-lived tasks.

In some contexts, TrustZone has been used as a thin, affordable hypervisor [118, 228] to support specialized virtualization. A concrete example present in automotive & aerospace environments is using the secure world to host a Real-Time operating system (RTOS) that executes in parallel with the main OS. Here, not only does the secure run largely independent from the normal world, but it necessarily has to execute under high duty cycle in order to meet the real-time constrains demanded by the RTOS. TOPPERS SafeG [249] is an example of a framework targeting this use case. However, TrustZone's support in ARMv7 processors cannot be used for executing a general purpose hypervisor since it does not count on hardware virtualization extensions. More specifically, TrustZone does not support trap-and-emulate primitives. There is thus no means to trap operations executed in the non-secure world to the secure world. This means that non-secure software running at its highest priority level can access any peripheral and memory assigned to the non-secure world, making it impossible to guarantee isolate if multiple virtual machines (VMs) are running in the non-secure world. For the same reason, virtualization within the secure world is not possible either. Dall et al. have studied this in depth while designing and developing KVM/ARM [81] for the Linux Kernel.

Using concrete processors as examples this means that a Cortex-9 processor can leverage a secure world since it counts on the TrustZone security extensions, but not a general purpose hypervisor since it does not support hardware extensions. A Cortex-15 can support both since both sets of extensions are incorporated in the processor. ARMv8 processors such as the Cortex-A53 and -A57 also support both sets of extensions. At the time of this writing all publicly available information describing the ARMv8 architecture points to TrustZone and virtualization extensions being maintained separated.

While TrustZone was introduced more than 10 years ago [6], it is only recently that hardware manufacturers such as Xilinx, Nvidia, or Freescale, and software solutions such as Open Virtualization[19], TOPPERS SafeG[20], Genode[21], Linaro OP-TEE, T6[22], or Nvidia TLK have respectively proposed hardware platforms and programming frameworks that makes it possible for the research community [131], as well as industry to experiment and develop innovative solutions with TrustZone. This turn towards a more open TrustZone technology is

[15] `http://www.arm.com/products/processors/technologies/trustzone/index.php`

[16] `http://www.globalplatform.org/`

[17] `http://www.w3.org/2012/webcrypto/webcrypto-next-workshop/papers/webcrypto2014_submission_25.pdf` We provide TLK's presentation slides since there is still no publicly available white paper or article describing its architecture.

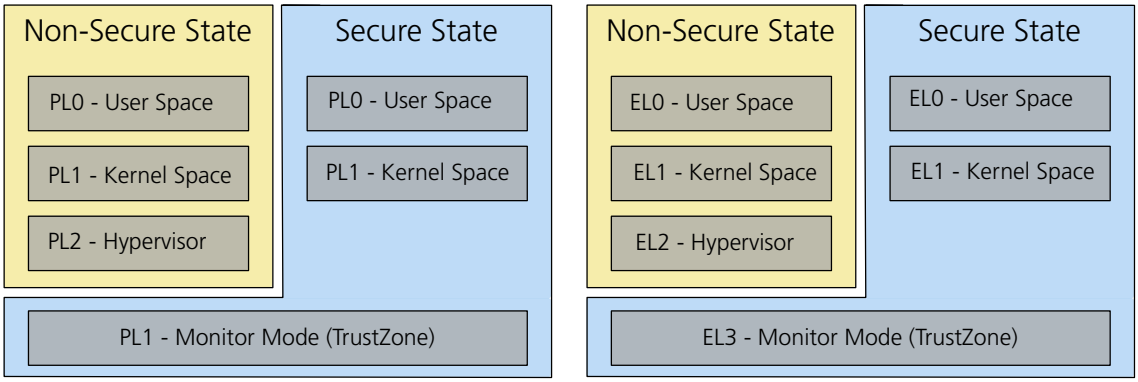
[18] `https://github.com/OP-TEE/optee_os`

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

[20] `http://www.toppers.jp/en/safeg.html`

[21] `http://genode.org/documentation/articles/trustzone`

[22] `http://www.trustkernel.org/en/`



(a) ARMv7 Processor Modes **(b)** ARMv8 Processor Modes

Figure 2.6: Difference between processor modes in ARMv7 (Privilege Levels) and ARMv8 (Exception Levels)

in part thanks to the work done by Johannes Winter and the Graz University of Technology [290, 293].

2.5 Other Techniques

Apart from the presented secure hardware, there is a number of techniques that also can provide a TEE. We cover them in this section, also in an increasing order of integration of (equivalent to) the trusted and untrusted areas.

2.5.1 Hardware Replication

One way of providing a trusted area is by completely replicating the hardware that forms the untrusted area and placing a trusted software stack on it. A good way to think about this approach is through a smart phone with two screens, two RAM chips, to SD Cards, etc. This abstraction can apply to any device (e.g., set-top box, server). Architecturally, this approach is very similar to that of a smart card or a TPM; the trusted area is completely separated from the untrusted area. Indeed, not hardware components are shared. The level of logical integration between the two areas depends then on the communication interface that the two pieces of hardware define.

While this is a good abstraction to reflect upon when discussing different techniques for secure hardware (What does these secure hardware extensions offer me, that two pieces of the same hardware without them would not?[23]), the level of extra tamper-resistance that this approach offers is minimal. While isolation can be guaranteed (specially if there is no logical communication between the two areas), the tamper resistance of the base hardware is not improved. Moreover, since all components of a device must be replicated (i.e., CPU,

[23] We thank Luc Bouganim for asking this question in the first place.

main memory, secondary storage, peripherals), the software stack in such trusted area would be as complex as the one in the untrusted area. Note that, even if the trusted software stack is minimal and formally verified, all drivers to manage the trusted area are necessary. Still, this is a valid approach to provide an isolated trusted area.

2.5.2 Hardware Virtualization

A powerful way to provide isolation without the need of hardware replication is virtualization. Virtualization supported by hardware extensions can provide full isolation by means of trap-and-emulate instructions. This would allow to create two virtualized areas representing the trusted and the untrusted one. The main benefit of this approach is that it is hardware-agnostic; more and more architectures support hardware virtualization, thus the trusted and untrusted areas could be directly ported to different hypervisors. Moreover, since the hypervisor executes in higher privilege mode and manages its own address space, hardware separation is guaranteed. A number of open source and commercial projects have indeed attempted to secure commodity applications using full-system virtualization. Examples include Qubes OS [241], Bromium [57], and XenClient [63].

Many compare hardware virtualization with TrustZone, since they consider TrustZone as set of limited virtualization hardware extensions. However, while TrustZone extends to the system bus – thus enabling to secure peripherals – virtualization remains at the same level than other secure hardware, whose trusted area stops in main memory. In this way, virtualization also offers a sharp definition of the trusted and untrusted areas, relying all integration to software communication mediated by the hypervisor.

2.5.3 Only Software

Finally, it is also possible to provide a TEE only in software. While this approach does not comply with the definition that we have given for a TEE since there is not hardware separation, we include it here for completeness. In fact, many security approaches have relied in software-only solutions to the computation and storage of sensitive data, special operations, and peripheral manipulation. In this case, integration between the trusted and untrusted areas is maximal; they are the same. However, under the assumption that the operating system can be compromised, any application executing in the same address space as the OS could also be compromised.

2.6 Discussion

In this Chapter we have presented the state of the art in secure hardware, and we have used it to define different types of Trusted Execution Environments. This comes as a product of the hardware itself, which defines the properties for the trusted area, as well as the interfaces through which it communicates with the untrusted area. Looking at the different TEEs and analyzing the design space that secure hardware manufactures confront, we observe

that the characteristics of a TEE come as a trade-off between the **isolation** of the trusted area, and the level of **tamper-resistance** of the hardware leveraging that trusted area (and therefore protecting the TEE). By isolation we refer to the level in which the trusted and untrusted areas integrate. This is, the type of computation carried out in the trusted area, the complexity of the code, the interfaces with the untrusted area, and the extensibility of the security perimeter around the trusted area. In this way, we define the opposite if isolation as **scope**; a technology that allows for a high integration of the trusted and the untrusted area features low isolation and high scope. By tamper-resistance we refer to the level to which the trusted area can resist logical and physical attacks. In this way, a TEE that is very isolated exhibits a higher level of tamper resistance, while a TEE that interacts with untrusted components (less isolated) necessarily exhibits a lower level of tamper resistance.

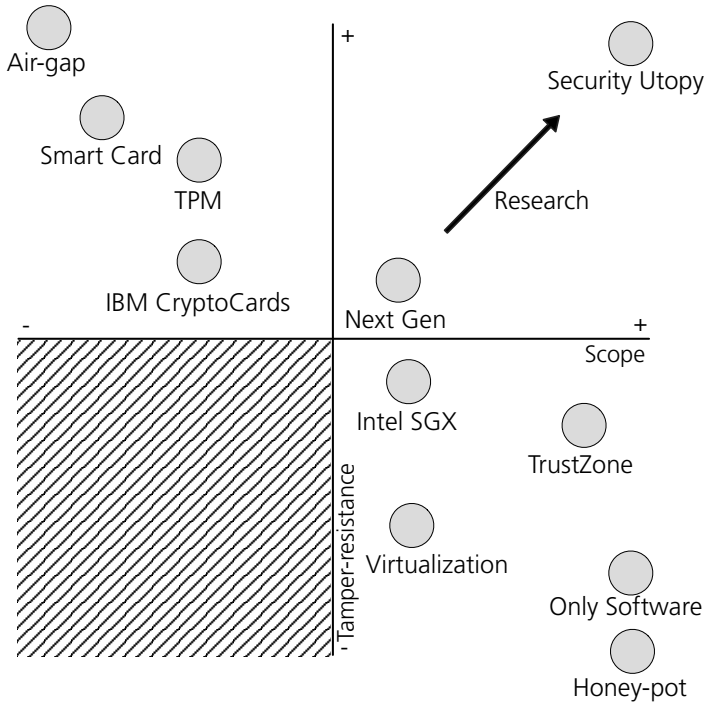


Figure 2.7: TEE Design Space. We define different TEEs as a function of the scope (1/isolation) and level tamper-resistance provided by the secure hardware leveraging them. We add an *air-gap*, a *honey-pot*, and a *security utopy* device to mark the extremes in TEE design space.

We represent this trade-off in Figure 2.7, where we position the technologies we have described throughout this chapter. We use level of tamper-resistance and scope as metrics. We also use three conceptual devices to represent the three extremes in the TEE design space: (i) an *air-gap*, which is a conceptual device that is physically isolated (0 scope) and never connected to any network, thus exhibits the highest level of tamper-resistance (the *air-gap* device is indeed *tamper-proof*); (ii) a *honey-pot*, which is represents the opposite extreme, where the trusted area extends to every single component of the system, but the level of tamper resistance is non existent; and (iii) a *security utopy*, which represents the utopy for secure hardware, where security extends to all components on the system (everything is

trusted area), but at the same time these components are *tamper-proof*. For completeness, we include technologies that we have not explicitly covered in this chapter since the properties they exhibit do not add value to the discussion; their properties have already been covered by other secure hardware. Note that we do not consider a piece of secure hardware that, being isolated, does not exhibit a positive level of tamper-resistance. Such piece of hardware is by definition not secure.

While today, hardware companies are developing the next generation of secure hardware, the techniques that they are using are either based on (or extend) the ones presented in this chapter. We expect to see more tamper-resistant components, specially in secondary storage; faster secure co-processors dedicated to cryptographic operations, but also for general purpose manipulation of sensitive data; and a more flexible security perimeter that (i) extends to more components (e.g., all levels in the cache hierarchy, cross-privilege CPU modes, DMA), and (ii) becomes a run-time property of the system (i.e., a programmable trusted area). Still, in this thesis we limit the scope of the security perimeter defined by the trusted area to the secure hardware that is commercially available at the time of this writing. We consider then that the largest incarnation of a trusted area extends to CPU, system bus, memory, and other peripherals[24].

Our prediction is that research in secure hardware and TEEs will be about bridging the gap between isolation and tamper-resistance. In this way, as peripherals become more complex, we can expect secure hardware being present in all the components of of system, pushing down security responsibilities to dedicated devices. Examples include I/O devices such as network cards and storage devices incorporating cryptoprocessors to provide default encryption of I/O operations[25], main memory and the entire cache hierarchy being partitioned in hardware to support secure high-computation while avoiding the overhead of maintaining a encrypted page table [204], or DMA differentiating I/Os and mapping them to peripherals assigned to the trusted and untrusted areas.

[24] We consider memory as a special peripheral.

[25] Indeed, solid state drives (SSDs) already incorporate dedicated cryptoprocessors to randomize data pages and distribute the usage of flash chips, therefore maximizing the life of the SSD. We expect to see these co-processors being used for security-specific tasks.

Chapter 3

Run-Time Security

Since Schroeder's seminal work on Multics [251] back in 1975, there has been much work on run-time security. A good review of existing work can be found at [112]. Both the industry and the academy have tried to solve security issues that emerge from a running environment: data corruption, secret leakages, or privilege misuse. In this Chapter we will cover the state of the art in run-time security from three different angles. First, we look at the theoretical work on run-time security, where we focus on access and usage control models. We focus on the latter since usage control models are powerful enough to define complex usage policies involving hardware and software components. Second, we look at run-time security from the user data perspective. Here, we look at the work done in data protection, where we cover trusted storage, secure storage, and other protection techniques for user data. Third, we look at work on reference monitors, where we cover different approaches to guarantee the the integrity of the execution environment in order to leverage trust to running applications.

3.1 Access and Usage Control

Still today, many associate run-time security with access control. In general, access control mechanisms ensure that a set of objects (e.g., data, operations, peripherals) are mediated by a set of access policies. When a subject satisfies these policies, access to an object is granted. Note that in this case, the subject is able to use the object at will after access is granted; object protection depends then on the granularity and frequency of access control points.

Access control has been materialized in many different ways [248]. Examples include Mandatory Access Control (MAC), Discretionary Access Control (DAC), and Role-Based Access Control (RBAC) [114]. Indeed, different implementations of these access control versions have been customized for different levels of the software stack: Operating systems [216, 263, 140], hypervisors [242, 64], or secure hardware [199, 247]. However, as for today, access control models can be divided in two main categories: Access Control Lists (ACLs) and Capabilities (Figure 3.1).

- **ACLs.** In an access control list model, the authorities are bound to the objects being

secured. Put differently, a subject's access to an object depends on whether its identity appears on a list associated with the object. Most Unix-based operating systems (e.g., Linux, BSD) support POSIX.1e ACLs [155].

- **Capabilities.** In a capabilities model, the authorities are bound to objects seeking access. Put differently, subjects and objects are bound through a reference (or *capability*) that enables a subject to access the object. Linux's capabilities are slightly different, and follow what is known as *"POSIX Capabilities"*, which are defined in POSIX draft 1003.1e [154]. Here capabilities are a partitioning of root privileges into a set of distinct privileges [173].

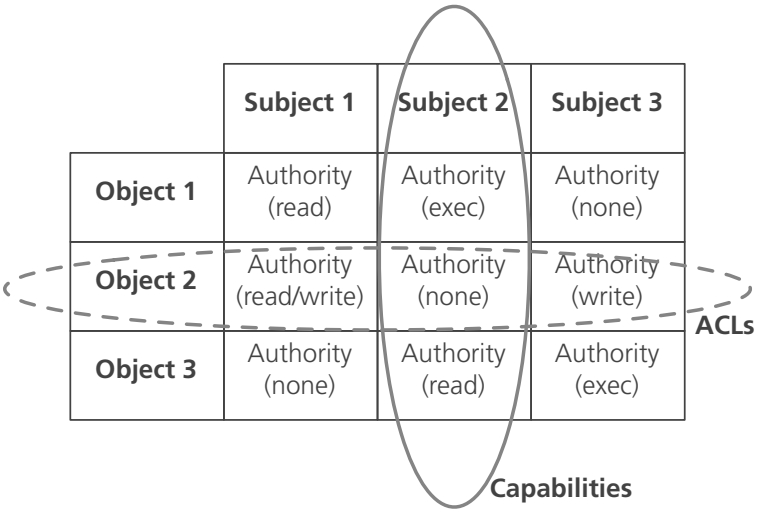


Figure 3.1: Access Control: Capabilities vs. ACLs[1]. Note that here we represent traditional capabilities, not *"POSIX Capabilities"*.

In any case, access control has the limitation that the granularity and frequency of the access control points is normally defined in terms of actions. For example, if a subject (e.g., a process) is given access to an object (e.g., file) in terms of a specific action (e.g., read), then the process can read any part of the file and copy it into its memory space. What the process does with the read information is out of the scope of the access control policies governing the file. What is more, such model cannot extend to different objects, even when they belong to the same type. In the example above, a process could read from different files to aggregate information from different sources (and keep them in main memory). Access policies cannot establish that a process can access files A and B, but not C, if A and B have been previously accessed.

Usage control is meant to address this deficiency. This is, allow to define (and enforce) policies based on continuous actions, i.e., usage. In a way, usage control is applying access control at higher granularity and frequency, and more importantly at a larger scope, so that previous accesses are taken into account when granting access to a new object. Usage control

[1]This representation of Capabilities vs. ACLs is inspired in `http://www.skyhunter.com/marcs/capabilityIntro/capacl.html`

models usually refer to UCON$_{ABC}$ [221]. Here, Park and Sandhu define the model depicted in Figure 3.2. The actions that a subject executes against an object are mediated by rights. Rights are divided in three categories:

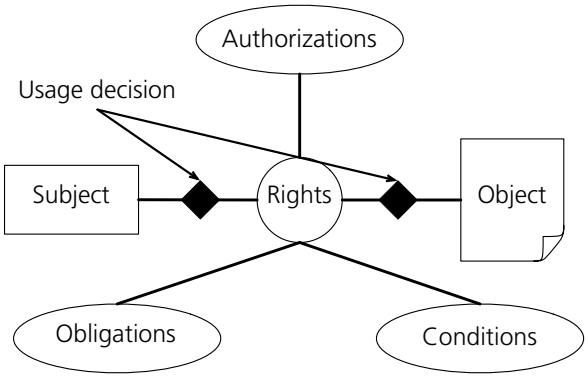


Figure 3.2: UCON$_{ABC}$ Usage Control Model

- **Authorizations.** Authorizations are functional predicates that have to be evaluated for usage decision and return whether the subject is allowed to perform the requested rights on the object. Authorizations evaluate subject attributes, object attributes, and requested rights together with a set of authorization rules for usage decision
- **Obligations.** Obligations are functional predicates that verify mandatory requirements a subject has to perform before or during a usage exercise. This is, actions a subject must take before or while it holds a right.
- **Conditions.** Conditions are environmental or system-oriented decision factors.

Apart from these tree types of rights, UCON$_{ABC}$ defines the concept of mutability, which denotes decisions based on previous usage[2]. Besides the model in itself, UCON$_{ABC}$ defines an algebra that enables to formalize usage policies. In this way, UCON$_{ABC}$ formally encompasses access control, since MAC, DAC, ACLs, capabilities, etc. can be expressed with UCON$_{ABC}$'s algebra. Actual examples can be found in [221].

UCON$_{ABC}$ is powerful enough to define complex usage policies that go beyond independent objects. Indeed, it can describe complete information flow frameworks born in the context of social sciences such as Helen Nissenbaum's Contextual Integrity [215]. However, there is, to the best of our knowledge, no actual implementation that complies with the security and trust assumptions in which UCON$_{ABC}$ is based. Park and Sandhu explicitly express their concerns with regards to an architecture that can support UCON$_{ABC}$; but for them *[...] keeping the model (UCON$_{ABC}$) separate from implementation is critical to separating concerns and reaching fundamental understanding of disparate issues [246]. [...].* It is important to understand that the motivation behind UCON$_{ABC}$ is to improve access control, which at

[2]In the example we used above, this would represent denying a process access to file C, if files A and B had been accessed before. Mutability is what enables usage control to have a larger scope when enforcing usage policies.

that moment was (and still today is) the main model enabling security in computer science. This means that the assumption is that there is something to protect against; if the model is not protected then security is by no means enhanced. Since UCON$_{ABC}$ is broader in scope than access control (i.e., it contemplates several access control policies at higher frequency involving more objects) the attack model is also broader. Implementations of UCON$_{ABC}$ that are directly implemented in the same environment as user applications, or even int the operating system fail to comply with the conception of a broader protection scope.

3.2 Data Protection

We now focus on the state of the art in data protection. With data protection, we refer to the work done in trusted storage, secure storage, and other protection techniques for user data. Since there is no widely accepted term for denoting data protection mechanisms, we prefer to refer to them in general term when describing the state of the art. Some researchers might prefer their work to be considered under specific terms (e.g., secure storage), and we want to respect that. When we present our data protection proposal in Section 8.2.1 we will use the term *Trusted Storage* to refer to it, since we believe that the adjective *trusted* suits better the actual decision making behind the use of TEEs; remember that the trusted area only provides isolation, not correction.

Probably the most relevant work in data protection in the systems community is VPFS [285]. Here, Weinhold et al. propose an encryption wrapper that, reusing the paravirtualized untrusted I/O stack of L4Linux[3], can provide data confidentiality, integrity, and recoverability to trusted applications. They do this in order to build a data protection mechanism with untrusted components while maintaining a low TCB. Their solution is to store file system metadata in the trusted area. The assumption is that their virtual private file system is exclusively used by trusted applications. While untrusted applications share a physical disk with trusted applications, they cannot make use of the trusted storage capabilities. Any new trusted application (e.g., a DBMS) increases the size of the TCB, thus increasing the attack surface. Moreover, the fact that new applications need to be tailored to execute inside of L4Linux limits the scope of innovative services. In this line, but in the context of networked file systems (NFS) approaches like SUNDR [186], SiRiUs [126], and Plutus [169] define the state of the art in securing block and file-level storage on an untrusted environment.

Indeed, building data protection mechanisms reusing untrusted components is not a novel idea. In the database community, the Trusted Database System (TDB) [197] was one of its precursors. Here, Maheshwari et al. propose to leverage a small amount of trusted storage to protect a scalable amount of untrusted storage in the context of a database system. They define their trusted storage as a different disk from the one where the untrusted database system executes. While they understood the necessity of counting on a trusted area, separated by hardware from the database system they intend to protect, they fail to acknowledge that the communication interface between the two disks represents indeed a large attack surface. Still, the ideas they present apply today to cloud storage, where the trusted storage part could be maintained locally, while the untrusted storage part would be the cloud in

[3] `http://os.inf.tu-dresden.de/L4/LinuxOnL4/`

itself. The attack surface, would in this case be larger since their approach would be exposed to remote attacks over the network. Another interesting approach proposed simultaneously (in fact, in the same conference) is [119], where Fu et al. propose a read-only file system that can be used as a root of trust in untrusted servers, which would again respond to cloud providers today.

Another relevant work is done by Overshadow [62]. Here, Chen et al. propose a method to provide confidentiality and integrity of an application address space by protecting data pages in a virtualized environment. The use *cloacking* to provide cleartext pages to *cloacked* applications, and encrypted pages to the kernel. This allows to reuse kernel resource arbitration, as well as the page cache. While they assume the commodity OS untrusted, they rely on a full-featured hypervisor to provide memory isolation (VMware VMM using binary translation for guest kernel code [3]) on top of x86. In their evaluation they mention that they add 6400 LOC to the VMM original code base, and 13100 LOC to support page protection in a component they refer to as *shim*. While they can guarantee protection of sensitive data, they cannot guarantee availability or durability since they rely on system calls in the untrusted kernel. In fact, reusing system calls makes Overshadow vulnerable against side-channel [178, 168] and the more recently introduced Iago [60] attacks. Still, the value of Overshadow to the systems and security communities is immense given that their work is a proof that virtualization can be used to force memory isolation even in an untrusted kernel.

In the operating system community we find approaches like Proxos [267], where Ta-Min et al. propose redirecting system calls generated in the (equivalent to the) untrusted area to the trusted one, where a private file system is implemented. A policy engine that generates a usage decision could eventually be implemented here. While we believe that making decisions based on system calls is a very powerful technique, assuming that they can be redirected from an untrusted area is a leap of faith. Since the untrusted area can be compromised, an attacker could inject kernel code via a loadable kernel module (LKM), or access kernel memory via */etc/kmem* to replace the system call, prevent that it reaches the trusted area, or modify its parameters. Other approaches such as Trusted Database System (TDB) [197], or DroidVault [188] suffer from the same attack vectors, all relying on untrusted modules that act as a bridge between the trusted and untrusted areas. If this module is compromised, all communication with the trusted area is compromised[4].

If we focus exclusively on file systems providing encryption, we find different alternatives implemented at all levels of the software stack. In one end of the design space we have examples such as EncFS [136], a user space cryptographic file system available for a variety of commodity OSs. From a security perspective EncFS is grounded on the assumption that an attacker cannot gain *root* access to the system. In this case, an attacker could impersonate any user to access a mounted EncFS partition, installing a rootkit to steal user passwords and mount encrypted partitions, or injecting kernel code to steal data while it resides decrypted in memory. Another example of a user space file system is Aerie [281], which defines a virtual file system interface to storage class memories for Linux systems. While Aerie does not approach directly encryption, it refers to a trusted file system service (TFS) in which it delegates in order to implement protection capabilities. If Aerie were to

[4]Note that this bridge has nothing to do with TrustZone's secure monitor, which runs in secure privilege mode in order to load TrustZone's secure stack.

implement cryptographic operations, it would be here they would be implemented. What we find interesting about Aerie is that it moves most of the file system logic to user space, only relying on the kernel to provide allocation, protection, and addressing through *libfs* [74]. More specifically, we are interested in its attack model. While it is argued that hardware protection guarantees memory partitioning among applications, from a security perspective Aerie is founded on the assumption that the kernel, and at least the *libfs* client library are trusted. Here, the encryption scheme would be compromised if considering an attack model where the attacker gains *root* access to the machine[5]. In general, user space security fail when presented to this attack model.

Moving down in the software stack we find file systems such as eCryptfs [139], which implements encryption at a page level; and NCryptfs [296], which takes a similar approach in the form of a stackable file system. While attacks become more complicated, an adversary with *root* access could inject kernel code to bypass the encryption mechanisms, use internal kernel operations in the I/O stack to decrypt random pages, or simply read kernel and physical memory to steal sensitive data in motion. Note that pages are decrypted while in the page cache[6]. Using other file systems supporting encryption such as CephFS [99] have the same issues. In general a compromised kernel cannot be trusted to perform encryption, or even delegate this task to a trusted area.

At the end of the software stack we find encryption implemented directly in hardware, at the time that physical blocks are being written. An example of this paradigm are flash memories such as Solid State Drives (SSDs), which normally make use of in-build cryptoprocessors to ensure that data is better distributed throughout flash memory. Using this devices for protecting data would reduce latency and overhead of performing cryptographic operations with respect to CPU-based encryption. However, there is to the best of our knowledge, no published work on hardware-assisted data protection similar to the ones presented in the software stack.

Finally, existing commercially available storage security solutions such as NetApp's SafeNet[7] require dedicated hardware. Also, McAfee's DeepSafe[8], which relies on TPM, is restricted to a specific hardware (Intel-based processor equipped with TPM) and software (Windows) combination. The same applies for BitLocker [78]. This form of storage security is effective, but rigid and it restricts users' ability to configure their computers [95]. We cannot analyze with certainty the attack model that would apply to these commercial solutions since we cannot freely access the code. However, based on our experience with the TPM driver in the Linux Kernel, all communication is still mediated by a component in the untrusted kernel, therefore being susceptible to a malicious root.

[5] TFS runs as any other process.

[6] To be more precise, EcryptFS uses its own page cache, but the pages are still decrypted. NCryptfs uses the kernel page cache, where pages are in cleartext

[7] `http://www.netapp.com/us/products/storage-security-systems/storagesecure-encryption/`

[8] `http://www.mcafee.com/us/solutions/mcafee-deepsafe.aspx`

3.3 Reference Monitors

We dedicate the last part of the state of the art in run-time security to reference monitors. In computer science, reference monitors are defined as components that mediate the access to sensitive assets based on a given policy. Traditionally, they have been used in the context of operating systems to protect kernel objects (e.g.,memory, files, network) from illegitimate access [12, 166]. Indeed, reference monitors been mainly used for access control, but they can be extended for other purposes [294].

We identify three classes of reference monitors that are interesting for our work: those used to enforce a given policy, those use for intrusion detection, and finally those that aim at protecting the integrity of OS and applications. We explore them separately.

3.3.1 Policy Enforcement

We start by presenting Multics [251], where back in 1975 Schroeder already discussed that usage enforcement was only possible if the execution environment contained a trusted area that could control the rest of the system. Schroeder was referring at that point to the Multics kernel. The argument was that the kernel could be trusted if it was small enough so that an expert could manually audit it.

More recently, Ligatti et al. [190] propose mandatory results automata (MRAs) as a way to enforce run-time usage policies. In their vision, a security model sits between an untrusted application and the executing environment, altering their communication in order to meet a set of policies. Other work has targeted specific run-time environments such as Java JRE [4] and CLR (.NET) [88, 4], or the Android platform in a similar fashion. In Android specifically, there has been remarkable contributions with TaintDroid [108], the Saint Framework [218], and RV-Droid [113]. While using different techniques TaintDroid, Saint, RV-Droid, and many others [35, 32, 148] carry out a dynamic analysis of the run-time behavior of untrusted applications, generally relying on Android sandboxing to provide isolation.

While all these approaches propose interesting frameworks to enforce usage policies - some of them with a very strong formal verification behind them -, they all lack a realistic attack model that responds to today's run-time attacks. By placing their policy engine in the same untrusted environment as the applications they intend to monitor, they all make the underlying assumption that the commodity OS (kernel) is trusted. In an attack model where the attacker gains *root* privileges, the policy engine could be directly tampered with, bypassed, or its output modified.

3.3.2 Intrusion Detection

In [226], Ganger et al. proposed host-based intrusion detection systems (HIDS). The goal was to detect intrusions in a system by observing sequences of system calls [144], patterns [210] or storage activity [226]. The main issue for host intrusion detection is that a smart attacker could feed the HIDS component with false information about the system and therefore bridge

its security. Approaches such as [226] where the storage activity is monitored by looking at system primitives allow to decouple the OS processes from the processes running on the storage device. Illegitimate accesses can be detected even when the OS is compromised. This approach assumes however that the storage system is separated from the OS and cannot be compromised.

In [297] Wurster et al. take this approach further, monitoring the file system while storage and OS run in the same physical device. While they present an interesting modification of the Linux's VFS to mark monitored files, the integrity mechanisms run in a demon in user space. Thus, they assume that the the user of the system is trusted not to be malicious. All these assumptions are a leap of faith; attackers can use widely spread rootkits to obtain root control of a system. If devices do not offer a hardware supported security perimeter, these attacks can be leveraged remotely without the user's knowledge, or published on the Internet as device class attacks.

3.3.3 Code Integrity Protection

Another specific use for reference monitors is providing code integrity. This has been an active area of research in the context of hypervisors and virtualization, mainly because the hypervisor executes in its own address space, which allows to define a trusted area where the reference monitor can be implemented. In fact, most architectures today enable CPU privilege modes specific for the hypervisor, thus providing hardware-assisted isolation. We investigate the state of the art in code integrity with the intention of understanding how main memory can be protected at run-time. We focus on those approaches that could support the enforcement of usage policies. As we will see in Chapter 8, this will be a fundamental part to guarantee the protection of sensitive data at run-time.

Since the early days of virtualization, using the hypervisor to isolate different execution environments that run simultaneously in the same hardware has been a popular topic of research. How isolation relates to security was first studied by Madrick & Donovan [196], Rushby [239], and Kelem & Feiertag [172]. More contemporary work such as Terra [121] provided a proof that virtualization could successfully be used to make several OSs with different security requirement coexist in the same environment. Isolation is now a topic research in itself. For us, this line of work is interesting in order to understand how software is used to achieve isolation. This is an extreme when designing the trusted and untrusted areas, similar to that of secure hardware that define a completely separated trusted environment, with the difference that virtualization cannot offer the same guarantees as hardware separation.

Moving away from isolation, towards monitoring, in Lares [223], Payne et al. present an approach for providing run-time monitoring of a commodity OS from the hypervisor. They place hooks in the commodity OS' kernel before a sensitive asset is used, and execute a monitoring routine in the hypervisor address space to mediate the access to the assets that are considered sensitive. While their contribution is significant both in terms of the mechanisms implemented within Xen [33] and the low performance impact of their implementation, they in the end make the assumption that the hooks that they insert in the commodity OS (untrusted area) are immutable. An attack model where the kernel is untrusted cannot hold

this assumption.

In SecVisor [255], Seshadri et al. propose a small hypervisor to verify the integrity of a host kernel at run-time. They make use of AMD SVM [7] virtualization extensions to allow SecVisor run at a higher privilege mode than the kernel. Using SVM, SecVisor can mediate access to the memory management unit (MMU), IOMMU, and physical memory by intercepting events coming from the OS and applications. They also use instructions similar to the ones found in Intel TXT (Section 2.3.1), to lock physical pages. Combining this techniques, they can verify the integrity of kernel code. While they introduce two hypercalls in the kernel, they maintain the attack surface to a minimal thanks to a well-defined, minimal interface. Still, they are vulnerable to attacks against the control flow (e.g., return-to-libc [213]), and direct modifications of kernel data in order to indirectly control the kernel's control flow. Also, even if these attacks were addressed, user data would still be exposed if an attacker gains root privileges and access kernel and physical memory in a conventional way (*/etc/kmem* and *etc/mem* respectively). While in [117] Franklin & Chaki expose these limitations, together with some implementation bugs, and provide a formal verification for SecVisor, it is not clear to us how attacks against kernel data are addressed. User data protection is not addressed either in this revision. A very similar approach is proposed in TrustVisor [200], where AMD SVM and the same techniques are used to implement a reference monitor to enforce MMU protection, in this case for both kernel and applications. While here, McCune et al. succeed to create a safe memory region that they can attest, they fail to provide the mechanisms that can verify the correct operation of an application (or the OS) based on the resources they use. Put differently, they have the framework to implement usage policies that govern I/O operations, but they do not use it for this purpose.

In HyperSafe [282], Wang et al. look at the problem of protecting a Type 1 hypervisor[9] at run-time by implementing a CFI [1] technique. Here, they consider memory attacks such as return oriented programming (ROP) [256, 51, 149] and return-to-libc. The cornerstone for their hypervisor integrity protection is a technique to protect memory pages that they denote *non-bypassable memory lockdown*. Here, they use (i) x86 memory protection mechanisms, more specifically paging-based memory protection (*paging* in x86 terminology) in combination with (ii) the Write Protect (WP) in the control register CR0 [164], which controls how the supervisor code interacts with the write protection bits in page tables: if WP is turned off, supervisor code has access to any virtual memory page, independently of the protection bits present in the page. Enabling write-protection in the page table and turning on WP, no code can modify the page table. By atomically escorting all page table modifications and verifying that no hypervisor code or data, nor double mapping is introduced, they guarantee that the hypervisors code-flow integrity (CFI) is protected. Inside of the atomic operation the propose they also propose use of policies to verify the page modification bu they do not implement it. All in all, HyperSafe succeeds to provide memory protection in the x86 architecture, being its only limitation the fact that they implement CFI, which has known conceptual vulnerabilities [82, 127].

The most comprehensive approach that we have found in this area is Code-Pointer Integrity (CPI) [182]. Here, Kuznetsov et al. propose a *safe region* in which they store pointer

[9]Type 1 hypervisors are native, bare-metal hypervisors (e.g., Xen, VMWare ESX [111], BitVisor [259]). Type 2 hypervisors require a hosted OS kernel (e.g., VMWare Workstation, VirtualBox)

metadata. They use this metadata to track memory operations and guarantee the integrity of code pointers in a program (e.g., function pointers, saved return addresses). Their work means a turning point in memory integrity verification, since it addresses memory attacks that until the moment surpassed the state of the art in memory protection (e.g., DEP, ASLR [271], CFI [1]). Examples of these attacks include return-to-libc [213, 256], return oriented programming (ROP) [256, 51, 149], side channel attacks [150], just-in-time code reuse attacks [264], and other buffer overflow techniques affecting specially the C language. CPI addresses these issues by guaranteeing the integrity of a concrete memory region. While they do not implement such *safe region* in their prototype, they express their intentions to use Intel MPX [160], as a set of hardware memory protection extensions that could support the *safe region*.

3.4 Conclusion

In this chapter we have covered the state of the art in run-time security. We started presenting access and usage control, and focused on UCON$_{ABC}$, formal usage control model that is the reference in the field. We then explore two concrete areas of research that relate to run-time security: data protection mechanisms (e.g., trusted storage), and reference monitors.

Before we present a discussion on how these techniques relate to the trusted and untrusted areas presented in Chapter 2, let us introduce boot protection mechanisms in next chapter. We take this discussion in Chapter 5.

Chapter 4

Boot Protection

Until this moment we have covered the state of the art with regards to the hardware, software frameworks, and formal models that directly relate to run-time security mechanisms. Indeed, the main focus of this thesis is in providing operating system support for run-time security. However, in order to guarantee that the components giving that support have not been compromise at boot-time, we need to understand how to protect the boot process so that we can guarantee a chain of trust. In this Chapter we (i) present the different approaches that have been explored to protect the boot process; (ii) we revise some of the terminology that overtime has been either misused, lost, or overseen; and (iii) point to the most relevant related work in the field.

4.1 Boot Protection Mechanisms

The first boot protection mechanism was proposed by Arbaugh et al. in [17]. They referred to it as a secure boot process and named it AEGIS. They describe a way to verify the integrity of a system by constructing a chain of integrity checks. Every stage in the boot process has to verify the integrity of the next stage. In formal terms, if we sequentially number the stages that take place in the boot process $i = 0, 1, 2, ...$, then the integrity chain can be expressed as:

$$\begin{aligned} I_0 &= true, \\ I_{i+1} &= \{I_i \wedge V_i(S_{i+1}) \quad \text{for} \quad 0 < i < sizeof(S) \end{aligned}$$

where V_i is the verification function that checks the integrity of the next stage. This is done by using a cryptographic hash function and comparing the result with a previously stored signature value. The result of the verification function is then combined with I_i, a boolean value indicating the validity of the current stage, using the conjunction operator $\wedge$.

Since Arbaugh's work a number of different approaches have been proposed. Outside of security specialized circles, protecting the boot sequence is usually referred to indistinctly

as *secure boot* or *trusted boot*. Other terms such as *verified boot*, *authenticated boot*, *certified boot*, or *measured boot* are also found in commercial products and some research articles, all with different connotations. As a result, different, contradictory definitions have been used over time. We will now define the terms according their acceptance, and relevance of the work referring to them. We will also present common discrepancies and overlaps both in the academia and the industry. The goal is to define the terms that we will refer to in Chapter 9 when we introduce *Certainty Boot*.

- **Secure Boot** is described in the UEFI specifications since version 2.2 [277]. UEFI secure boot verifies the integrity of each stage of the boot process by computing a hash and comparing the result with a cryptographic signature. Secure boot begins with the execution of immutable code from a fixed, read-only memory location. A key database of trustworthy public keys needs to be accessible during boot time so that the signature can be verified. In secure boot, if any integrity check fails, the boot will be aborted. If the boot process succeeds the system is expected to be running in a *trusted state*. However, secure boot does not store evidence of the integrity checks. Thus, attestation of the device at run-time is not possible. In its original form not even the user can verify that the integrity checks have taken place[1]. This definition of secure boot is widely accepted in the security community [201, 243, 91, 277, 18, 222]. Additional information on UEFI secure boot can be found here [217].

- **Trusted Boot** is defined by the Trusted Computing Group (TCG) as part of the same set of standards [274] where they define the TPM. In this case, the TCG target target boot protection with trusted boot. The TCG describes a process to take integrity measurements of a system and store the result in a TPM (Section 2.2) so that they cannot be compromised by a malicious host system. In trusted boot, integrity measurements are stored for run-time attestation, but the boot process is not aborted in case of an integrity check failure. In other words, integrity checks are not enforced. At boot-time the hash of each following stage in the boot sequence is sent to the TPM where it is appended to the previous hash. This creates a hash chain, called Platform Configuration Register (PCR), that can be used for various purposes. For example, it can be used to decrypt data only when the machine reached a specific stage in the boot sequence (*sealing*) or to verify that the system is in a state that is trusted (*Remote Attestation*). Even when in the TCG specification, a TPM - or any other tamper-resistant co-processor - is not required, it is normally accepted that such co-processor is used. In mobile devices, where TPMs are not available, the TCG and others are working on materializing a TPM-like hardware component called Mobile Trusted Module (MTM) [206, 275]. As mentioned in (Section 2.2), they are not being very successful in this effort due to the lack of tamper-resistance in TrustZone, which is the most widespread secure hardware in mobile devices.

  Since this definition of trusted boot, originally introduced by Gasser et al. as part of their digital distributed system security architecture [122], is backed by the TCG,

[1]Some extensions make use of a shared secret that the user creates after a *"clean boot"* and that it only becomes available when the *"clean boot"* state matches. PCR registers (Section 2.2) (or similar) are normally used to verify the state of the booted system.

its acceptance is also widely spread. Also, most of the community has adhere to it [243, 222, 200, 171].

Popular trusted boot implementations include OSLO [171], TCG's Software Stack (TrouSerS)[2], and tboot[3].

- **Certified Boot** has been used in [201, 105][4] as a term to combine *secure boot* and *trusted boot* in such a way that integrity checks are both stored and enforced. Since the introduction of *authorizations* in TCG's TPM 2.0 [276], certified boot is also defined as secure boot with TCG authorizations, with the later referring to trusted boot.

- **Measured Boot** is a mechanism to verify that core platform software components have not been compromised. Measured boot starts after *certified boot*. Typically, measured boot is supported by a TPM's PCR registers [270]. Conceptually, it relies in the idea that each core code component is measured before it is executed. Measurements are stored in immutable locations for later attestation (as it is done with boot stages in *trusted boot*). Several works adheres to this definition when using a TPM for storing the measurements in the PCR registers [301, 224, 289, 201].

- **Verified Boot** is a term coined by Google in Chrome OS for verifying that the firmware and the filesystem that holds the operating system are in a known, untampered state. Conceptually, verified boot is very similar to trusted boot. However, instead of using hash values to perform integrity checks, they use only digital signatures. Work targeting Chrome OS have already adopted this term [42, 125, 151]. Also, giving Google products' popularity, at least two independent efforts are driving its development: one by Google's Chromium OS team[5], and another inside of U-Boot [125, 124][6]. The fact that verified boot is directly supported in U-Boot guarantees it reaching a big proportion of the available target platforms.

- **Authenticated Boot** is sometimes used to designate *trusted boot* in some work [192, 16, 224]. The logic behind this is that *trusted boot* can then be used to define the combination of *secure boot* and *authenticated boot*, in such a way that integrity checks are both stored and enforced. This is what we referred to before as *certified boot*.

Besides the academic work, a number of patents [83, 84, 252, 19] have targeted protecting the integrity of the boot process. One patent worth mentioning is [257], where Shah et al. propose a firmware verified boot.

4.2 Discussion

Approaches implementing secure boot are often criticized for not giving users the freedom to install the OS of their choice. This is specially true for mobile devices, which today are

[2] `http://trousers.sourceforge.net/grub.html`

[3] `http://tboot.sourceforge.net/`

[4] Note that Ekberg et al. did not directly address terminology in [105], but in their presentation at ACM CCS `http://www.cs.helsinki.fi/group/secures/CCS-tutorial/tutorial-slides.pdf`

[5] `http://www.chromium.org/chromium-os/chromiumos-design-docs/verified-boot`

[6] `http://www.denx.de/wiki/U-Boot`

designed to run a single Operating System (OS). Original Equipment Manufacturers (OEMs) lock their devices to a bootloader and OS that cannot be substituted without invalidating the device's warranty. This practice is supported by a wide range of service providers, such as telecommunication companies, on the grounds that untested software interacting with their systems represents a security threat[7]. In the few cases where the OEM allows users to modify the bootloader, the process is time consuming, requires a computer, and involves all user data being erased[8]. This leads to OEMs indirectly deciding on the functionalities reaching the mainstream. As a consequence, users seeking the freedom of running the software that satisfies their needs, tend to root their devices. However, bypassing the security of a device means that these users lose the capacity to certify the software running on it. This represents a risk for all parties and services interacting with such a device[189, 237].

Moreover, even though TPM and UEFI are widely spread on desktop computers, again, they still did not reach the mobile platform: the efforts to port UEFI to ARM devices - mainly driven by Linaro - have been publicly restricted to ARMv8 servers, not considering mobile or embedded platforms[9]. Also the MTM, though especially designed for mobile devices, has not been integrated into current device hardware (except for the Nokia N900 in 2009). This leads to the necessity for different solutions in current off-the-shelf mobile devices.

In Chapter 9 we will present a novel mechanism to protect the boot process that addresses these shortcomings. The main goal is to provide users with the freedom to choose the software running in their devices, while giving them the certainty that this software comes from a source they trust. We denote this mechanism *Certainty Boot*.

[7] `http://news.cnet.com/8301-17938_105-57388555-1/verizon-officially-supports-locked-bootloaders/`

[8] `http://source.android.com/devices/tech/security/`

[9] `http://www.linaro.org/blog/when-will-uefi-and-acpi-be-ready-on-arm/`

Chapter 5

Discussion

In Part I we have covered the state of the art in the three core areas that define run-time security: (i) secure hardware that can provide a TEE (Chapter 2), (ii) formal and practical approaches to protect sensitive assets managed by a device at run-time (Chapter 3), and (iii) mechanisms to provide a root of trust at boot-time (Chapter 4). If we concentrate on the run-time parts - (i) and (ii) -, which are the focus of this thesis, it is easy to observe that there is a mutual mismatch between the formal mechanisms that *could* enable run-time security in a wide range of devices, the techniques that have been the focus of research in the security and systems community, and most of the secure hardware produced by the industry. Indeed, this mismatch has prevented the implementation of complete formal models such as UCON$_{ABC}$.

As we briefly mentioned when we presented usage control, there has been remarkable advances in social sciences in terms of information flow. Here, we gave Helen Nissenbaum's work in Contextual Integrity [215] as an example. If we come back to it, Contextual Integrity (CI) defines a framework to reason about the norms that apply, in a given social context, to the flows of personal data (i.e., information norms) based on two main points:

- Exchange/sharing of personal data is at the core of any social interaction - privacy is not about *not sharing* personal data, but about controlling the flow of information.
- Any social context (work, education, health, ...) defines - more or less explicitly - a social norm, i.e., an appropriate behavior that is to be expected.

If we move CI from a social to a computer science context in such a way that (A) a given social context is represented by a running environment, (B) social norms are represented by usage policies, and (C) information flow control is represented by the enforcement of such usage policies, we are then describing a complete usage control model. In fact, a model conceived for information flow in a social context, which is at least as complex it is in a computational one. Note that we are not arguing for the direct applicability of CI to run-time security, but to the fact that a usage control model such as UCON$_{ABC}$ is able to represent it. Here, while (A) and (B) are part of the model (however complex the usage policies materializing them

are), (C) is a technological issue: the enforcement of the usage policies defined in UCON$_{ABC}$ depends (i) on the hardware and software platform, and more importantly (ii) on the attack model to which such platform is exposed. Note that, as we have already mentioned, access and usage control models only make sense under the assumption that there is something to protect against. Depending on how intricate these attacks are (i.e., attack model), the hardware and software platform should exhibit different security properties in order to be able to resist them.

If we assume that UCON$_{ABC}$ can be implemented using current programming languages, the remaining problem is then (C), i.e., enforcing the usage control model. When we mentioned above that there was a mismatch between run-time security, current security approaches, and secure hardware, it was indeed referring to the enforcement part; many have implemented usage engines, but without enforcement the implementation is either incomplete, or responds to a limited attack model. If we can define an architecture that allows to enforce usage control policies, it is then a natural step to define *Trusted Services*, which is our approach to support innovative services while protecting the sensitive assets used to provide these services. How to enforce usage policies to provide trusted services is ones of the core problems that we address in this thesis. Note that enforcing usage control policies is a necessary part to protect sensitive data (e.g., archived mails, encryption keys), secure communication with sensitive peripherals (e.g., camera, microphone), or mediate sensitive operations (e.g., log into a remote service).

Now, we are in a position to better understand the discussion at the end of Chapter 2; the scope of a trusted service is in part defined by the secure hardware that leverages the trusted and untrusted areas (the other part is the software leveraging the TEE). In one extreme of the design space we have smart cards, where the complete physical isolation of the co-processor (trusted area) permits to focus its design in reducing its size and therefore providing high tamper-resistance. This comes at the cost of having (i) reduced RAM - thus, limiting the amount computation -; and (ii) a narrow communication interface - thus, limiting the collaboration between trusted services and untrusted applications. This design can be used to implement trusted services that are very limited in scope due to the environment restrictions (i.e., not innovative services), but on the other hand, these trusted services are protected against a wide range of intricate software and hardware attacks. Usage policies are then very limited in this type of secure hardware; the low integration between the trusted and the untrusted areas gives high tamper-resistance at the cost of a low scope (Figure 2.7).

In the other extreme we have TrustZone, where the trusted area integrates across the system bus, being no separated physical co-processor. Tamper-resistant requirements can therefore not be integrated in the trusted area design. On the other hand, the trusted area can extend to CPU computation, main memory, and peripherals - thus, increasing the computational power of the trusted area, as well as allowing for high integration between trusted services and untrusted applications. In this case, a variety of usage policies that extend to the whole device can be implemented (and enforced); a high scope in trusted services is leveraged at the cost of a lower tamper-resistance (Figure 2.7). We now move to Part II, where we present our work on bridging the gap between tamper-resistance and the scope of trusted services.

Part II

Contribution

In Part II, we present our contributions. We divide these contributions in four chapters. In Chapter 6 we introduce the conceptual contribution of this thesis. Here, we present our hypothesis, discuss the design space, and finally argue for a concrete design path. In Chapter 7, we report on the design an implementation of our operating system support proposal for a TEE based on the ARM TrustZone security extensions: a generic TrustZone driver for the Linux kernel. In Chapter 8, we use this operating system support to provide *Trusted Services*. Here, we present the architecture for a *Trusted Cell*, and describe how the *Trusted Modules* we implement can provide these trusted services. We also have a discussion about the limitations of the platform we have used, and the lessons learned in the process. We expect this discussion to be useful for future iterations of our work. In Chapter 9, we present a novel boot protection mechanism that leverages the framework formed by a TEE, operating system support, and trusted services. These four chapter are the core contribution of this thesis.

A glossary is available at the end of this thesis. We invite the reader to consult it at any time. We point to the terms we are using persistently throughout the whole thesis in order to help the reader understanding our explanations. If in doubt, please consult it[1].

[1]Note that if this thesis is read in PDF format, all terms link to its entry in the Glossary (12.2.3).

Chapter 6

Split Enforcement

As we discussed at the beginning of this thesis, software is one of the main drivers of innovation. Quoting Marc Andreessen again, *"[...] world-changing innovation is entirely in the code [...]"*. However, such innovation comes at a high price: complexity. This complexity facilitates the introduction of software bugs [146, 280, 170], and makes it easier for malicious developers to hide unspecified software in commercial products [299, 142, 260]. As we asked ourselves in the beginning of this thesis, the question is then: *How do we handle software complexity, which to the date has inevitably lead to unspecified behavior, without killing innovation? Is it reasonable to trust third party applications without necessarily trusting the parts involved in its creation (e.g., independent developers, organizations, governments)? If not, can we do something else to protect our sensitive data in a digital context?*

One way to answer these questions is to provide the device with a set of primitives to (i) monitor the behavior of the software executing on it, and (ii) enforce a set of usage control policies (Section 3.1) to remain in (or come back to) a known, trusted state. In the end, these primitives define a mechanism for a device to control how system resources are being used, and actuate over that use. Allowing users to define their own usage policies helps them protecting their sensitive data. We refer to these primitives as *Run-Time Security Primitives*.

In order to support run-time security primitives, we propose a state machine abstraction that captures the utilization of system resources as state transitions. As long as these state transitions adequately match actual system resource utilization, by controlling the transitions, it is then possible to enforce how system resources are accessed and used. Since usage control can be expressed as policies [221], it is possible to define a set of policies that manage (and enforce) resource utilization. A device that supports run-time security primitives can then define its own set of *Trusted Services*. In fact, usage control is a necessary component to support trusted services such as trusted storage, user identification, or boot image verification.

There are three problems that we need to solve to support such trusted services: (i) we need to define the states and transitions that allow to represent the utilization of system resources based on actions (triggered by an application or the OS); (ii) we need a *Reference Monitor*,

as the ones in Section 3.3, that can interpret these actions and generate a usage decision, and (iii) we need to enforce the generated usage decision.

Using this state machine abstraction as our central hypothesis, we present in this Chapter the main conceptual contribution of this thesis: a two-phase enforcement mechanism that addresses the three problems presented above. This mechanism we propose enforce that applications cannot access system resources unless they are enabled to do so by the reference monitor, which implements usage control policies. We denote this mechanism ***Split-Enforcement***. Our contribution relaxes the trust assumptions made in all previous work on run-time security. Code that executes in the untrusted area remains untrusted; this includes applications, the OS, and all the peripherals associated with it (Chapter 3). This defines our attack model.

The rest of this chapter is organized as follows. First, we present our hypothesis. Second, we explore the design space. Here we will cover different approaches, including split-enforcement, and argue for the benefits and inconveniences in each of them. Finally, we present the approach that we will develop in the rest of the thesis, and argument our decision in terms of technology and use cases. We invite the reader to refer to the Glossary (12.2.3) to get familiar with the concepts that we are referring to throughout this chapter. Before we continue with our hypothesis, we provide a background description of our approach, using the same terminology as we will use in the rest of this thesis[1].

6.1 Background

We assume an architecture that supports the hardware separation between a *Trusted Area* and an *Untrusted Area*, as described in Chapter 2. Thus, we also assume two execution environments: a *Trusted Execution Environment (TEE)* that leverages a software stack using resources from the trusted area, and a counterpart for the untrusted area that we denote *Rich Execution Environment (REE)*. The software stack in the untrusted area (i.e., REE) is composed by a *Commodity OS* and applications that execute on top of it. Since all applications in the untrusted area are by definition untrusted, we sometimes refer to them as *Untrusted Applications* in order to emphasize this fact. We reserve the right to also denote them as *Untrusted Services*, specially when we refer to applications running as services (e.g., cloud services). The software stack in the trusted area defines the *Secure Area*; namely, the execution environment in the trusted area. The secure area is synonymous with a TEE. In some hardware platforms, the trusted area and the secure area might be the same, but this is not the general case. The secure area can then be composed by a *Secure OS* and *Secure Tasks*, which execute as applications on top of the secure OS.

Since not all system resources are sensitive, and in order to not introduce unnecessary overhead, we define the subset of system resources that are sensitive as *Sensitive Assets*. Note that system resources also include data stored in secondary storage. In this way, sensitive assets gather any component of the system that is sensitive: sensitive information (e.g., contact list, archived mails, passwords), software resources (i.e., libraries, interfaces), and

[1]Note that if this thesis is read in PDF format, all terms link to its entry in the Glossary (12.2.3).

peripherals (e.g., microphone, secondary storage, network interface). The next step is informing the device about such sensitive assets. For this purpose we define the *Device Usage Policy*. Here, a set of policies orchestrate the behavior the device as a whole with regards to its sensitive assets.

In order for untrusted applications to explicitly state which sensitive assets they intend to use at run-time, we define the *Application Contract*. In platforms such as Android, this is already done within the application manifest[2]. In [97], Dragoni et al. also propose the use of contracts that are shipped with applications. It is important to mention that the application contract is not enforcing, but a statement of *"good will"*. As we will see, it can help the reference monitor to classify untrusted applications when they violate the device usage policy.

Finally, we have referred before to the *Reference Monitor* as a component that enforces a set of usage policies. These usage policies include the application contract and the device usage policy. A core component of the reference monitor is the *Usage Engine*, which generates the usage decision as a function of (i) the state of the application using sensitive assets, (ii) past use of sensitive assets (mutability in Section 3.1), (iii) the actions carried out by the application, and (iv) contextual information of the device. We will describe in detail how the reference monitor generates and enforces these usage decision in our hypothesis.

6.2 Hypothesis

In order to enforce usage control, every application should embed an application contract. When an application is installed, its contract is evaluated against the device usage policy governing the device. If the application contract satisfies the device usage policy the application is installed, and the application contract is associated to the application; if not, the installation is aborted[3]. Note that we assume all applications and services to be untrusted. This means that the evaluation of the application usage contract is just a pre-agreement where an application states what it intends to do when it executes; the actual enforcement of the contract occurs when the application actually executes.

At run-time, application access to system sensitive assets depends on its application contract. Every time that an application requests a sensitive asset, the request is evaluated against its usage contract. If the materialization of the request is consistent with the contract, the request is served; otherwise the request is denied. Here, it is the application's responsibility to deal with a denied request. The same applies for the OS, which is governed by the device usage policy. In general terms, every component conforming the typical software stack (i.e., applications, libraries, kernel, and drivers) is untrusted, and therefore any use of sensitive assets must comply with the device usage policy. Note that this assumption alone differentiates our attack model from the rest of the state of the art on run-time security (Chapter 3), where in one way or another assumptions are made regarding the scope of

[2] `http://developer.android.com/guide/topics/manifest/manifest-intro.html`

[3] Actually, the decision of aborting the installation or triggering a prompt window to let the user decide is, in itself, a policy in the device usage policy.

environment where applications execute, the attacks against the OS, the immutability of the kernel, or the resilience of the calls from the (equivalent to the) untrusted to the trusted area.

From a user perspective, it is important to understand how the enforcement of usage policies affects the usability of the device. From the expectations that users might have of the applications running on their devices (e.g., use of sensitive assets, compliance with the policies the user has defined), to the user interface (UI) through which policies are accessed. Indeed, how the user interacts with devices implementing usage policies [266], and trusted areas in general [187, 273, 236], is a central question for interaction designers. Here, how to educate users to perceive their devices as potential targets of a cyberattacks, and *pick security over dancing pigs* [203] is also a relevant problem for the security community, and also for the educational community in general. Also in this context, we believe that the role of the *default setting* will become more relevant. Research in this area is active in the context of privacy settings [2, 185], electronic commerce [41], or retirement portfolios [272, 44]. As these services get materialized in applications, default settings will gain more and more importance. In our proposal, default setting would naturally extend to (i) application contracts, where applications not embedding their own could be assigned a *Default Application Contract*, restricting the application capabilities to a minimum; and specially to (ii) the device usage contract, where the default setting prevents leakage of sensitive data. We acknowledge that the adoption of our proposal relies in part on these issues being solved, but they are out of the scope of this thesis. We consider them however as an interesting topic for future research.

6.2.1 Application State

A key issue is to devise an efficient way to represent the usage of sensitive assets. This representation will shape the mechanisms provided by the commodity OS to enforce usage control. Conceptually, we want untrusted applications to be able to directly manipulate sensitive assets, dealing with the enforcement of usage policies as a separate problem. We propose a state machine for representing the use of sensitive assets. In this way, an untrusted application oscillates between *untrusted state*, when using non-sensitive assets, and *sensitive state*, when using sensitive assets. Untrusted applications should also be able to offload specific tasks to the TEE. Hence, we define a third state - *outsource state* -, in which the application's control flow is passed to a secure task in the TEE. Figure 6.1 represents the states and transitions for an untrusted application.

***Untrusted State*:** The untrusted application has no access to sensitive assets. The untrusted application reaches this state either because it has not made use of any sensitive assets, or because it has released them.
***Sensitive State*:** The untrusted application has access to sensitive assets. The untrusted application reaches this state if and only if the reference monitor has allowed it. This decision is based on a usage request, accessibility to other sensitive assets, and the device's context.
***Untrusted State*:** The untrusted application outsources a specific task to the secure area (i.e., executes a secure task). The secure task takes the control flow, executes, and returns it to the untrusted application when it finishes. This corresponds to traditional secure co-processor's use cases where sensitive assets are only accessible from within the TEE.

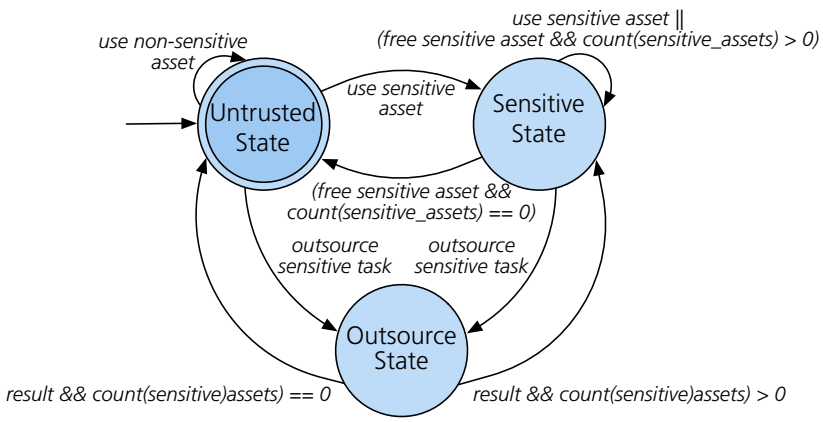


Figure 6.1: Application State and Transitions. While the states represent an untrusted application, the state machine needs to be trusted; it resides in the secure area.

An untrusted application starts and finishes its execution in untrusted state. This is a way to enforce that any sensitive asset used by the untrusted application has been properly released. Thus, during its execution an untrusted application can move between untrusted, sensitive, and outsource state based on the assets it is using. The state in which it executes is transparent to the untrusted application itself; the state is only used by the reference monitor as application metadata to generate a usage decision. Since the state machine impacts the actions that an untrusted application is able to perform, it must reside in the trusted area, and only be accessible to the reference monitor (which executes in the secure area).

6.2.2 Application Monitoring

While the state machine in Figure 6.1 is fairly simple, it allows to create a general execution context to verify and enforce application contracts at run-time. Still, when an application is in sensitive state, further accesses to sensitive assets cannot be captured. The missing part is a usage engine that maintains a historical record of sensitive assets in use for each untrusted application in sensitive state (i.e., mutability in Section 3.1). The reference monitor would then have enough information to generate a usage decision based on the application contract and the device usage policy. Figure 6.2 depicts such a usage engine.

Every untrusted application executes in the untrusted area. While no access is attempted to sensitive assets, the application's control flow is never given to the trusted area. When an action involving a sensitive asset takes place, the reference monitor captures it and feeds it to the policy decision point, where the historical record is maintained. The policy decision point generates then a usage decision based on the current request, past actions, and the device's context (i.e., the state of other applications, and the device's internal state). The usage decision is returned to the reference monitor, which can then evaluate it and either allow, or abort the action. Depending on the usage decision, the reference monitor triggers a corresponding transition in the application's state machine. When the untrusted application regains the control flow, it is expected to interpret the usage decision given by the reference monitor. Afterwards it continues its execution. This flow is repeated until the untrusted

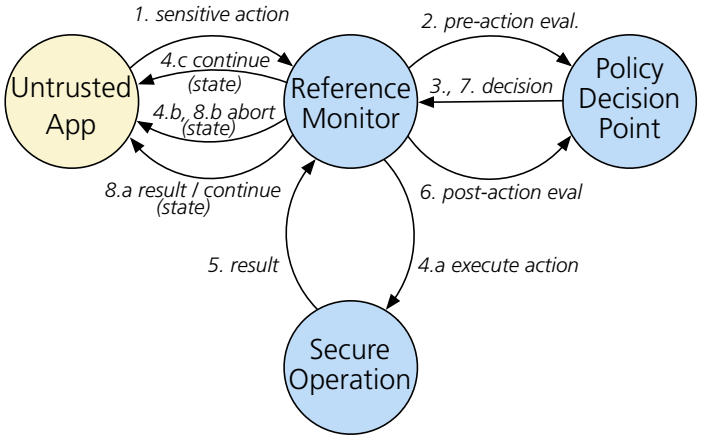


Figure 6.2: Untrusted application execution monitoring. *Untrusted Application* state corresponds to the states depicted in Figure 6.1. The decision (d) made by *Policy Decision Point* responds to $d = f(App\_S, Int\_S, Act, Assets, C, P)$ / *App_S = Application State (Figure 6.1), Int_S = Internal State, Act = Requested Action, Assets = Assets Involved, C = Service Usage Contract, P = Device Usage Policy*

application finishes its execution.

Together, the application state machine (Figure 6.1) and the usage engine (Figure 6.2) define the mechanism to generate a usage decision based on an application's use of sensitive assets. The remaining question is how to define an architecture that (i) prevents that untrusted applications bypass the reference monitor, and (ii) enforces the generated usage decision.

6.3 Design Space

In Chapter 3 we covered the state of the art in reference monitors and trusted storage. While all the approaches discussed propose interesting frameworks to (i) enforce usage policies in the case of reference monitors, and (ii) guarantee the integrity, confidentiality, availability, and durability[4] of sensitive data in the case of trusted storage - some of them with a very strong formal verification behind them -, they either restrict the scope of innovative services, or lack a realistic attack model that responds to today's run-time attacks. By placing their core protection engines (e.g, usage engine, encryption engine, integrity engine) in the same untrusted environment as the applications they intend to protect, the latter make the underlying assumption that the commodity OS (including its kernel) is trusted. In an attack model where the attacker gains root privileges, a protection engine located in the (equivalent to the) untrusted area could be directly tampered with, bypassed, or its output modified.

By using the secure hardware presented in Chapter 2, a protection engine placed in the trusted area would be resilient against different types of attacks; depending on the technology

[4]We name all the properties that a storage solution should guarantee, even when most of the approaches presented for trusted storage only covered a subset of these properties.

used, the trusted area would exhibit different degrees of tamper-resistance. However, attack vectors where the protection engine is either completely bypassed, or its output is modified remain the same. Even if we assume that the trusted area hosting the protection engine is *tamper-proof*, we would need to face these attack vectors. We articulate them in two concrete problems: If the untrusted area can be fully compromised, how can we prevent that an untrusted application bypasses the calls to the protection engine in the trusted area completely? And more importantly, if the protection engine gets to generate a legit output (e.g., usage decision, ciphertext) how can it be enforced in an environment that is untrusted by definition?

This problem is represented in Figure 6.3 by the continuous, red line. Here, a component - usage driver - is added to the commodity OS with the responsibility of communicating with the reference monitor in the secure area. While the usage decision is legitimately generated, the enforcement takes place in the untrusted area (in the usage driver). In this scenario, if sensitive assets are accessible to the untrusted area (it cannot be otherwise), the policy engine could be either completely bypassed, or its output modified. Indeed, since attacks could target any component in the untrusted area, the commodity OS could be tampered with in such a way that applications are tricked to believe that the sensitive assets they are using are governed by the reference monitor, when in reality they are not. From a security perspective, approaches following this design either (i) consider that the kernel is immutable, and therefore discard a malicious root, or (ii) make the underlying assumption that specifically targeting the usage driver entails an intricate attack, and therefore the attack can be ignored. Today, either of them is a leap of faith.

Assuming that the protection engine executes in a trusted area separated by hardware, we are able to define two main design options for run-time security that do not make these assumptions:

1. **Sensitive assets are only processed in the secure area.** Sensitive assets belong in the trusted area, and are not exposed to the untrusted area. Any use of a sensitive asset must be done within the secure area. In this way, the intervention of reference monitor for accessing a sensitive asset is guaranteed. (Figure 6.3 - green, dashed line).

2. **Protected sensitive assets are processed in the untrusted area.** Sensitive assets are directly available to the untrusted area, but protected by the trusted one. Any use of a sensitive asset is mediated by the reference monitor, however the actual access to the sensitive asset is done from within the untrusted area (Figure 6.3 - blue, dash-dot line).

6.3.1 Secure Drivers

In this model, sensitive assets are only processed in the secure area. This is, sensitive assets belong in the trusted area, and are not exposed to the untrusted area. Any use of a sensitive asset is mediated through the reference monitor in the secure area. Put differently, an untrusted application is forced to go through the reference monitor in order to access a

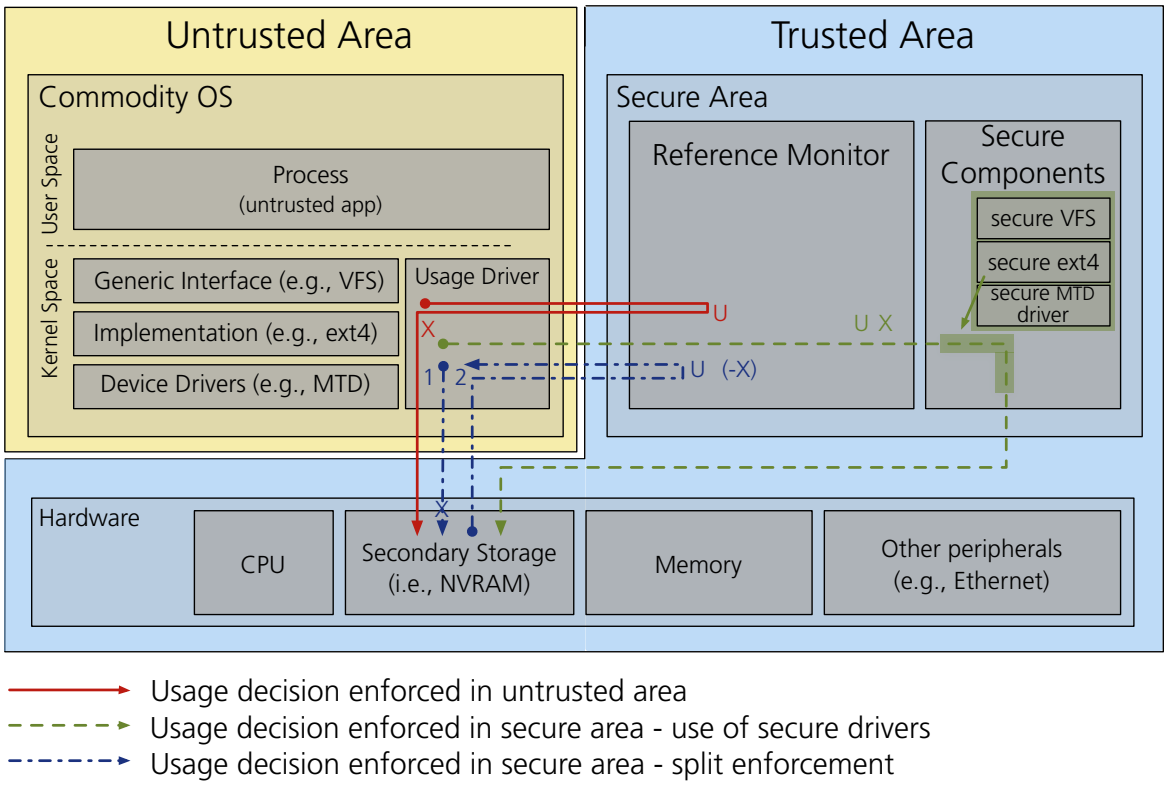


Figure 6.3: Three models of how access and usage control decision, and enforcement can be organized between the trusted and untrusted areas.

sensitive asset. In this way, the reference monitor can generate a usage decision and enforce it with the guarantee that the untrusted application cannot bypass the usage control point; the sensitive asset is not available if the control does not go through the secure area. An advantage of this approach is that the software path simply switches over from the untrusted area to the trusted area whenever a sensitive asset (data or peripheral) is accessed. This minimizes coupling between the trusted and the untrusted areas, and simplifies communication. On the down side, this approach entails a dramatic increase of the TCB; each system primitive interacting with a sensitive asset must be replicated in the trusted area.

Let us take trusted storage as an example to see the consequences of this approach. If sensitive data is stored on tamper-resistant storage, then it can be stored in clear. Indeed, tamper-resistant storage is not exposed to the untrusted area, and it can resist to large classes of hardware attacks in case the device gets stolen or physically compromised. In this way, the secure area writes sensitive data on the tamper-resistant storage while the reference monitor is guaranteed to mediate all accesses to it. The secure area must thus contain the entire IO stack. This involves at least replicating: the appropriate disk device driver (e.g., MTD), a block device interface, and a secure version of a file system (e.g. ext4). For a flexible solution, several disk drivers, files systems, and an equivalent to the Linux virtual file system (VFS) layer would also be necessary. Only considering the ext4 file system and the VFS layer we are in the order of several tens of thousands of LOC, not counting the

buffer cache and cryptographic algorithms[5]. The number of LOC gets only higher if we start considering more complex file systems (e.g., btrfs). If a usable trusted storage solution is to be provided, the trusted area should probably also include a database system. As a consequence, the TCB in the trusted area becomes complex and large.

This approach represents the green, dashed path in Figure 6.3. Here, the usage decision is generated when the call to the secure stack occurs (U), and the enforcement (X) takes place right after. In this example, targeting storage primitives, at least an implementation of the ext4 file system, and the appropriate disk driver must be provided in the trusted area.

The same applies for any other sensitive assets. If a network peripheral (e.g., Ethernet card) is identified as a sensitive asset, then the whole network stack needs to be replicated in the trusted area. This includes at least: a device driver, an Ethernet interface, an implementation of the IP and TCP/UDP protocols, and a socket interface. Again, we are in the order of several tens of thousands of LOC.

Such an increase of the TCB without a formal verification of the secure area is not an option; the potential number of bugs in the secure area would counteract any benefit added by the hardware security extensions isolating it [146]. Verifiable OSs such as seL4 [176, 211] or MirageOS [194] are interesting candidates to support this approach. Here, it is important to emphasize that current verification techniques however, set the limit on 10K LOC [176] to handle proper verification. Even if we assume that techniques in the near future can handle one order of magnitude more of LOC (i.e., 100K LOC), general purpose OSs are in the orders of millions of LOC (e.g., the Linux kernel only is over 15M LOC). In this way, most "secure" software stack approaches rely on non-verified microkernels, which are considered trusted merely because they are isolated. Here, we do not consider certified microkernels as a good solution since we do not believe that run-time security should rely on simply placing trust in third party entities, who might have other interests. Certification is a good way to maintain a clean codebase, in the sense that it can be conceived as a code peer-review process, but it does not provide a solution to the problems we address in this thesis. In other words, we define trust based on policy enforcement at run-time, not on the possibility to sue a bad certification *post-mortem*. Another major problem with this approach is the cost and storage capabilities (size) of tamper-resistant units with respect to normal storage hardware. In a world where data is measured in zettabytes[6] [107], a solution that does not scale both technically and economically is simply not good enough. Here, it can be argued that an encryption scheme can be used so that encrypted data is stored in normal storage hardware, while encryption keys are stored in a tamper-resistant unit. Still, the problem concerning the increase of the TCB to support the I/O stack remains.

6.3.2 Split Enforcement

In this scenario, protected sensitive assets are processed in the untrusted area. Here, while sensitive assets are still protected by the trusted area, they are processed directly in the

[5] Even when storing plaintext, calculating a hash of what is stored (e.g, page, key-value, object) is necessary to provide integrity.

[6] `http://techterms.com/definition/zettabyte`

untrusted area. In order to prevent the attacks against the reference monitor that we have discussed above (bypass and output modification), we propose a solution based on a two phase enforcement, where sensitive assets are locked at boot-time in a first phase, and unlocked on demand at run-time on a second phase, always that the reference monitor allows it. Ultimately, the idea is to force untrusted applications to make use of the trusted area in order to operate correctly; even when accessible, locked assets are unusable. As mentioned in the beginning of this chapter, we denote this technique *Split-Enforcement*.

Let us take the same trusted storage example as before to illustrate how split-enforcement works. In this case, external storage devices are not secure. Sensitive data must thus be encrypted before it is stored there; encryption keys are stored in tamper-resistant hardware (e.g., secure element), only accessible to the trusted area. The secure area is in charge of encryption/decryption, but not of reading from or writing to the storage device; the untrusted area has access to the storage device and can directly issue IO requests to it. An advantage of this approach is that there is no IO stack in the trusted area[7]. Another advantage is that the TCB is limited; all components around the I/O stack need not be replicated in the trusted area (e.g., the Linux VFS). Here, we trust the encryption scheme, there is thus no need to control how encrypted data is accessed from the untrusted area. The untrusted area stores encrypted data; there is thus no guarantee that data is actually stored. This can be a problem in terms of availability, but not in terms of integrity or confidentiality (we discuss this in detail in Section 8.2.1.

While encryption and decryption must take place in the trusted area, it is possible to let untrusted components process decrypted data, under a set of conditions:

1. Decrypted data should, by default, never leave the trusted area.
2. When authorized to reference decrypted data, untrusted components should not be able to transmit it (via memory copy, storage on the local file system, or via the network) unless they are authorized to do so.

Split-enforcement is depicted in Figure 6.3 by the blue, dash-dot line. The first phase is numbered with *1*. Here, a trusted component locks down all sensitive assets (e.g., decrypted data, peripherals). The first phase always takes place for sensitive assets, it is the default setting, and changing it entails that an asset is not sensitive anymore. The locking of sensitive assets is represented by X. In the second phase (*2*), before the sensitive asset is unlocked in the trusted area, a usage decision (U) is generated by the reference monitor. If the usage decision complies with the application contract, the locked sensitive asset is unlocked ($-X$). The second phase is triggered by untrusted applications on demand. If the component in the trusted area that unlocks the sensitive asset is not called, then the sensitive asset remains locked and therefore cannot be used. The applicability of split-enforcement relies thus on the specific mechanisms used to lock sensitive assets. Before the sensitive asset is unlocked, the usage engine generates a usage decision and evaluates it (Section 6.2.2). If positive, the sensitive asset is unlocked. A permission error is returned otherwise. This way, all attack vectors aiming to bypass the reference monitor are eliminated. In Section 10.1 we provide an

[7]Storing cryptographic keys to tamper-resistant storage requires a simple driver, not an IO stack.

extensive analysis of other attack vectors such as DoS, return oriented programming (ROP), or secure path hijacking.

Since the actual processing of sensitive assets takes place in the untrusted area, such an approach maintains a low TCB, and maximizes the reuse of existing components (in the untrusted area). There are two additional advantages. First, there is no need to replicate sensitive, complex, pieces of code that are both time-tested and community maintained (e.g., lock-less synchronization algorithms such as RCU, or the slab allocator in the Linux Kernel), which can lead to bugs and errors that have already been resolved. Second, this approach does not impose any specific software stack to be used in the trusted area. That is, formally verifiable OSs as the ones mentioned above. An additional benefit is that this solution can scale as storage hardware evolves; a small portion of tamper-resistant storage can maintain the encryption keys that protect petabytes of data stored in normal storage hardware.

The remaining question is how to lock different types of sensitive assets in the trusted area in order to fulfill the second condition. This should be done without interferences from untrusted components. While the concept of split-enforcement is generic, some the actual locking primitives are platform-specific, since they are supported by concrete hardware extensions. In next section we present the approach that we use to implement split-enforcement. In Chapter 8 we describe in detail the specific mechanisms to lock and unlock sensitive assets, and report on our prototype implementation.

6.3.3 Approach

We choose to explore the second alternative, where sensitive assets are directly available to the untrusted area, but protected by the trusted one. In other words, we pursue an implementation for split-enforcement. We believe that formal verification is an interesting path for research. However, we do not think that it alone can protect against all attack vectors[8]. Besides, it introduces, by design, strong limitations in terms of the software stack that can be used in the trusted area, as well as in the process of producing new drivers - all the components need to be verified, first independently, and then as a whole. As a consequence, it constrains the scope of trusted services. Also, we are interested in investigating the problem of providing run-time security from a systems perspective, not from a formal one. Finally, we would like our research to be applied widely in widely used operating systems such as Linux, and impact its future design to improve security.

We choose ARM TrustZone for our prototype implementation of split-enforcement. The main argument is that TrustZone, unlike traditional secure co-processors, allows to define a dynamic security perimeter defined by CPU, memory, and peripherals, that can change at run-time (Section 2.4). From our perspective, this is a move away from security by isolation and obscurity, and towards a better synergy between trusted and untrusted areas.

[8]In [240], an interesting discussion regarding the scope of formal verification is presented. The main argument is that without IOMMU to guarantee the isolation of device drivers, formal verification that only targets software components, and ignores the hardware, is useless. While the discussion leads to exalting the Qubes OS architecture [241], the argument for IOMMU is very valid, and represents a limitation for current formally verified OSs.

6.4 Target Devices

The fact that TrustZone is only supported in ARM processors limits the scope of our implementation (though not of split-enforcement as a concept). Still, there are three types of devices featuring ARM processors that are of interest to our research, and that have backed our decision of implementing split-enforcement using TrustZone:

1. **Mobile devices.** Specially after the adoption of BYOD, mobile devices have become a niche for data-centered attacks; one single device stores personal and enterprise data, executes a combination of enterprise-oriented apps (e.g., mail, VPN) and personal apps (e.g., social networks, games) that have access to all (sensitive) data simultaneously, and connects to private and public networks indifferently. Hence, maximizing the device's exposure to software attacks. Also, mobile devices are by nature more subject to being stolen. Hence, maximizing the device's exposure to hardware attacks. Even when companies put limits to BYOD as a way to prevent their services being compromised, research shows that employees force their devices into their enterprise services [87]. This is a scenario where usage control based run-time security can be used to protect sensitive data and other sensitive assets from being leaked or misused (e.g., connecting a *rooted* device to the company's VPN, accessing sensitive enterprise data from a public network, or unwillingly synchronizing personal pictures to the company's cloud). Since most mobile devices today feature one (or several) TrustZone-enabled ARM processor(s) (e.g., Cortex-9), they represent the perfect context for usage control experimentation using split-enforcement. Note that at the time of this writing Android and iOS devices combined have a market share of 96%[9]. Most Android devices are powered by Cortex-A processors; Apple devices are powered by Cortex-A9 processor in older models, and ARMv8 processors in newer ones[10]. All are TrustZone-enabled processors. From the software perspective, Android is powered by a Linux-based kernel, iOS is powered by a UNIX-based one.

2. **I/O devices** Next generation converged enterprise I/O devices will share responsibilities with the host in order to improve performance [220, 48, 49, 50]. A class of storage devices run a full-featured OS to manage them (e.g., Linux). Examples of this type of peripherals include enterprise storage devices (e.g., high-end PCIe cards) and rack-scale storage components. We can see this trend in hardware that became recently available: the Samsung 840 EVO SSD is powered by a Samsung 3-core (ARM Cortex-R4) MEX Controller [245], the Memblaze PBlaze4 is powered by the PMCs Flashtec NVM Express (NVMe) multicore controller[11]; their previous model, PBlaze3, was powered by a full-featured Xilinx Kintex-7 FPGA[12]. Here, ARM processors are expected to be more frequent, given their compromise between performance and power consumption. Powerful ARM-based processors such as ARMADA 300[13] or Spansion Traveo[14] that market themselves as solutions to power high-end media servers, storage,

[9] http://www.idc.com/prodserv/smartphone-os-market-share.jsp
[10] http://www.devicespecifications.com/en/model-cpu/5d342ce2
[11] http://pmcs.com/products/storage/flashtec_nvme_controllers/pm8604/
[12] http://solution.zdnet.com.cn/2014/0303/3012962.shtml
[13] \unskip\penalty\@M\vrulewidth\z@height\z@depth\dpff
[14] http://www.spansion.com/Products/microcontrollers/32-bit-ARM-Core/Traveo

and networking, together with the expansion of SoCs combining ARM processors and a FPGA (e.g., Xilinx Zynq[15,16]) are good examples that this trend is gaining momentum.

There are two reasons why these upcoming devices are of interest from a security point of view.

i) Hardware security extensions are by definition part of the root of trust, the CPU is expected to execute the instructions given to it, and I/O devices are assumed to work as expected (i.e., when reading from a logical address, we expect to receive data from the correct physical address; when receiving a write completion, we expect the write to have completed). The question is: do these assumptions hold when I/O devices maintain their own software stack? We believe that the answer is no. We believe that these devices are going to be the target for elaborated attacks with the objective to leak sensitive data, without the host's OS having any chance to detect them. As a result, the attack model for systems incorporating these new devices will have to change. In general, we believe that local trusted I/O devices will become untrusted as they grow in complexity.

ii) Cloud service providers are becoming less and less trusted. From the Snowden revelations about mass surveillance [142], to repeated attacks to cloud services[17,18,19,20], the general conception is that the cloud is not secure, and therefore untrusted. As a consequence, the infrastructure - which includes the hardware - is also untrusted. In this way, research in secure cloud has focus on implementing security-critical operations locally (e.g., encryption), and using the cloud for storing and sharing encrypted sensitive data. However, the fact that cloud services might be running on top of hardware leveraging security extensions opens the possibility for enabling secure processing in the cloud. A good example is Haven [39], a framework to remotely upload secure tasks using Intel SGX's enclaves (Section 2.3.2) as a TEE. If I/O devices in the cloud enable a trusted area, we can expect more research moving towards this direction, i.e., defining remote trusted services (e.g., trusted storage) in an untrusted infrastructure. In general, we believe that cloud untrusted I/O devices will become trusted as they grow in complexity.

We envision that a solution that is able to mitigate attacks for this new generation of I/O devices will revolutionize the area of run-time security, specially in the context of the secure cloud. Therefore, we consider this new generation of I/O devices as the perfect context for usage control experimentation using split-enforcement. Here, we will narrow our work to investigating a trusted storage solution in software-defined storage supported by new generation SSDs (Section 8.2.1).

3. **ARM-powered servers** The advent of ARMv8 will open the possibility for server-side ARM-powered machines. The two known ARMv8 cores at the time of this writing

[15] http://www.xilinx.com/products/technology/ultrascale-mpsoc.html
[16] http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html
[17] http://www.securityweek.com/apple-sees-icloud-attacks-china-hack-reported
[18] http://www.infosecurity-magazine.com/news/ddos-ers-launch-attacks-from-amazon-ec2/
[19] http://techcrunch.com/2014/10/14/dropbox-pastebin/
[20] http://en.wikipedia.org/wiki/2014_celebrity_photo_leaks

- Cortex-A53, and -A57 - include the TrustZone security extensions [27, 28]. Here, we expect to experience a combination of the problems described for the former two types of devices. From the server's point of view, we would have a system powered by one (or several) ARM processor(s) with access to a trusted area supported by TrustZone where its peripherals also feature a corresponding trusted area, being accessed by untrusted clients. From the client's point of view, we would have a device that also has access to a TrustZone-enabled trusted area, executing trusted services in the trusted area of an untrusted cloud. How the trusted areas will interact together to verify each other, leverage trust, and divide workloads will be, in our view, an interesting topic for future research. This case serves more as a motivation than as an actual experimentation context. However, since the problems we have been dealing with are equivalent to the ones in a future ARMv8-powered cloud, we expect our contribution to be directly applicable.

Since TrustZone is not tamper-resistant, we need external hardware to implement tamper-resistant storage. This is necessary to store encryption keys and certificates in cleartext from the secure area, and leverage this to store encrypted sensitive data in the untrusted area. Counting on such tamper-resistant unit is a requirement for split-enforcement; other wise physical attacks would be out of scope. Since ARM-power device do not necessarily feature a default tamper-resistant unit, we incorporate a secure element (SE) to the trusted area. As described in Section 2.1.1, a secure element is a special variant of a smart card, which is usually shipped as an embedded integrated circuit in mobile devices, and can also be added to any device equipped with Near Field Communication (NFC), a microSD, or an Universal Integrated Circuit Card (UICC). This allows secure elements to be easily added to laptops, set-top boxes, or servers. As a reminder, a secure element provides data protection against unauthorized access and tampering. It supports the execution of program code in form of small applications (applets) directly on the chip, as well as the execution of cryptographic-operations (e.g., RSA, AES, SHA, etc.) for encryption, decryption and hashing of data without significant run-time overhead [147].

The combination of TrustZone and a SE allows us to define a security architecture under an attack model that we consider realistic for mobile devices, new generation of I/O devices, and upcoming ARMv8 servers. This is, an attack model where all untrusted components (i.e., untrusted area) remain untrusted, and do not implement any security-critic operations. TrustZone allows for the implementation of split-enforcement. The SE provides tamper-resistant storage for critical assets (e.g., encryption keys). At this point, we have defined our vision, the motivation behind it (i.e., target devices), and the hardware to support it. In the following chapters (7, 8, 9)) we focus on the software part, and present our design and implementation.

6.5 Conclusion

In this chapter we have presented the main conceptual contribution of this thesis: *Split-Enforcement*: A two-phase enforcement mechanism that allows to enforce usage policies without increasing the TCB in the secure area with replicated functionality (e.g., I/O stack).

In the first phase, *Sensitive Assets* are locked are boot-time. In the second phase, these sensitive assets are unlocked on demand at run-time. Since the locking and unlocking mechanisms reside in the secure area, untrusted applications are forced to go through the reference monitor in the secure area, thus being subject to the usage decision generated by it.

We start by presenting our hypothesis: A state machine abstraction where (i) states are represented by the resources available to an untrusted application, and (ii) state transitions represent the utilization of system resources. As long as these state transitions adequately match actual system resource utilization, by controlling the transitions, it is then possible to enforce how system resources are accessed and used. While the state machine can represent all system resources, we are specially interested in sensitive assets.

Based on this hypothesis we cover the design space in terms of usage policy enforcement. Here, we present the deficiencies of the state of the art in reference monitors and usage control implementations and propose two alternatives: one where sensitive assets are only processed in the secure area; and one where sensitive assets are protected by the secure area, but available to the untrusted area. We argue for the benefits of the latter, which represents in fact split-enforcement. Finally, we present an approach for implementing split-enforcement based on the ARM TrustZone security extensions, where we argue for the benefits of using this secure hardware in terms of (i) the integrity of the trusted and untrusted areas, (ii) the scope that we envision for trusted services, and finally (iii) the devices that we intend to target with our contribution.

Chapter 7

TrustZone TEE Linux Support

In this Chapter we present the components that give OS support for a TrustZone TEE. We can divide these components in two parts: (i) a framework that can leverage a *Trusted Execution Environment (TEE)* using TrustZone to support it; and (ii) the driver that allows to use it from within the Linux Kernel. Since all our implementation is done in Linux, all technical descriptions and terminology is restricted to Linux systems. Also, since we use git[1] for development, we use some git-specific terms to describe the process of implementing these components. A good introduction to git, its terms, and scope can be found here [59]. Finally, also with regards to terminology, when describing device driver support in the Linux kernel we use the terms device and peripheral indistinctly. Note that here a device (peripheral) is a component attached to *Device*. We apologize for using confusing terms, but since this chapter is entirely about Linux kernel development, we want to respect the terminology used in this community. When describing target platforms, we define them in terms of the configuration files (*defconfig*) for compiling the Linux kernel, and the *device tree* describing the device in itself; a target platform denotes the architecture, the peripherals attached to the specific device, and the properties of these peripherals.

7.1 TrustZone TEE Framework

TrustZone introduces a new privilege mode in the processor that is orthogonal to the existing processor modes for user space, kernel space, and hypervisor. This allows to define the secure world and the non-secure world that serve as a base to provide the run-time security primitives we reason about in this thesis (Section 2.4). However, TrustZone does not define what is placed in the secure world; not a specification, nor a list of secure primitives. While Global Platform has defined a set of TEE specifications [66, 67, 68, 70, 71, 69, 72, 65] primarily targeting use cases in the mobile community (e.g., banking, DRM), this vagueness allows TrustZone to be used in many different contexts: from a secure environment designed to execute specific *Run-Time Security Primitives* (e.g., trusted storage, secure keyring), to a thin hypervisor designed to meet real-time constrains, as it is the case in the automotive

[1] `http://git-scm.com`

& aerospace industry. Today, an increasing number of companies and frameworks covering all these different use cases are appearing as run-time security gains popularity. Examples of companies and organizations include Mentor Graphics[2], Sysgo[3], Wind River[4], Sierraware[5], Linaro[6], and the Nicta - Open Kernel Labs collaboration[7,8]. The most relevant TrustZone TEE frameworks (covered in Section 2.4) include Open Virtualization, TOPPERS SafeG, Genode, Linaro OP-TEE, T6, and Nvidia TLK.

Choosing a TrustZone TEE framework depends then on the use cases defined for the trusted area. Frameworks such as Nvidia TLK or Open Virtualization cover traditional Global Platform use cases targeting short duty cycle offload of secure tasks to the trusted area, while others such as TOPPERS SafeG target a secure area that runs largely independent (not as a slave) executing at high duty cycle, and probably under real-time constrains. While the latter covers more general use cases, it comes at a price; typically in order to meet real-time requirements, a processor is dedicated to the trusted area. Put differently, the choice of the TEE framework has an implication on the hardware platform itself. Another important aspect to consider when choosing the TrustZone TEE framework is the license under which it is released. Here, given the systems, experimental nature of our research, and considering that we do it in an academic context, counting on an open source TEE framework is a strong requirement for us.

In order to meet the requirements defined in Chapter 6 for implementing split-enforcement we would need a TEE supporting a largely independent trusted area so that the reference monitor could generate and enforce usage policies, while the commodity OS is still properly attended by the untrusted area. In this way, processes ruled by complex usage decisions could be put to sleep while other processes - governed by simpler usage policies - would still execute in a multi-process, preemptive fashion. TOPPERS SafeG provides such a TrustZone TEE. However, at the time we were making design decisions regarding the framework that could support split-enforcement[9], SafeG was still under development at Nagoya University in Japan[10]. In fact, most of the open source frameworks that are becoming available today - which we have analyzed and gathered in this thesis - were not available back then. At that point, the only open source framework we could find was Open Virtualization[11]. Open Virtualization was developed by Sierraware, an embedded virtualization company based in India. To the best of our knowledge, Open Virtualization *was* the first open source alternative leveraging ARM TrustZone security extensions. Open Virtualization is composed by (i) SierraVisor, a hypervisor for ARM-based systems, and (ii) SierraTEE, a TEE for ARM TrustZone hardware security extensions. While the hypervisor is an interesting contribution, it is the TEE open source implementation that represented a turning point in the TEE community, since it opened the door for developers and researchers to experiment with

[2] `http://http://www.mentor.com`
[3] `http://www.sysgo.com`
[4] `http://www.windriver.com`
[5] `http://www.sierraware.com`
[6] `https://www.linaro.org/blog/core-dump/op-tee-open-source-security-mass-market/`
[7] `http://ssrg.nicta.com.au/projects/seL4/`
[8] `http://l4hq.org`
[9] The idea of split-enforcement that we had at that time; mainly focused on enforcing usage control policies.
[10] `https://www.toppers.jp/en/safeg.html`
[11] `http://www.openvirtualization.org`

TrustZone [131].

7.1.1 Open Virtualization Analysis

Sierraware designed and implemented Open Virtualization targeting traditional TEE use cases for mobile devices: mobile banking, DRM, and secure communications as defined by Global Platform's TEE specification. As we have mentioned, in this type of TEE the secure world is designed to be a slave of the non-secure world, where secure tasks are executed on demand in a low duty cycled environment. This represents the *offload state* in Figure 6.1, where an untrusted application offloads a specific task to the TEE. Let us analyze in detail Open Virtualization. We will scrutinize the framework at all levels, from the design, to the interfaces, and the actual implementation (i.e., code quality). The idea is to find the strengths and deficiencies of Open Virtualization. As a matter of fact, this is a crucial task, since it will help us understand the environment in which we will implement our trusted services.

7.1.1.1 TEE Communication Interface

Open Virtualization provides two interfaces to communicate with the secure area (TrustZone's secure world): a custom made user space interface called OTZ (*otz_main_client*); and a subset of Global Platforms TEE Client API [66]. Global Platform's specification is only partially implemented; however this is a normal practice in the TEE community, since the specification is very large, and its functionality can easily be decoupled. Both interfaces are implemented in a library-like format accessible to untrusted applications. In order to support these interfaces in Linux, Sierraware provides a Loadable Kernel Module (LKM) that acts as a TEE driver. This driver requires at the same time that the commodity OS kernel is patched in order to introduce support for calls to TrustZone's secure monitor (SMC call). Both interfaces use the same LKM support. We will cover Linux TEE support in detail in Section 7.2.

In general terms, applications in user space use an equivalent to system calls to make a SMC call and execute the secure monitor. The secure monitor captures the call and evaluates a flag that associates the call with a secure task, which resides in the secure area's address space. This design achieves a separation between untrusted and trusted code; untrusted applications use secure tasks as services in a client - server fashion[12]. As long as a well defined communication interface is defined for secure tasks, untrusted applications are not aware of how the service is performed, moving the responsibility entirely to the secure area. This creates low coupling between the two worlds. Moreover, since secure tasks reside in secure memory and do not depend on any untrusted resource, it is possible to authenticate the caller or evaluate the system before executing. This enables secure tasks to increase the security of the monitor, not relying entirely in the SMC call to execute secure tasks. By using asymmetric authentication, it is possible for untrusted applications to verify that the service is being provided by a specific secure task.

[12]The untrsuted application is the client (master) and the secure task is the server (slave)

While the non-secure side of the communication interface in Open Virtualization is simple and well defined, the secure side is complex and error-prone. Once the secure monitor call (SMC) is issued to the secure world and captured by the secure dispatcher, the call is forwarded to the correspondent secure task. Here, the parameters sent to the secure task need to be manually obtained from a shared stack which is implemented as a piece of shared memory between the two worlds. This same stack is also used to return values to untrusted applications. Pointers need also to be handled manually from each secure task. This mechanism is error-prone and makes it hard to verify and certify code executing in Open Virtualization's TEE. Moreover, it creates high coupling between the definition of secure tasks and the framework itself; adding a new parameter to a secure task requires code refactoring, as well as additional pointer and stack manipulation. An extra layer implementing stack operations and parameter verification is needed so that TEE developers can pass parameters between untrusted applications and secure tasks transparently, without worrying about the underlying mechanisms supporting it. We have not prioritized implementing this since it does not add any value to our research, and once the secure stack is familiar it is possible to replicate code at the cost of occasional debugging. Improving Open Virtualization in this area is future work.

7.1.1.2 Clocks, Timers, and Interrupts

In TrustZone TEE terms, a consequence of Open Virtualization's secure world being a slave of the non-secure world is that the secure world does not maintain a global state, only a per-session state, which is defined by (and depends on) non-secure world applications. Here, a particular problem of Open Virtualization is that the secure monitor does not adjust software clocks (nor timers) in the non-secure world when the secure world executes. As a consequence, the commodity OS executing in the non-secure world does not account for CPU time when secure tasks execute, thus causing software clocks in the commodity OS kernel drift with respect to the device's hardware clock. This comes as a product of requirements established for the TEE. One of the TEE design principles is that secure world has always higher priority than kernel space in the non-secure world. Open Virtualization implement this - erroneously - by making the secure world non-preemptable (i.e., disabling interrupts when the secure world controls the CPU). On ARMv7 there are two types of interrupts: external interrupts with normal priority (IRQ), and fast external interrupts with high priority (FIQ) [21]. In TrustZone, FIQs are normally associated with secure space, and IRQs with kernel space [29]. Open Virtualization disables IRQs when entering the secure world. As a result, long-running secure tasks are not scheduled out of execution, not even by a secure scheduler. While this indeed guarantees that the secure area has priority over the commodity OS, it also prevents the commodity OS from operating correctly due to clock inaccuracies; clock drift is a known issue, specially in real-time systems. Put another way, Open Virtualization does not support executing a process in the secure world in parallel to a process in the non-secure world. This will be a deficiency we will have to deal with when designing trusted services in the trusted area (Chapter 8).

7.1.1.3 OpenSSL Support

Open Virtualization offers support for OpenSSL[13]. OpenSSL is a toolkit implementing Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) protocols, as well as a full-strength general purpose cryptography library. Facebook, Google, or Yahoo are examples of enterprises relying on OpenSSL to secure their communications and data.

In our experience, Open Virtualization's integration with OpenSSL is very good. OpenSSL libraries are the default for both Global Platform's TEE and OTZ interfaces, which not only allows for code reutilization, but moves security critical operations to one place. However, there are two aspects of the OpenSSL support that called our attention. The first one is the use of EVP deprecated operations for encryption and decryption. The second one is that Open Virtualization is configured to support openssl-1.0.1c released in May, 2010. This version of OpenSSL is affected by the *Heartbleed Bug*[14], which was made public on April 7th, 2014. Last Open Virtualization release (September, 2014) still uses this OpenSSL version. We cannot however, evaluate Sierraware's commercial TEE.

Since all security critical algorithms rely on OpenSSL, we have updated the OpenSSL version integrated in Open Virtualization to a newer one where Heartbleed is fixed. At the time of this writing we are using openssl-1.0.1j, which was last updated in October, 2014. We have also modified some of Global Platform's TEE and OTZ encryption procedures to avoid the use of deprecated operations. We also use these non-deprecated operations when implementing trusted services.

7.1.1.4 Debuggability and Testing

Open Virtualization does not provide a debugging or testing framework. For our development we ported CuTest[15], i.e., a simple unittest framework with a small footprint, inside Open Virtualization. However this framework can only be used to test results in the launching application (which is located in the untrusted area). This means that it can be used for testing the behavior and correctness of untrusted applications. However, even when a bad result produced by a secure task would be exposed, such an unittest framework cannot help pointing at the specific bug, as it could be located in the secure task, the communication interface, or in the TEE framework.

In order to test and debug the secure area (i.e., secure world), we had to use traditional, low-level mechanisms and tools such as the *JTAG*, leds, or a logic analyzer. Most of the debugging mechanisms we have used were provided by the Zynq-7000 ZC702; Xilinx provides a very complete debugging framework which we could take advantage of in order to confirm the good operation of the secure area. In a few cases we had to resort to a logic analyzer in order to verify that the secure area was still executing. An example in which such a technique was necessary was during the implementation of the reference monitor in the secure area. A bad memory allocation in the secure area resulted in part of the TEE being overwritten, thus

[13] `https://www.openssl.org`
[14] `http://heartbleed.com`
[15] `http://cutest.sourceforge.net`

causing the TEE to hang. This prevented the control flow to be returned to the commodity OS; as described above, interrupts are disabled when the secure area controls the CPU. In this case, we debugged the TEE with a combined used of the logic analyzer and leds, in order to create a map of the specific operations executing in the secure area.

Debugging the untrusted area is as simple (or intricate) as it is debugging in a normal embedded environment. Again, using Xilinx's debugging framework made the task much more bearable. Also, the development process became much simpler and agile thanks to the work done by Dave Beal and Sumanranjan Mitra's team in India (all Xilinx employees) in identifying and solving the issues that the ZC702 had in relation with the Ethernet interface when Open Virtualization was used as a TEE and L2 cache was enabled. Counting on remote access to the board made things much easier.

7.1.1.5 Compilation and Execution Workflow

Open Virtualization TEE is compiled as a unique binary where the commodity OS, the hypervisor and the secure microkernel are wrapped. While this is a good method to deploy the environment in the end product, it slows down development. Indeed, each time that the a TEE component is modified, the whole TEE needs to be recompiled, and the binary regenerated. The process requires to (i) compile the kernel in commodity OS and Open Virtualization's TEE, (ii) create the binary, (iii) copy it to the SD card, (iv) move it physically to the ZC702, (v) start the board and stop the boot sequence, and (vi) boot the binary at the address where the secure world starts. By default this address is 0x3c000000[16].

Note that some of these steps can be shortened: it is possible to load the binary using *JTAG*, and it is possible to script u-boot so that the boot sequence is stopped and the right address loaded in memory. We have also created a set of scripts to speed up the compilation process. Some of the most relevant shortcuts include avoiding re-compiling unnecessary parts of the commodity OS, compiling only the necessary parts of Open Virtualization (e.g., compiling OpenSSL only one time, unless changes to OpenSSL are made), and the mentioned u-boot scripts to directly boot to the secure world. However, the fact that there is no support for fast image burning on specific parts of the Open Virtualization software at a lower granularity slows the development process down. We have tried to implement a solution for this problem, but the dependencies between the TEE and the drivers supporting it in the compilation process makes the decoupling difficult. Indeed, it would entail a whole redesign of the components, which we have not prioritized given that it did not have an impact neither in our research and implementation of split-enforcement, nor for the community, given the contribution impediments that we are about to describe.

[16]This address depends on the amount of RAM assigned to the secure world. By default, Open Virtualizations allocates 64MB of secure RAM. The ZC702 has 1GB of RAM memory (0x40000000). If 64MB (0x0400000) are assigned to the secure world, then 960MB (0x40000000) are assigned to the non-secure world. Non-secure world's RAM starts at 0x00000000, thus 0x40000000 - 0x04000000 = 0x3c000000, where secure world RAM starts. Secure world RAM ends at 0x40000000.

7.1.1.6 Licensing

Open Virtualization was first released under GPL v2.0 between 2011 and 2012. Since then, Sierraware has maintained an open source distribution of it[17]. However, their focus has been in their commercial distribution. The main issue with this approach is the risk of contaminating one of the versions with the other. This is specially significant when it comes to IP blocks and other licensed parts of their software (e.g., TrustZone secure registers). As a consequence, releasing code for the open source version creates a big overhead since it has to be audited by different OEMs, making the process tough and slow, and preventing code distribution via repositories. Also, maintaining a commercial product implies inevitably that publicly available documentation is limited and incomplete. This has also affected the maintenance of the open source codebase, which is slow and incomplete.

For a period of time we invested time working on improving Open Virtualization. Our contributions were both in the code plane, where we identified and fixed several bugs including a number of memory leaks that we will detail later in this chapter, and in the documentation and dissemination plane, where we (i) worked with Xilinx on improving Open Virtualization's documentation for Xilinx Zynq-7000 ZC702, (ii) showed how it could be used to enforce usage control policies [131, 53], and finally (iii) refactored most of their code and organized it in git-friendly patches that we made available via a public git repository, practicing Open Virtualization's GPL license and maintaining Sierraware's copyright. The git repository containing Open Virtualization was available in github and could be directly compiled for the Xilinx ZC702 board[18] for some time. However, Sierraware failed to comply with their GPLv2.0 license and filed a DMCA takedown[19] against us on the grounds that the code was not GPL. Given our academic position, we did not respond to the DMCA, and took the repository down. At the time of this writing the repository is still not available, but Open Virtualization can still be downloaded from `www.openvirtualization.com` under an apparent GPLv2.0 license. After filing the DMCA however, target platform drivers, including the ones for the Zynq ZC702, have been removed from the available codebase.

7.1.2 Summary

All in all, Open Virtualization has helped us experimenting with TrustZone, and most importantly, understanding how a TrustZone TEE can be used to support run-time security primitives (e.g,. usage control). The improvements that we have made to Open Virtualization have also allowed us to understand the ARM instructions supporting TrustZone, the SMC calls, and ARM assembler in general. This has given us the knowledge to reason about how TrustZone use cases can be stretched, and cover much more than the ones defined by Global Platform. Moreover, it has allowed us to abstract the functionality of a TEE and come up with a proposal for a generic TrustZone driver for the Linux kernel, which we will describe in the next section.

Had it been today, we would have most probably chosen TOPPERS SafeG to provide the

[17] `http://openvirtualization.org/download.html`
[18] `https://github.com/javigon/OpenVirtualization`
[19] `http://www.dmca.com`

TEE supporting the trusted area. The overall design and the use cases that it covers align much better with the ones required to implement split-enforcement than Open Virtualization does. If we had to use a framework to support traditional Global Platform use cases we would most probably use Nvidia TLK or Linaro's OP-TEE. Both are very well organized code-wise, present clean user-space interfaces, and address the deficiencies that we have pointed out in Open Virtualization. The time that we have employed analyzing different TrustZone TEEs allowed us to define a generic TrustZone interface that does not introduce policy in terms of TEE use cases. In the next section we present a proposal for a generic TrustZone driver for the Linux kernel. Its goal is to support the different TrustZone TEEs, so that both kernel submodules and untrusted (user space) applications can use them without requiring a LKM nor TEE-specific kernel patches. In this way, developers could move to a different TEE that better suits their use cases, minimizing the engineering overhead of porting their code to a new framework.

7.2 Generic TrustZone Driver for Linux Kernel

In TrustZone, the TEE supporting the trusted area executes in a different memory space than the untrusted area. As a result we need a way for the two areas to communicate. This communication is necessary to support Global Platform's API specification. In fact, it is necessary to support any framework using TrustZone. One example is Open Asymmetric Multi Processing (OpenAMP)[20], which provides OS support for Asymmetric Multiprocessing (AMP) systems [137]. OpenAMP targets different environments such as commodity OSs (e.g., Linux, FreeBSD), RTOSs (e.g., FreeRTOS, Nucleus), and bare-metal environments; and supports the two main approaches to leverage AMP: supervised AMP by means of a hypervisor, and unsupervised AMP by means of modification in each environment. To achieve this, OpenAMP requires an initial piece of share memory that is used for communication. In an environment where TrustZone is used to support such as framework - probably as a thin hypervisor -, the creation of this shared memory region has to be supported by TrustZone primitives.

Although the communication between the secure and the non-secure areas can be implemented in different ways, TrustZone's design requires that calls to the secure monitor (SMC) from the non-secure world stem at least from the privilege mode that answers to kernel space. This corresponds to Privilege Level 1 (PL1) in ARMv7 [25], and Exception Level 1 (EL1) in ARMv8 [26] (Figure 2.6). In other words, this means that switching to TrustZone's secure world requires support from the kernel.

7.2.1 Design Space

In Linux-based OSs there are two ways of implementing this support: through a loadable kernel module (LKM), or through a Linux kernel submodule that is compiled within the kernel binary image. From a security perspective, these two approaches are equivalent; a

[20] `https://github.com/OpenAMP/open-amp`

LKM executes in kernel space with full kernel privileges and has access to all kernel APIs. This means that a LKM can provide the same functionality as a kernel submodule, with a minimal performance penalty[21]. This also means that a malicious LKM installed at run-time can damage a running system, as much as if it had been compiled in the main kernel binary.

From a design perspective however, there is a fundamental difference: LKMs are not part of the mainline kernel[22], which prevents them from defining standard kernel APIs. While this is not an issue for user space applications using a LKM, since they can access it through the interface that the LKM itself exposes to them, it hinders access from kernel space. A non-upstreamed kernel API is unknown to other kernel components - be it LKMs, submodules, or core routines -, and even if known, LKMs are optionally loaded at run-time, which means that the kernel should be compiled with support for a potentially large number of LKMs. Since hardware security extensions such as TrustZone are hardware-specific, such an LKM would be used to implement platform-specific drivers.

A solution here would be the systematic use of $ifdef$ as a design pattern to support the different platform-specific LKM TrustZone drivers. However, such programming constructs are strongly discouraged in the kernel community [77, 193], because the use of $ifdef$ does not provide platform-independent code. In general, the use of the preprocessor is best kept to a bare minimum, since it complicates the code, thus making it more difficult to understand, maintain, and debug. Also, note that this practice cannot scale, and would make the Linux kernel larger and unsustainable. LKMs are normally used as attachable kernel components[23] (e.g., device drivers, filesystems, system calls) that implement a concrete task using a specific part of the kernel; as a norm, a LKM uses kernel functions, not viceversa. A security kernel component on the other hand is by definition orthogonal to the kernel, extending to the different parts using it. Thus, even when designed as an optional modular component, its API needs to be well known and standard, so that it can be used by mainline kernel submodules.

On the contrary, if the driver that allows communicating with the TrustZone TEE is implemented as a kernel submodule, it opens the possibility to upstream it, i.e., incorporate it to the mainline Linux kernel. In this case, kernel submodules would be more likely to use it to implement secure kernel internal operations. When upstreamed, a kernel submodule defines a kernel internal API that is prone to becoming a standard in the community. Besides, since this interface is only exposed to the kernel, it can change with time without breaking user space applications; user space APIs, such as Global Platform's client API [66] for TrustZone, can then be implemented using this internal kernel interface.

Examples of kernel submodules that would benefit from a TrustZone interface in the kernel include the kernel's integrity subsystem (IMA/EVM)[24] [75, 104], and the kernel key man-

[21] There is one performance disadvantage of using LKMs that is commonly referred to as *fragmentation penalty*. The base kernel is typically unpacked into contiguous real memory by the setup routines which avoids fragmenting the base kernel code in memory. Once the system is in a state where KLMs may be inserted, it is probable that any new kernel code insertion will cause the kernel to become fragmented, thereby introducing a performance penalty [110]. There is however no conclusive analytical or experimental study showing the real performance impact due to kernel memory fragmentation when loading a LKM.

[22] `https://github.com/torvalds/linux`

[23] `http://www.tldp.org/HOWTO/Module-HOWTO/x73.html`

[24] `http://linux-ima.sourceforge.net`

agement subsystem [98, 101] (also known as kernel keyring). These two kernel subsystems already make use of the TPM kernel interface in order to implement security-critical operations when a TPM is available to the device. Using TrustZone in ARM-powered devices would be a natural extension for them. Other obvious candidate submodules that would benefit from hardware security extensions such as TrustZone include the crypto API [209], the kernel permissions subsystem (i.e., kernel capabilities)[25] [174], and the linux security module (LSM) framework [294, 295].

There are two extra advantages if support for TrustZone-based TEEs is implemented as a kernel submodule, and accepted into the mainline kernel. First, in terms of correctness, any driver accepted in the mainline kernel goes through a strict code review process[26] [193] by submodule maintainers, experts in the field, and anybody that follows the patch submissions through the *Linux Kernel Mailing List (LKML)*[27], the LKML archive[28], or LWN.net[29]. While peer-reviewed open source is not at all a synonym of bug-free code [146], we can argue that a mainline driver gets more attention - hence, more feedback - than one designed, implemented, and used by a single organization[30]. Here, formal verification would be a better solution to prevent indeterministic behavior caused by programming errors (though not algorithmic ones). However, as we discussed in Section 6.3.3, commodity OSs such as Linux, as well as the development process driving them, are far from being formally verifiable.

Second, in terms of adoption, a patch accepted in the mainline Linux kernel automatically reaches millions of devices. Note that Linux-powered server demand has increased year after year, reaching 28.5% of all server revenue in 2014[31]. It is estimated that between 60% and 80% of all public servers are powered by Linux[32,33]. Large data centers such as the ones used by Facebook, Amazon, or Google are also powered by Linux. We expect these large data-centers to move progressively towards ARMv8 (at least partially) in order to be more power efficient. In the area of mobile devices, as mentioned before (Section 6.3.3), Android is powered by a Linux-based kernel, and iOS by a UNIX-based one. Together, they have a market share of 96%[34].

Today, all frameworks leveraging a TrustZone TEE implement their own complete driver to switch worlds. The main consequence is that the development of TEEs enabled by TrustZone has become very fragmented, forcing service providers using TrustZone to either depend on LKMs to add support for these TEEs, or patch their kernels with specific interfaces for the TEE framework(s) they want to support. While applications can still move among different TEE frameworks - providing that they implement Global Platform's APIs -, this has prevented kernel submodules from enabling hardware-supported secure services (e.g., trusted storage, key management, attestation) in ARM processors. More importantly, this

[25] http://man7.org/linux/man-pages/man7/capabilities.7.html
[26] `https://www.kernel.org/doc/Documentation/SubmittingPatches`
[27] `http://vger.kernel.org/vger-lists.html`
[28] `https://lkml.org`
[29] `https://lwn.net/Kernel/`
[30] Big companies, such as Apple, Microsoft, or Amazon are in the end one organization, with more limited resources than a distributed community.
[31] `http://www.idc.com/getdoc.jsp?containerId=prUS24704714`
[32] `https://secure1.securityspace.com/s_survey/data/201410/index.html`
[33] `https://secure1.securityspace.com/s_survey/data/man.201311/apacheos.html`
[34] `http://www.idc.com/prodserv/smartphone-os-market-share.jsp`

has prevented the adoption of TrustZone in the open source community. A proof of this is the lack of TrustZone support in the mainline Linux kernel. As a mean to give OS support for run-time security, part of our contribution has focused on providing a generic TrustZone solution for the Linux kernel. We report now on the design and implementation of our submitted proposal for a *Generic TrustZone Driver* to the Linux kernel community[35,36].

7.2.2 Design and Implementation

The main challenge when designing a generic TrustZone interface for the Linux kernel is not introducing policy for TrustZone use cases. In Chapter 2, we presented TrustZone as a technology (Section 2.4), and discussed traditional use cases (e.g., banking, DRM). Since then, and specially in Chapter 6 where we presented split-enforcement, we have introduced new use cases that depend on a tighter collaboration between the trusted and the untrusted areas. One extreme use case that is the base for split-enforcement is enforcing usage control policies. Our design goal for the generic TrustZone driver is to cover all these use cases, giving developers the flexibility to freely design their specific drivers and libraries, and introduce policy there if necessary. In order to satisfy this requirement, we propose that the TrustZone driver is composed of two parts: one generic part defining the interface, and different specific parts that implement this interface for specific TEE frameworks, targeting each their different use cases. This design pattern is widely used in the Linux kernel. Examples include several device drivers such as I2C or TPM, the VFS layer, or the LSM framework. Figure 7.1 depicts the architecture for the generic TrustZone driver.

The generic part is in charge of three important tasks: (i) it defines a common kernel internal interface so that other kernel components can use it, (ii) it exposes a management layer to user space trough *sysfs*[37] [207], and finally (iii) it connects to the different specific TEE drivers. In order not to introduce policy, we propose a simple *open/close*, *read/write* interface. As we have already mentioned, it is important to understand that a TrustZone TEE only provides the secure world abstraction; it does not implement a specification for the secure tasks, as it is the case in TCG's TPM. In this way, a kernel submodule, or a user space library making use of the generic TrustZone driver need to know which specification is implemented in the secure area. In the case of user space applications, Global Platform's APIs cover the traditional TrustZone use cases, and will most probably be the ones demanded by service providers in the short term. If a new specification appears, such a generic interface would most probably allow to implement it. If not, since internal kernel interfaces are not tied to legacy code, the interface can change, and in-kernel use of it can be adapted. From the kernel point of view, such a generic interface allows to define run-time security primitives to support TrustZone-enabled trusted services. We will discuss this in detail in next Chapter (Chapter 8).

The specific parts are vendor-specific, and they implement the generic TrustZone interface with the concrete mechanisms to communicate with the TEE they represent. While con-

[35] `http://lwn.net/Articles/623380/`

[36] `http://lkml.iu.edu/hypermail/linux/kernel/1411.3/04305.html`

[37] Note that this is an implementation decision. Other alternatives such as *dubugfs*, or *ioctl* can be easily implemented - each for its different purpose -, if necessary.

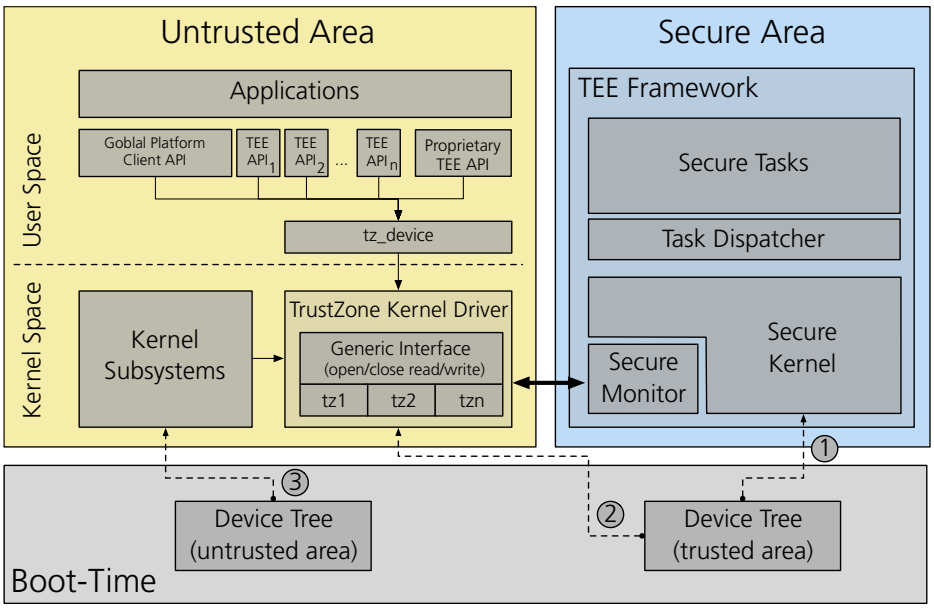


Figure 7.1: Generic TrustZone Driver Architecture. Boot-time information is include to depict the concept of the *Secure Device Tree* which corresponds to Device Tree (trusted area).

ceptually, these specific drivers would only need to implement different parametrized calls to the secure monitor, our experience porting Open Virtualization's driver to work with the proposed generic interface exposed two engineering challenges:

1. **Give peripherals the correct base address.** In TrustZone, any device can potentially be mapped to the secure world at boot time. More importantly, this mapping can be dynamic, meaning that it might be changed at run-time. The set of devices that can be assigned to the secure world, as well as the set of mechanisms to do so at boot-time are platform-specific. The possibility to re-map peripherals to a different world at run-time is also platform-specific. From the kernel perspective, this means that the memory for each device needs to be mapped to a different address space depending on the world the device is assigned to. Typically, the kernel maps devices to a I/O memory region and manages this mapping without the intervention of the device. The kernel knows the base address for each device, and the offsets to communicate with it are implemented according to the device's specification. In TrustZone, this is the case for devices assigned to the non-secure world. However, when a device is assigned to the secure world, the kernel is not the one mapping it to a I/O memory region. Indeed, the kernel cannot access the device since accesses to ARM's Advanced Peripheral Bus (APB) (Section 2.4) bus stem from the non-secure world. This makes the access to be rejected; the AXI-to-APB bridge guarantees that secure peripherals cannot be accessed from the non-secure world, which includes the kernel in the untrusted area. Still, the kernel is the one orchestrating the communication with the secure world, and therefore it needs to know the base address for all peripherals - also secure peripherals -, in order to (i) access non-secure peripherals directly, and (ii) forward accesses to the

right secure peripheral in the secure world.

Generalizing how base addresses are assigned to secure peripherals is a challenge. Not only does each TEE framework implement its own address space (including I/O memory) in a different manner, but also each target platform defines its own set of registers to access secure peripherals. These registers are normally not publicly available, and it requires a NDA to access them. Thus, hard-coding these secure registers in a per-platform basis in the kernel is not a viable solution. A good way of approaching this problem is assigning the base address of a secure peripheral to its physical address. The TEE maintains then a mapping table storing physical addresses and secure registers. In this way, when the kernel gets an I/O request to a secure peripheral, it forwards the request to the secure world. Since the base address is the physical address of the device, the secure world can obtain the secure address (using the mapping table) and serve the request. Note that since physical addresses are unique, there is no need to maintain an extra identifier for the devices. The remaining question is then, how to dynamically assign the base address depending on the world the peripheral is assigned to? This defines the next engineering challenge:

The TEE frameworks that we know of (Section 2.4) approach the problem of assigning base addresses by directly hard-coding these to physical addresses (or equivalent unique ID) for secure peripherals. This is done by patching the kernel. Note that this patch is independent from the actual TrustZone driver, and it is a necessary step even if the driver is implemented as a kernel submodule. After applying the patch, the kernel is TEE-enabled, meaning that the TrustZone driver is operational[38]. There are three problems with this approach: First, from an operational perspective, this solution forces a static assignment of peripherals to the trusted and untrusted areas. Since the kernel is patched to hard-code base addresses in the case of secure peripherals, these are not properly initialized (probed in kernel terms) at boot-time; among other things, a I/O memory region is not assigned to the device. What is more, this prevents peripherals from being initialized at any other time, and from being re-assigned to the non-secure world at run-time. As a consequence, peripherals are statically assigned to the trusted and untrusted areas. This introduces policy in terms of secure peripheral usage, and prevents us to satisfy one of the split-enforcement requirements: dynamic locking/unlocking of peripherals by assigning them to the secure and non-secure worlds.

Second, from an engineering perspective, hard-coding addresses normally signals a bad design. In this specific case, it prevents to use a kernel with several TEEs; the kernel is patched with a hard-coded set of registers that are platform- and TEE-specific. It can be argued that each target platform can hard-code its own set of registers in target-specific files when TEE support is enabled. However, if the TEE changes, the patches enabling the previous TEE would need to be rolled-back, and the new TEE-enabling patches applied. This process requires the kernel to be recompiled. In this way, we end up with a per-TEE and -platform kernel combination, where TEE-enabling code is pushed down to target drivers, thus increasing driver complexity (and forcing the massive use of $ifdef$). It is easy to see that this creates a big overhead in terms of

[38]Note that if the patch is not applied, the kernel will try to assign I/O memory to secure peripherals at boot-time. Since all accesses to the APB bus stem from the non-secure world, this will cause the AXI-to-APB bridge to reject them, which results in a kernel panic.

engineering for service providers supporting different TEEs and platforms.

Third, an extra negative consequence of this design is that new versions of the kernel need to be rebased on top of all these local patches, which also creates a significant engineering overhead. This is specially true for major kernel revisions.

To mitigate these issues we propose extending the device tree with TrustZone information. More specifically, we propose a new property for devices in the device tree that contains the TrustZone world to which they should be assigned to at boot-time. We denote this property *arm-world*. The Device Tree is a data structure for describing hardware defined by OpenFirmware (IEEE1275-1994). It allows to define nodes and properties to describe a whole hardware platform in such a way that it can be easily extended with new nodes and properties. In this way, rather than hard-coding every detail of a device into the operating system, many aspect of the hardware can be described in the device tree, which can then be passed to the operating system at boot-time. Device trees are supported in a number of architectures (e.g., ARC, ARM(64), x86, PowerPC), and have recently become mandatory in ARM architectures in all new SoCs [227]. Therefore, this extension for describing TrustZone-enabled devices is in line with our current design.

In order to support peripheral remapping at run-time, we also propose decoupling device initialization and termination routines from specific system events - boot and halt processes respectively[39] -, so that a device can be initialized and terminated at any time at run-time. This permits the kernel to manage I/O memory correctly, while peripherals can be dynamically assigned to the trusted and untrusted areas at run-time. Since physical addresses are already present in the device tree, no extra information is necessary. In the case that a TEE requires the use of a different identifier, then the device tree can easily be extended with this identifier.

In this way, base addresses are not hard-coded in the kernel. It is thus possible to dynamically assign peripherals to the trusted and untrusted areas at run-time. Since base addresses are loaded from the device tree, kernel device drivers supporting TrustZone become simpler and more portable. This design also makes it simpler to maintain an updated version of the kernel, as the hardware information in the device tree is orthogonal to the kernel version. Additional local patches for TEE support are not necessary[40]. With this design, we tackle all the issues pointed out with regards to secure peripherals and base addresses.

2. **Introduce secure read/write.** Once base addresses are generalized, the kernel still needs to know how to forward I/O requests to secure peripherals via the secure world. This is typically done by introducing two new instructions: *secure_read* and *secure_write*, which are the secure equivalents to the different kernel versions of *read/write* (e.g., *ioread32/iowrite32*, *readl/writel*). However, since the interpretation of *secure_read* / *secure_write* is framework-specific[41], existing TrustZone TEE frame-

[39] Note that some drivers already do this as a way to better organize their code.

[40] Note that support for reading TrustZone properties need to be added to all device drivers supporting TrustZone. However, this is a one time patch to be added to the mainline kernel, not an evolving patch that needs to be rebased for every kernel revision.

[41] These instructions are implemented as commands sent to the secure monitor, and interpreted within the TEE.

works have also implemented this as part of the patches enabling their TEE for a given kernel. This is an inconvenient for designing a hardware-agnostic TrustZone driver; if TEE frameworks do not have support for these instructions directly in the kernel, a common interface is not possible.

The solution follows the same design as the one presented for generalizing base addresses: adding this information to the device tree. Since a TEE is in the end a framework enabling a hardware feature (TrustZone), it makes sense to describe its characteristics in the device tree. Representing the TEE as a device tree node enables then making a device TrustZone-aware. Such TEE node can contain information about which peripherals can be secured in a specific target platform can support, which run-time security primitives are implemented in the TEE (i.e., *secure_read* and *secure_write*), or which TEE specifications it supports (e.g., Global Platform). What is included in the TEE node is decided by each specific TEE framework. Here, one idea that we have explored (but not yet fully implemented) is allowing the device tree to be modified by the TEE before it is passed to the kernel at boot-time. In this way, the TEE would have control over the TEE node in the device tree, thus over the information shared with the kernel.

One of the security requirements for the definition of a TEE is that it always boots up before the untrusted software stack; this is the case for TrustZone's secure world in relation to the non-secure world. We think about this as a *Secure Device Tree*. It is however not clear for us whether the secure device tree should be implemented as a TEE node, or as an independent device tree also passed to the commodity OS at boot-time. This is part of our ongoing research. In fact, in the process of writing this thesis, Greg Bellows from Linaro started this same discussion in the Linux kernel device tree mailing list[42]. Our current intuition is that having two separated device trees is more difficult to manage, even when we already have two software stacks. One way to handle this would be to define a TEE node in the original device tree that is interpreted at boot-time by the secure world. Here, the TEE framework could use this information to generate (an complete) a secure device tree. This would be useful if an unmodified Linux kernel is used in the secure world[43]. In this way, specific secure world information would not be leaked to the non-secure world. The non-secure world could still use the TEE node in the original device tree to configure the appropriate base addresses for secure peripherals, be aware of the available run-time security primitives, implemented specification(s), etc. We are at the time of this writing defining which is the best way to collaborate with Linaro in order to find an adequate solution for this issue.

Even if such a secure device tree becomes a standard practice, we believe that *secure_read*/*secure_write* should become a standard instructions in the Linux kernel (i.e., be given a magic number[44]), and a standard commands among TEE frameworks. This would be a step towards a TEE standardization. The fact that other architectures besides ARM are both beginning to offer their own set of security extensions (e.g., Intel

[42] `http://www.spinics.net/lists/devicetree-spec/msg00074.html`

[43] This is not a use case we contemplate given our requirement of having a small TCB, but it is a possibility for others.

[44] `https://www.kernel.org/doc/Documentation/magic-number.txt`

SGX) enabling a TEE, and making use of the device tree to describe their hardware characteristics[45,46] add straight to our argument.

Our proposal for adding TEE support require minimal changes to the kernel. These changes are necessary for each device that can be mapped to the trusted area, but the functions implementing this support can be abstracted and generalized. In this way, adding support for new devices becomes a simple task. Based on our experience porting devices for the Zynq ZC702 (e.g., clocks, timers, AXI devices), the process of adding TrustZone support entails: reading the *arm-world* property from the device when it is being initialized (probed), assigning the base address depending on this property, and finally forwarding the I/O request to the right function (i.e., *read/write* or *secure_read/secure_write*) also depending on this value. It is also necessary to decouple the functions for allocating, assigning and freeing I/O memory regions from the initialization and termination routines so that they can be managed on request at run-time. All these modifications entail in average modifying less than 15 LOC per device.

It is important to highlight that our proposal solves the issues that we have described in the beginning of this chapter without introducing policy, which was one of our primary requirements. Now that we have explained in detail the specific mechanisms to add kernel support for TEEs, it is easy to see that the TEE supporting the trusted area can be switched without forcing changes into the kernel using it. Not only does this allows for service providers to support different TEEs and platforms, but it also gives the possibility to exchange the TEE at run-time, or even to support several TEEs running simultaneously. An example of the latter would be having TrustZone and a SE being accessed separately from the untrusted area to support different trusted services. Our solution would also support new revisions of hardware security extensions without requiring more modifications than the definition of a new TEE node in the device tree. We have seen this problem in the latest revisions to the Intel SGX specification [159, 162], or the porting from TPM 1.2 to 2.0. In ARM circles, a revision of the TrustZone security extensions has been discussed for years. When such a revision is introduced, TEE frameworks and their support in the Linux kernel (as well as in other OSs) should be ready for it.

7.2.3 Open Virtualization Driver

In order to test the viability of our generic TrustZone driver proposal, we have ported Open Virtualization to work with it. Sierraware implemented Open Virtualization support in form of (A) a kernel patch to enable communication with their TEE, and (B) a LKM implementing the Open Virtualization TEE kernel driver (i.e., *otz_main_client*). In order to integrate these two components in the generic TrustZone driver design we have just described, we replicate the functionality of the first patch (A) in the generic TrustZone support we have added in the kernel - moving the description of the specific TEE and platform to the device tree -, and we (ii) port Open Virtualization's TEE LKM driver (B) as a TEE-specific subdriver

[45] `http://lwn.net/Articles/429318/`

[46] `https://www.kernel.org/doc/Documentation/devicetree/usage-model.txt`

that implements the generic TrustZone interface with Open Virtualizations concrete SMC operations and routines.

We use Xilinx's Zynq-7000 ZC702[47] as our experimentation platform, and we use the ZC702's device driver as a base to implement the dynamic TEE support described in previous section. Here, we only need to add a TEE node where we specify (i) the magic numbers for the commands corresponding to *secure_read* and *secure_write*, (ii) the world to which each device is assigned to at boot-time through the *arm-world* property, and (iii) the base address for secure peripherals, which in the case of Open Virtualization correspond to each device's physical address.

In the process of porting Open Virtualization's driver, we had to refactor most of its code so that core functionality could serve both user space applications and kernel submodules. In the original driver, the *ioctl* operations used to communicate with user space embedded part of the core logic regarding SMC calls. These were mixed with operations typically used for communicating with user space (e.g., *copy_from_user* / *copy_to_user*, which are used to translate addresses from user to kernel space and viceversa). As part of our re-design, we refactored the driver to decouple the interfaces (now user and kernel) from the core functionality. Note that ideally, all TEE drivers would implement user space specifications on top of the kernel internal interface. In this way, we provide a user space interface to allow user space applications use the TrustZone driver through *ioctl*[48]. We denote this interface *tz_device*. Indeed, as a proof of concept we have ported part of Global Platform's Client API using the tz_device. However, since Open Virtualization also supports its own specification besides Global Platform's APIs, we maintain their *ioctl* interface. As we have defended during this whole chapter, we do not want to introduce any form of policy in the generic TrustZone driver; TEE frameworks should be able to provide as many extra interfaces as they want, as long as they implement the common generic interface.

Finally, while measuring the performance of our driver implementation (we cover the evaluation part in detail in Part III), we uncovered two memory leaks in Sierraware's implementation. Both memory leaks (of 4KB and 32KB) were due to bad memory management. The 32KB leak could be found using *kmemleak* [76] since memory ended up being dereferenced; the leak was caused by freeing a structure that contained a memory pointer which had not been previously freed. The 4KB one, on the other hand, was a consequence of storing a bad memory reference, which prevented an internal list from being freed. This was in the end caused by bad type assignation. Since the amount of memory lost per call to the secure world was that small, the leak was very difficult to find and required a manual analysis of the code. Note that the leak did not leave any dereferenced memory, which prevented *kmemleak* to find it. In general, detecting such small leaks in production when the TEE driver is exposed to TEE typical use cases is almost impossible; the secure world is not used that intensively, and a few KB lost in kernel memory are not an issue in today's devices. As we will see in Part III, the memory leak was exposed because the kernel run out of memory when forward-

[47] We will describe in detail our experimental setup using Xilinx's Zynq-7000 ZC702, as well as the reasons behind this decision in Section 11.1.

[48] Since in this case we use the user space interface to expose kernel operations, *ioctl* is a good design option. In general, if a device driver is used in the kernel *ioctl* is accepted, even though it is a highly overloaded interface. Note that the fact that it provides atomicity is a big plus when used in compiled user space applications.

ing a secure I/O request to the trusted area per system call, while using a heavy workload (compiling the Linux Kernel). This is one example use case of how we have tested the limits of TrustZone. We have reported the issue to the interested parts, as well as a patch fixing it. The memory leak is now fixed for the open source and commercial version of Sierraware's TEE framework.

7.3 Driver Status

At the time of this writing we have a working *Generic TrustZone driver*, where both the interface, and Open Virtualization's driver are implemented, and we have submitted a first set of patches to the Linux Kernel[49]. We have received feedback from the Linux community and we are working on a revision of these patches to submit them again[50]. Apart from proposing the generic TrustZone driver, we propose a new subsystem for secure hardware (*drivers/sechw*). The idea is to have a subsystem that is only dedicated to secure hardware and hardware security extensions, and promote its support in the mainline Linux kernel. The TPM driver that today is implemented as a char device (*drivers/char*) would be a good candidate to be ported if the new new subsystem is accepted.

Development is taking place in a public git repository located in github[51]. Here, we have two repositories. The first one is a fork of Xilinx's official Linux Tree (linux-xlnx) where we implemented the first version of the driver for Xilinx's Zynq Base Targeted Reference Design (TRD) v.14.5[52], which uses Linux kernel 3.8. This implementation allowed us to identify and solve the problems we have discussed throughout this chapter. We maintain it as a legacy implementation of the driver. The second repository is a fork of Linus Torvalds mainline Linux tree (linux). We had to refactor most the the code concerning base addresses in Zynq since the mechanisms to map physical memory for I/O devices has changed substantially from version 3.8 to 3.18 (current version at the time of this writing). We have moved the main development to this repository (linux) in order to continue the support for the generic TrustZone driver in future kernel releases. We encourage TrustZone developers to contribute to these repositories; we intend to continue using these repositories for primary development after the driver is accepted in the mainline Linux.

Since the submission of this patch we have received many positive feedback from part of the open source community. From developers and organizations that share the interest on counting on such a generic TrustZone driver in the mainline Linux, to technology reporters[54], and in general people with an interest in this effort be;ing made. We have presented our advances at several venues, including the current latest Linux Con Europe in Düsseldorf[55] [129], and we are at the moment working on a publication dedicated to the generic TrustZone driver.

[49] `http://lwn.net/Articles/623380/`

[50] The process of writing this thesis has overlapped with this effort, but we will come back to it as soon as this thesis is submitted.

[51] `https://github.com/TrustZoneGenericDriver`

[52] `http://www.wiki.xilinx.com/Zynq+Base+TRD+14.5`

[53] `http://cloc.sourceforge.net`

[54] `http://www.phoronix.com/scan.php?page=news_item&px=MTg1MDU`

[55] `http://www.linuxplumbersconf.org/2014/ocw/users/2781`

Subsystem	Lines of Code Modified
Generic TrustZone Driver	+ 519
Sierraware OTZ driver	+ 1511, - 423
Total	+ 2030, - 423

Table 7.1: Lines of coded (LOC) modified in the current version of the generic TrustZone driver for the Linux kernel. Since Open Virtualization integration with the TrustZone driver is based on Sierraware's original kernel driver, we only count the modifications made to it. These include both code re-factorization and bug fixes. The size of the driver including Open Virtualization's driver is 4030 LOC. Open Virtualization alone is then 3511 LOC (4030 - 519) LOC. Note that the LOC include header files. We use *cloc*[53] to count LOC, and we omit comments or blank lines in our numbers.

We intend to continue bringing attention to the driver in form of presentations, LWN articles, and research publications. We also intend to present advances in the drivers in future Linux events, as well as in other security conferences. Ideally, we would like to drive the TrustZone subsystem, as well as to bring more attention to security in the Linux community. Also, the fact that organizations such as Linaro, which have a big voice in the Linux community, have expressed their interest in our work[56] motivates us to continue working and contributing with our efforts.

7.4 Conclusion

In this chapter we have presented two contributions that enable operating system support for a TrustZone TEE in Linux-based systems: the contribution to an existing TEE framework, and the design and implementation of a generic TrustZone driver for the Linux kernel.

In terms of the TEE framework, we have first presented different types of TEEs and compared them. We then have presented the TEE that we have used for our experimentation: Open Virtualization. Here, we have provided a full analysis of it in terms of its design, interfaces, and code quality. Moreover, we have presented our contributions to it in each of these areas. The most relevant contributions to Open Virtualization include: (i) updating OpenSSL to a version not containing *heartbleed*, and changing the used set of operations to avoid deprecated interfaces; (ii) refactoring and cleaning code to allow for a decoupling between user and kernel space support; (iii) creating a set of scripts to allow for a better compilation and execution workflow; (iv) organizing and documenting patches to make them understandable and git-friendly; and (v) uncovering and fixing two memory leaks in Open Virtualization's kernel driver. Apart from this, we have also ported an unittest framework, pointed to bad design for passing secure parameters, identified synchronization problems caused by interrupt disabling, and documented the framework at different levels, specially in which refers to its use in the Xilinx Zynq-7000 ZC702. Despite the impediments caused by Sierraware, that have prevented us from publicly sharing these contributions in a git repository under GPL

[56] As mentioned before, Linaro is behind OP-TEE, and is collaborating with us in developing the new patch submission to the mainline Linux kernel.

v2.0, we hope that reporting them here can help other researchers and developers to overcome similar problems.

In terms of support for Linux-based systems, we have reported on the design and implementation of a generic TrustZone driver for the Linux kernel. First, we argue for the need of TrustZone support in the Linux kernel in the form of a mainline kernel submodule. Then we present our design for a generic *open/close, read/write* interface that allows user and kernel space objects to make use of a TEE without introducing policy. In order to test it, we port Open Virtualization's driver to work with it. In the process of doing so, we identify two engineering challenges that we needed to overcome: assigning base addresses for secure peripherals, and generalizing *secure_read* and *secure_write* operations in the Linux kernel. In order to solve them, we propose the use of a TEE node in the device tree that can generate a *Secure Device Tree* in the secure are. The complete design for how such secure device tree should be supported by the device tree community is still being discussed with other kernel developers that have encountered the same issues. Development can be followed in github[57].

This generic TrustZone driver will be the base to enable run-time security primitives in the untrusted area. In this way, this driver is a necessary component for untrusted applications to make use of trusted services.

[57] `https://github.com/TrustZoneGenericDriver`

Chapter 8

Trusted Cell

Now that we have a TrustZone TEE framework that enables the *Trusted Area*, and a driver to communicate with it from the *Untrusted Area*, we can focus on (i) the actual implementation of *Split-Enforcement*, and (ii) the *Run-Time Security Primitives* that it can leverage. The ultimate goal is to utilize these primitives to enable *Trusted Services* in *Commodity OSs*.

It is relevant to mention that as we began to understand the technical barriers to implement usage control policies (which was our first attempt to define what we now conceive as run-time security primitives), our research moved towards hardware security extensions and TEEs. As we understood the problems associated with using a TEE, our research moved towards providing OS support for a TEE.

In this section, we close the loop around our early research ideas on run-time security, and present the design and implementation of a *Trusted Cell*, i.e., a distributed framework that leverages the capabilities of a given TEE to provide trusted services. A Trusted Cell is formed by two main sub-frameworks: a trusted sub-framework to support trusted modules in the secure area (i.e., *TEE Trusted Cell*), and an untrusted sub-framework that allows the use of trusted services from within the commodity OS (i.e., *REE Trusted Cell*), which includes untrusted applications and kernel components. Trusted services at the OS level (kernel) are leveraged by the mentioned run-time security primitives.

We use a Trusted Cell to implement split-enforcement, introduced in Chapter 6. Here, we implement two trusted services that we consider to be pivotal for run-time security: a *Reference Monitor* to support usage c ontrol, and a *Trusted Storage* solution. Our Trusted Cell is the first design and implementation of the *Trusted Cells* [10] vision.

While we will describe in detail how we implement a reference monitor and a trusted storage solution, the core contribution in this chapter is at the architectural level. The central question we want to answer is: How can we design and implement trusted services using run-time security primitives enabled by a TEE, in such a way that they comply with split-enforcement? What is more, how do we integrate these trusted services into commodity OSs that have traditionally being founded on strong trust assumptions? Most of the work done in reference monitors (Section 3.3) has focused on the policy engine: inputs, scope, usage

decision generation, etc. Most of the work done in trusted storage (Section 3.2) has focused on concrete use cases: encryption scheme, reutilization of an untrusted infrastructure, enabling trusted storage services, etc. With Trusted Cell, we focus on providing an architecture that supports run-time security primitives to enable trusted services. Such an architecture was in away already envisioned in formal usage control models such as UCON$_{ABC}$ [221]; we provide the first design and implementation.

8.1 Architecture

The architecture for the Trusted Cell is common to all trusted services. Our design follows the idea that these trusted services can be leveraged by a set of *Trusted Modules*, each implementing a specific behavior. One trusted service can make use of several trusted modules; a trusted module can be used by several trusted services. A central design decision here is that the Trusted Cell is modular in such a way that new modules can be added at any time to provide new trusted services. While the process of loading and attesting trusted modules in a TrustZone TEE is out of the scope of this thesis, the Trusted Cell supports it by design. Note that this design decision extends to the TEE OS support; the generic TrustZone driver proposed in Section 7.2 does not introduce usage policy, and moves the responsibility of knowing the specific secure task to be executed out of the driver, into the component using the TEE.

A Trusted Cell is as a layer that sits on top of the generic TrustZone driver in the untrusted area, and on top of secure tasks in the trusted area in order to define trusted services and make them available to the commodity OS. Put differently, it allows the secure area to expose different communication interfaces (e.g., specifications, protocols) to the commodity OS. Thus, we introduce a division between TEE Trusted Cell and REE Trusted Cell: the TEE Trusted Cell provides the framework for executing the trusted modules leveraging trusted services in the secure area, while the REE Trusted Cell uses the generic TrustZone driver to add OS support for these trusted services.

Apart from the flexibility that this design provides in terms of not imposing a fixed specification[1], it allows service providers to easily port their applications to execute on a TEE so that they can leverage the hardware security extensions present in a given device (Chapter 2), while reusing their communication protocols - which might be proprietary. Making our attack model explicit, the TEE Trusted Cell is located in the secure area, thus it is trusted; the REE Trusted Cell is located in the untrusted area, thus it is untrusted. The challenge is then to implement split-enforcement to guarantee the integrity and protection of *Sensitive Assets* (Section 6.2).

Since TrustZone is not tamper-resistant, we complement the trusted area with a Secure Element (SE) as stated in our hypothesis (Setion 6.2). The SE is configured as a trusted peripheral and therefore only accessible from within the trusted area. Note that while the SE is not a peripheral in itself, it is connected via the APB (e.g., I2C, etc.), and therefore

[1]Refer to Section 2.2 to find an example on how imposing algorithms in the TPM 1.2 specification has forced the TCG to define TPM 2.0, which does not impose specific algorithms.

treated as such. In this way, untrusted applications make use of secure tasks to access the SE. They do so by using a run-time security primitive that exposes the SE in the secure area. Such primitive would resemble a system call to the secure area. As a consequence, if the TEE becomes unavailable, untrusted applications would be unable to communicate with the SE. What is more, since the SE is used to store encryption keys and other the secure data in the secure area, if the SE becomes unavailable, the trusted services relying on it would stop operating correctly. The TEE becoming unavailable could be a product of a failure, but also a defense mechanism against unverified software. In general, both at the SE and secure area level, we give up on availability in order to guarantee the confidentiality and integrity of the sensitive assets that are manipulated in the trusted area. This will be an important discussion when we present *Certainty Boot* in Chapter 9, since it depends strongly on the SE functionality.

In the process of designing the Trusted Cell to implement split-enforcement, we identify three problems: (i) we need an efficient way to integrate the REE Trusted Cell in the commodity OS; (ii) we need to do that in such a way that both kernel submodules and applications can make use of trusted services; and (iii) we need to identify and implement the locking mechanisms that enable split-enforcement. Problems (i) and (ii) belong in the REE Trusted Cell; problem (iii) in the TEE Trusted Cell. We describe the locking mechanisms to enable split-enforcement when we present our reference monitor prototype (Section 8.2.2), since we consider them a part of this trusted service. We now focus on the Trusted Cell architecture, which is depicted in Figure 8.1. Note that in this Figure we introduce all the components of the Trusted Cell. We will describe them throughout the rest of the chapter.

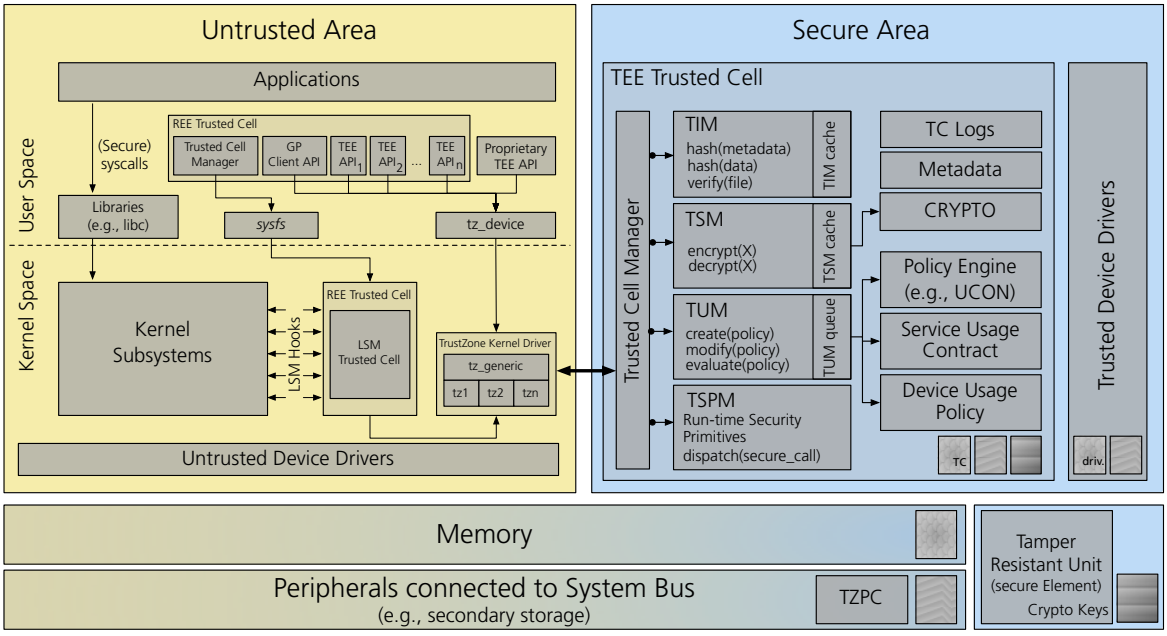


Figure 8.1: Trusted Cell Architecture: TEE Trusted Cell in the secure area (right, blue); REE Trusted Cell in the untrusted area (left, yellow).

8.1.1 REE Trusted Cell

The primary responsibility of the *REE Trusted Cell* (Figure 8.1) is to allow components in the untrusted area to use trusted services. This includes kernel submodules, and user space applications. Given their disparate run-time properties - user space is intrinsically more dynamic than kernel space -, support for them is necessarily different. One extreme in the design space is directly using the generic TrustZone driver to execute secure tasks (providing that untrusted applications know which secure tasks are available in the secure area), the other is that the Trusted Cell (in both areas) hides TrustZone logic an exposes a number of interfaces. We explore kernel and user space support for trusted services separately. Figure 8.2 depicts the REE Trusted Cell in detail.

Support for Kernel Submodules. Before we discuss specific mechanisms, it is important to understand that kernel mainline components (e.g., core kernel objects, kernel submodules) are static at run-time. The kernel main binary is compiled once and cannot be modified while it is executing. Since we want to support mainline kernel components, adding new interfaces as LKMs makes poor sense; kernel submodules have already been compiled to use a fix number of interfaces[2]. We have already discussed LKMs use cases in depth in Section 6.3, when arguing for the implementation of TrustZone kernel support as a mainline submodule. In this context, we seek a mainline module in the Linux kernel that is accessible from the rest of the kernel, so that the different kernel submodules can use it to execute trusted services. In this way, changes to the interface, or support to new interfaces would require changes in only one place.

The Linux Security Module (LSM) [294, 295] is the perfect place to implement the REE Trusted Cell in the Linux kernel. LSM is a framework that mediates access to kernel space objects in Linux. The framework consists of a set of software hooks placed inside the kernel that enable a security function to be executed before a kernel object is actually accessed. The idea is that when a security module subscribes to the LSM interface, it implements the functions linked to the hooks, and thus allows or denies access to kernel objects based on its internal state and on pre-defined policies. LSM is primarily used today in the Linux kernel for (i) extending the default Linux permissions to implement mandatory access control (MAC). The five current frameworks implementing different paradigms for MAC are: SELinux [263], AppArmor [216, 37, 107], Tomoyo [140], Smack [250], and Yama [73]. Also, the LSM framework is used to support the already mentioned (ii) Linux integrity subsystem (IMA/EVM) [75, 104], and (iii) the kernel keyring subsystem [98, 101]. Adding support to stack different LSM modules to work simultaneously has been proposed in the research community [233], and repeatedly discussed in the Linux kernel community, where up to three LSM stacking proposals by Kees Cook [100], David Howells [102], and Casey Schaufler [103] have been submitted to the LKML. However, at the time of this writing, none of these proposals (or any other) have been accepted in the mainline Linux kernel (v3.18). This affects the adoption of our solution, since most OSs built around the Linux kernel use one of the

[2]Note that we discuss standard LKM procedures and developing techniques. It is possible to implement a LKM that "hijacks" a system call an implements a different operation by finding the address of the *syscall table* and overwriting specific system calls. This is however not a realistic (nor recommended) way of implementing new interfaces in the kernel.

mentioned systems using a LSM for MAC. As an example, AppArmor is enabled by default in Ubuntu [38] and SUSE [36], and it is available in most Linux distributions; SELinux is used by default in Android [262]. Note that for this same reason IMA/EVM [75, 104] does not currently use LSM hooks for its implementation; it adds its own hooks throughout the whole kernel in order to be compatible with the use of LSMs implementing MAC. This design is only possible because IMA/EVM is the only integrity submodule in the Linux kernel. We believe that a LSM stacking solution should be accepted in the mainline, and IMA/EVM ported to use LSM standard hooks. Since we are implementing a prototype for REE Trusted Cell, which at this point we do not consider submitting to the LKML, we have prioritized good design over adoption. Exploring how to implement LSM stacking is future work.

We developed a new LSM submodule that we call *LSM Trusted Cell*, which uses the LSM framework to provide the kernel with run-time security primitives. While the LSM framework is designed to support access control, not auditing [295], this is enough when implementing split-enforcement; before a sensitive asset is used, the request is captured by the appropriate LSM hook and sent to the reference monitor in the TEE Trusted Cell. If the request is accepted by the reference monitor, the sensitive asset is unlocked; the request is denied otherwise. The reference monitor in the secure area gathers historical accesses and post-actions, therefore implementing auditing. Note that, since all sensitive assets are locked, an attacks targeting LSM hooks to bypass them bring no benefit to the attacker. We use the LSM framework because it is an established mode in the Linux kernel to add security checks before allowing access to kernel objects. It is easier to implement new hooks to the LSM framework than coming up with a new solution that might adapt better to split-enforcement. Add-ons to the LSM framework are also more likely to be accepted in the mainline kernel than a whole new framework. An extreme here would be to trigger the reference monitor for each single action being carried out by the kernel. This can be done by directly altering the system call table to issue a SMC call prior executing the corresponding kernel operation. In the ARM architecture the system call table is located at *arch/arm/kernel/entry-common.S*. We take this approach to carry out part of our experimental evaluation in Chapter 11 since it represents the worst case in terms of performance. However, identifying which assets are sensitive, and using the LSM hooks that safeguard them in order to execute the secure tasks that unlocks them not only is a better solution, with higher possibilities to becoming a standard practice, but also entails lower performance overhead[3].

Since LSM hooks extend to the whole kernel (e.g., I/O stack, network stack, capabilities subsystem), the LSM Trusted Cell represent a powerful tool to give secure hardware support to current system primitives. Security-critical operations that are typically executed by the different LSM submodules in the same address space as the rest of the untrusted kernel, would execute in the secure area only after the reference monitor approves it. What is more, having this framework in place it is possible to define security-critical system primitives that can execute in its entirety in the secure area. Both cases are part of what we denote run-time security primitives. These run-time security primitives could be exposed to user space applications in form of secure system calls. Examples of kernel operations that would naturally make use of (and define new) run-time security primitives include the Crypto

[3]In the worse case, which is that all assets are sensitive, the overhead would be the same than triggering the reference monitor each time that the kernel carries out an action.

API [209], the kernel capability subsystem [174], and the different LSM submodules[4]. As a matter of fact, the boot protection mechanism that we propose in Chapter 9 (*Certainty Boot*), makes use of various run-time security primitives in order to, for example, validate the integrity of the kernel image at boot-time, or verify the running OS at run-time. Also, *secure_read* and *secure_write*, as proposed in Section 7.2.2, are run-time security primitives. We define the run-time security primitives that we use to implement the reference monitor an trusted storage later in this chapter, when we describe them.

Finally, in order to allow users with administrator privileges to have control over the Trusted Cell and configure it, we provide a simple Trusted Cell manager in user space, which we consider part of the REE Trusted Cell. Here, an user that can configure the Trusted Cell is validated in the secure area, where traditional authentication mechanisms such as passwords or pin-codes, or biometric sensors configured as secure peripherals can be used. Configuring the Trusted Cell is done through run-time security primitives, and as such the configuration is subject to the device usage policy (Section 6.2). Any change of the Trusted Cell configuration goes trough the reference monitor, which controls which properties of the Trusted Cell can be accessed (read and write). In order to implement the communication between the Trusted Cell Manager and the Trusted Cell[5], we use *sysfs* since it is a non-overloaded interface (such as *ioctl*) that allows easy scripting, representing the ideal interface for a management user space interface[6]. Note that this setup (*sysfs* + kernel submodule) is also used in popular Linux tools such as AppArmor[7]. Support for kernel space is represented in the lower part of Figure 8.2.

Support for User Applications. Providing support to user space is much less constrained. This is largely due to the fact that we already rely on the *tz_device* user space library that connects to the TrustZone driver submodule. In this way, the REE Trusted Cell in user space can implement different interfaces that use *tz_device* and expose them to different user applications. Since Global Platform specification is widely used in the mobile community, we implement a subset of their Client API on top of *tz_device* as a proof of concept. Implementing the full Client API in the Rich Trusted Cell would give application developers the same functionality they have at the moment, without needing to install a LKM. Indeed, we successfully execute all Global Platform TEE test applications provided with Open Virtualization using our approach. In fact, all our experimentation in Chapter 11 is driven from user space by triggering LSM hooks, using REE Trusted Cell interfaces, and directly using *tz_device*. If our proposal for the generic TrustZone interface is accepted in the mainline Linux kernel, this support would be directly available to user space applications without having to modify the kernel whatsoever.

[4]These are the same submodules that we mentioned in Section 7.2 as components that would benefit from the existence of TrustZone support in the mainline Linux kernel.

[5]Strictly speaking, the Trusted Cell Manager only communicates with the LSM Trusted Cell, which orchestrates the communication with the TEE Trusted Cell using the generic TrustZone driver.

[6]Note that *sysfs* has become the preferred way of communicating user and kernel space in Linux, when device drivers are not involved. Even though *sysfs* does not yet support atomicity, this is not an issue for a management module in user space; each file in the *sysfs* file system can encapsulate all the necessary kernel logic for each command.

[7]`https://wiki.ubuntu.com/LxcSecurity`

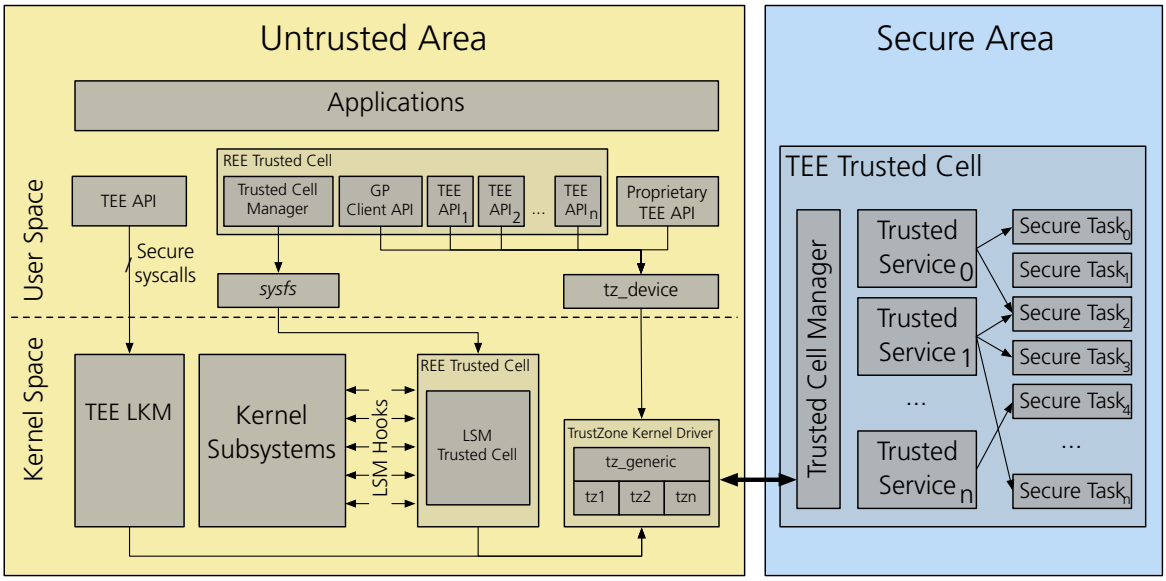


Figure 8.2: REE Trusted Cell Architecture: Commodity OS support for trusted services implemented in the TEE Trusted Cell. Support for both user and kernel space is represented.

The general design of the Trusted Cell allows for a flexible way to implement communication with the TEE. Assuming that *tz_device* exposes a standard interface, service providers could implement their own (proprietary) TEE interfaces. Also, system developers would count on a number of secure system calls they could use, where the actual TEE implementation would be transparent to them; the kernel would implement the communication with the TEE, be it TrustZone, Intel SGX, or a Secure Element. What is more, system developers would have the possibility to create their own ad-hoc secure system calls, supported by run-time security primitives. Here, the use of Loadable Kernel Modules (LKM)s using standard kernel interfaces would be needed. Note that in this case, the LKM does know the interface of the TrustZone driver, since it is a kernel submodule. In this way, the LKMs would define and expose secure system calls to user space, and implement them in the LKM by directly using the generic TrustZone interface, or the LSM framework. This kind of support is reserved for platforms that enable the use of LKMs. Still, this approach surpasses the current alternative, which requires recompiling the kernel. All the proposed alternatives are represented in the upper part of Figure 8.2.

The missing part is loading and attesting a trusted module at run-time, so that it can provide TEE support to the added user space interfaces. This is in itself a large problem that has been partially addressed in the academy and the industry in the context of other secure hardware [39, 276]. Still, a general solution is yet to arrive to ARM-processors using TrustZone. We discuss this further when we get to future work (Section 12.2).

8.1.2 TEE Trusted Cell

Presenting a general overview of the REE Trusted Cell was important in order to clarify the interfaces that we defined for the Trusted Cell, as well as its modular design. For the TEE Trusted Cell however, we will focus on the actual trusted modules that we have implemented, their functionality, and scope. These trusted modules are the base to implement the reference monitor and the trusted storage solution that we present later in this chapter.

Our TEE Trusted Cell prototype is composed by four trusted modules: (i) *Trusted Security Module (TSM)*, which implements cryptographic operations; (ii) *Trusted Integrity Module (TIM)*, which is in charge of hashing and logs to provide integrity; (iii) *Trusted Usage Module (TUM)*, which represents the *Usage Engine* as we have presented it in Section 6.2; and (iv) *Trusted System Primitive Module (TSPM)*, which acts as a TEE Trusted Cell manager. Our prototype is implemented in the secure area, which is leveraged by Open Virtualization's TEE (Section 7.1). We invite the reader to refer to this section (and Chapter 7 in general) to understand Open Virtualization's strengths and limitations, since they shape the actual implementation of the TEE Trusted Cell prototype.

8.1.2.1 TSM

Trusted Security Module (TSM) is the trusted module in charge of cryptographic operations. One of TSM's responsibilities is to abstract these cryptographic operations into functionality that is available to the rest of the TEE Trusted Cell. An example is implementing cryptographic operations over files, where the algorithms to do so can be optimized as a function of the size of the file, the type of file, or the usage of the file[8]. Other examples include encrypting specific payloads or metadata. In general, TSM provides an encryption abstraction. Since cryptography is abstracted, we can choose the encryption algorithms used by TSM. For encryption we use AES-128 since there is no known attack which is faster than the 2^{128} complexity of exhaustive search [46]. AES-192 and AES-256 have shown more vulnerabilities even when they use longer keys [47, 46]. For hash calculations we use SHA-3, which uses the Keccak scheme [43][9] since, even when collision attacks against it are possible [93, 94], the algorithm still remains stronger than the best attacks targeting it[10]. As new attacks appear, and new algorithms address them, these algorithms can be changed transparently to the components using TSM. Note that while we choose these default algorithms, other are also supported and available to the secure area. TSM makes direct use of the *Secure Element (SE)* to store encryption keys in cleartext.

For the actual algorithm implementation we rely on OpenSSL. As mentioned in Section 7.1.1, Open Virtualization exhibits a good integration with OpenSSL. In fact, Open Virtualization delegates the implementation of Global Platform's cryptographic operations to OpenSSL. Since the scope of OpenSSL is much larger than TSM, we do not limit the use of OpenSSL

[8] The encryption of a log file that only grows and should not get modified is optimized differently than a configuration file that is encrypted to be stored in untrusted storage.

[9] Keccak became the SHA-3 standard after winning the NIST hash function competition in October, 2013 (`http://www.nist.gov/itl/csd/sha-100212.cfm`).

[10] `http://gcn.com/Articles/2013/03/04/Attacks-dont-break-SHA-3-candidate.aspx`

in the TEE Trusted Cell; any new trusted module can make use of OpenSSL functions directly. In our TEE Trusted Cell prototype however, only TSM uses the OpenSSL library, and the other trusted modules (i.e., TIM, TUM, and TSPM) use TSM operations instead. The reasons behind this decision is to (i) limit the impact of library changes to trusted modules using TSM, and (ii) streamline the adoption of security critical updates. Changes to OpenSSL can be triggered by the apparition of faster algorithms, stronger algorithms, or the deprecation of certain functions. Critical updates are normally triggered by the apparition of new attacks or the discovery of bugs in the implementation. Abstracting cryptographic operations allows to deploy changes more rapidly without propagating these changes to other parts of the Trusted Cell. The Heartbleed Bug, as mentioned in Section 7.1.1 is a good example of a critical update. Also, maintaining a low coupling between the current encryption library (OpenSSL) and the rest of the Trusted Cell makes it simpler to switch encryption libraries in the future[11].

Future work in the TSM focuses on making cryptographic operations library-agnostic, borrowing the design of the Linux kernel Crypto API [209], where specific algorithms are parameters in a set of standard cryptographic operations. We envision TSM supporting different cryptographic libraries, where these are parametrized, just as cryptographic algorithms are. In this way, TSM would be the one supporting both higher level operations (e.g., encrypting a file), and the direct use of cryptographic operations, instead of exposing library-specific interfaces to the rest of the Trusted Cell.

8.1.2.2 TIM

Trusted Integrity Module (TIM) can be seen as a trusted counterpart of IMA [75] implemented in the secure area. In fact, most of the functionality implemented in TIM is equivalent to the ones provided by IMA. There is however one fundamental difference: IMA executes in the same address space as the rest of the kernel, thus in IMA's attack model, the kernel is trusted. Even when IMA/EVP uses the TPM to implement security-critical operations (e.g., storing integrity measures), calls to the TPM are made from an untrusted environment; if the kernel is compromised those calls could be bypassed. TIM, on the other hand, implements integrity operations entirely in the secure area address space.

TIM stores cryptographic hashes of the assets whose integrity is to be guaranteed. These assets can be anything (e.g., data, keys, files), as long as they belong to the secure area. TIM uses the cryptographic operations provided by TSM to calculate these hashes (i.e., OpenSSL). In case of entire files, TIM stores a cryptographic hash of the file and of its metadata (i.e., inode and dentry) [193]. In general, we refer to any data that TIM store as a *TIM entry*. Any operations affecting a TIM entry are logged. Such a log contains the metadata about the TIM entry, the id of the process modifying it, and the time. Logs and hashes are stored using TSM. Any TIM entry can then be checked for integrity [175]: if the hash of a TIM

[11] After the appearance of Heatbleed, Google forked OpenSSL (`http://www.pcworld.com/article/2366440/google-develops-own-boring-version-of-openssl.html`) and started working on their own encryption library, i.e., BoringSSL (`https://boringssl.googlesource.com/boringssl/`). Supporting a design that allows for the rapid integration of safer encryption libraries is paramount to guarantee confidentiality and integrity in the Trusted Cell.

entry does not correspond with the hash stored in TSM it means that either the TIM entry has been corrupted or that the integrity functions in TIM have been bypassed, which can be considered as an attack. Simple usage control emerges from this mechanism, since a device can define policies based on regular integrity checks. In case of an integrity failure the device can inform the user immediately, log the anomaly and continue the operation for later audit, or lock the device down for later inspection. Extensions to MAC policies are also simple to implement, in such a way that access to a sensitive asset depends on its integrity being verified. These all are examples of integrity-centered usage policies that could be modeled by TUM (Section 8.1.2.3), and enforced by the reference monitor 8.2.2.

One of the most relevant improvements we have done to the IMA functionality is the way we calculate the hash of files, which is one of the TIM entries we want to support. At the time of this writing, IMA (re)calculates the hash of the entire file each time the file is modified. This approach presents two issues:

1. **Performance.** From a performance perspective, calculating the hash of a whole file introduces an unnecessary overhead; a modification that only affects a portion of the file requires recalculating the hash for the entire file. If these modifications are frequent, most of the cryptographic calculations (which tend to be expensive without a dedicated cryptoprocessor [13]) are redundant and unnecessary. Logs files are examples of such scenario. **One solution** is using a Merkle hash-tree [205] to calculate the hash of the file. In this way, the file is logically partitioned in blocks, where the hash for each block is calculated. Then a group of hashed blocks is taken as a new block entry and hashed again. If this process is continued recursively, one unique hash value representing the whole file is produced. Note that actual data is only present in leaf nodes in such a hash-tree; internal nodes are exclusively hashes. The process of calculating the hash for a new file is more expensive than just reading the file and calculating one unique hash. However, in the case described above, if only a portion of the file is modified, then only the hashes concerning that portion of the file have to be recalculated. The effectiveness of hash-trees has been demonstrated in the Trusted Database System [197] and in VPFS [285]. In VPFS two extra beneficial properties that emerge from the use of hash-trees are discussed: (i) correctness and freshness, and (ii) completeness. Since chain of hash sums represent each leaf node, the attacker cannot modify leaf nodes (data), or replace it by an older version undetected. This guarantees correctness and freshness of data. Since the hash-tree is built recursively, forming a tree structure, data blocks cannot be moved, added, or removed undetected. This guarantees correctness of data.

2. **Adoption.** In order to calculate the hash of a file, the file must be read before. If calculating the integrity of a hash to which the process using the file only have privileges to write (not to read) the kernel would reject the integrity check due to lack of enough permissions. IMA solves this by introducing an internal kernel read operation in IMA (*ima_kernel_read()*) which checks the rights that the calling process has over the file, and forces read rights (*FMODE_READ*) if necessary. This way, internal file system checks would be satisfied. While this practice does not introduce a security breach,

given IMA's attack model[12], this approach would not work on a network file system (NFS) since file privileges are managed by the server, as pointed by Alexander Viro in the LKML[13]. Here, if the size of the hash-tree leaf nodes matches the page size (block size in disk), the kernel would take care of bringing the necessary pages to memory (page cache). In this way, those pages could be check for integrity without needing to re-read them from secondary storage, thus avoiding the permission issue discussed here. This approach would also work on NFS since no privileges would be enforced by the host.

Improving our Merkle tree implementation and porting it to IMA is part of our future plans.

In our TEE Trusted Cell prototype we only use TIM to guarantee the integrity of assets entirely handled in the secure area. Examples include: metadata produced by other trusted modules, the file system stored in tamper-resistant storage, or Trusted Cell logs. When we first implemented TIM and TSM [133], we envisioned them as a way to create *secure files* by extending the Virtual File System (VFS), thus protecting file descriptors [193] in the commodity OS. We implemented this by storing the hashes of these files as TIM entries. The goal was to (i) provide an intrusion detection mechanism to guarantee the integrity of a device, and (ii) guarantee the integrity of critical system files. Examples of the files we targeted include: system commands that can help administrators to detect an attacker (e.g., ps, ls, wc), initialization scripts (e.g., init, rc), and system libraries (e.g., libc, libssh, libssl). However, we realized that in an attack model where the kernel is untrusted, such a use of TIM could be neutralized by an attack to the VFS extensions that made calls to the secure area. The problem here is the same we want to solve for split-enforcement; forcing untrusted applications to go through the secure area before using sensitive assets. Note that the sensitive assets described here require different treatment in the secure area than the ones described until the moment; the goal is not to restrict their usage in the untrusted area, but to guarantee their integrity before untrusted applications use them. For some time we thought that such sensitive assets could not be locked, thus impeding the use of split-enforcement. In the process of writing this thesis we reconsidered the problem and came up with some new fresh ideas that we discuss in Section 12.2 as part of our future work.

[12]We expressed our concerns regarding this practice to Dmitry Kasatkin and Mimi Zohar, the IMA kernel maintainers. Dmitry kindly addressed our questions and clarified that hash calculations indeed occur at open-phase, where the process still does not have the file handle, thus cannot access the file. If a second process tries to open the the same file descriptor, a mutex would block it and make it wait until the hash calculation is completed. IMA's assumption, as mentioned above, is that the kernel is trusted. The fact that a process is given extra rights before having access to a file is only represented in the page cache, which is only accessible to the kernel. User space has not access to it. Therefore, if the kernel is trusted, this approach does not entail an extra risk. Our concerns were founded on the incorrect assumption that a second process could potentially get a handle of the same file description due to arbitration. The introduction of *ima_kernel_read()* prevents this (`http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=0430e49b6e7c6b5e076be8fefdee089958c9adad`).

[13]`https://lkml.org/lkml/2014/5/6/523`

8.1.2.3 TUM

Trusted Usage Module (TUM) represents the policy decision point in Figure 6.2. As any other policy engine, TUM receives a set of inputs and generates an output, which in this case is a usage decision. TUM generates the usage decision based on 7 inputs: untrusted application (pid), requested action, sensitive asset involved in that action, sensitive assets to which the untrusted application already has access too, historical actions on sensitive assets, application contract, and device usage policy. Based on these inputs TUM generates one usage decision.

In our TEE Trusted Cell prototype, TUM implements a simple usage control model, where the usage decision is generated by a rule-based, deterministic policy engine. We implement this policy engine through a Finite State Machine (FSM) in the C language, where states are defined in an enumerated type, and transitions are implemented by C functions. The states and transitions resemble the ones in Figure 6.1. In order to allow the FSM to generate a usage decision based on past usage of sensitive assets, the *Sensitive State* stores a list of all the sensitive assets that are accessible to the untrusted application. This list needs to be cleared before the untrusted application can come back to *Untrusted State*, and finish its execution. Since Open Virtualization permits to maintain the internal state of different secure tasks in a per-session basis, FSMs capturing the state of different untrusted applications can be maintained in main (secure) memory simultaneously. By maintaining the state of all untrusted applications in the same session, TUM can generate usage decisions satisfying both each individual application contract, and the device usage policy (which governs the device as a whole).

Our TUM implementation allows us to generate usage decisions based on the inputs stated above. However, it is not a complete usage control model such as UCON$_{ABC}$ [221], which implements a formal model through an algebra, and provides a language to define usage policies. In this way, our implementation of TUM is more a proof of concept for testing the Trusted Cell framework than an actual usage control solution. Here, we want to emphasize that the focus of this thesis is to provide support for run-time security in general. Usage control enters in the definition of run-time security and therefore we seek providing support for the generation, and specially the enforcement, of usage decision, but our research is not on usage control models.

This said, TUM is designed to support a complete usage control model such as UCON$_{ABC}$, where a policy engine receives a number of inputs and produces an output. If generating the output entails complex computations, the computational requirements might increase, which will require more engineering. However, from the architecture perspective changes are minimal. Also, TUM meets the strong security requirements made in UCON$_{ABC}$ in terms of isolation, availability, and usage policy enforcement established in the UCON$_{ABC}$ specification [221]. In this way, switching TUM's policy engine to a UCON$_{ABC}$ implementation not only would be simple in terms of the architecture, as the inputs and outputs of any policy engine are very similar, but to the best our our knowledge, it would represent the first UCON$_{ABC}$ implementation that meets the required security assumptions. Note that we are not saying that implementing UCON$_{ABC}$ is simple, but that porting an implementation of UCON$_{ABC}$ (or any other usage control model) to TUM would be.

Finally, TUM delegates the confidentiality and integrity of its data to TSM and TIM respectively. By making use of other trusted modules, the TEE Trusted Cell follows the modular design described in its architecture (Section 8.1).

8.1.2.4 TSPM

Trusted System Primitive Module (TSPM) is the TEE Trusted Cell manager. It has two responsibilities: (i) it defines the run-time security primitives that the TEE Trusted Cell implements, and (ii) acts as a dispatcher to forward run-time security primitives to the trusted module implementing them. Note that TSPM is a trusted module in itself, therefore run-time security primitives related to TEE management are directly implemented in the TSPM.

In Open Virtualization, each secure task is given a global identifier. These are organized in a per-session basis, in such a way that sessions define the identifiers for the secure tasks they implement. A secure task can be assigned to different sessions. In this way, secure tasks are called in a command fashion; untrusted applications open a session and issue commands to the secure world, which Open Virtualization's dispatcher forwards to the correct secure task. TSPM sits on top of Open Virtualization's dispatcher and abstracts the identifier definition; TSPM assigns a unique identifier to each secure task, since this simplifies how secure tasks are mapped to trusted modules. The assumption is that secure tasks will be heavily reused in order to minimize the TCB. Then, TSPM maps each secure task to run-time security primitives. Since TSPM is the general TEE manager, it also exposes the specifications that are implemented by the TEE Trusted Cell. In our prototype, where Global Platform's API is supported, TSPM also takes care of mapping the operations in Global Platform's specification to actual secure tasks. Note that Sierraware had already implemented in Open Virtualization most of Global Platform's specification; we added some missing functionality and improved some parts, specially regarding OpenSSL integration for cryptographic operations.

One of the main design goals for TSPM is that it serves as a hub for the interfaces supported by the TEE Trusted Cell. The reason behind this design decision is twofold: (i) if the TSPM knows all the interfaces (and specifications) supported by the Trusted Cell, then it can implement the *Secure Device Tree* we introduced in Section 7.2.2, when we discussed how the secure area shares information with the generic TrustZone driver at boot-time. Also, (ii) this would help a trusted module implementing the installation of trusted modules as introduced earlier in this chapter, to keep track of which trusted modules are already installed, and more importantly, which secure tasks are supported. If we envision such a trusted module, i.e., *Trusted Service Installer (TSI)*, connected to an online application market for trusted services, it would be of particular relevance to count on detailed information about the trusted services supported by a device. While in the untrusted area having several applications that overlap functionality (as it is the norm in general purpose devices) does not have any downside more than the actual space they take in secondary storage, in the secure area maintaining a low TCB is paramount. As mentioned, TSI is out of the scope of this thesis. However, we have designed our TEE Trusted Cell manager (TSPM), to support it. We discuss this further in Section 12.2.

8.2 Trusted Services

In the beginning of this chapter we introduced the concept of *Trusted Services*, and described their materialization in the Trusted Cell in form of *Trusted Modules*. Now that we have described the four trusted modules implemented in our TEE Trusted Cell prototype, as well as the mechanisms to interface them from the REE Trusted Cell, we present two trusted services that we have implemented with this support: a trusted storage solution and a reference monitor. The trusted storage solution guarantees the confidentiality and integrity of sensitive data at rest; the reference monitor mediates access to secure peripherals and sensitive data (at rest and in motion). We refer to secure peripherals and sensitive data as sensitive assets. Note that protecting sensitive data in motion is synonym with guaranteeing integrity and confidentiality of main memory; at least of the memory regions containing sensitive data. We present our trusted storage solution first, since the reference monitor makes use of it as its locking mechanism for sensitive data at rest.

8.2.1 Trusted Storage

A complete *Trusted Storage* solution should guarantee the integrity, confidentiality, availability, and durability of sensitive data. While availability and durability are desired properties, which are common to all storage solutions, it is integrity and confidentiality that are specific to trusted storage. Therefore, these are the properties that we focus on in the trusted storage solution based on our Trusted Cell. In Section 3.2 we presented the state of the art in trusted storage. Given our attack model, where the whole commodity OS is untrusted, all the presented solutions that are integrated in the commodity OS fail to provide confidentiality and integrity of sensitive data. Indeed, the only trusted storage solutions that, to the best of our knowledge, can guarantee integrity and confidentiality of sensitive data under our attack model are the ones based on secure drivers (Section 6.3.1). However, as already discussed in this section, solutions based on secure drivers entail a dramatic increase of the TCB, restrict use cases, and impose specific software stacks. One exception is VPFS [285], which successfully reuses untrusted legacy code by limiting the scope of user applications; applications execute in the (equivalent to the) trusted area, thus the actions that their actions are trusted. In fact, in VPFS usage policies could be directly enforced at the application level. We compare our solution with VPFS at the end of this chapter (Section 8.4).

Our trusted storage solution is based on split-enforcement. Thus, sensitive data is locked when is stored in secondary storage and unlocked when used in the untrusted area. In other words, data is locked when at rest, and unlocked when in motion. Trusted storage, as a trusted service, only takes care of data at rest since it is in this state that it resides in secondary storage; data in motion resides in main memory, thus it is the responsibility of a different trusted service to protect it. The trusted service protecting data in motion in our Trusted Cell prototype is the reference monitor, which we introduce next (Section 8.2.2).

Locking data at rest relies on cryptographic operations. In order to adapt the encryption scheme to split-enforcement and in order to be able to lock data in secondary storage, we decouple encryption, which corresponds to the first phase, from decryption, which corresponds

to the second phase. By decoupling these operations and associating them to different components (probably in different software stacks) we make encryption the locking mechanism and decryption the necessary unlocking one. At this point, we have two possibilities: (i) sensitive data is generated by a component in the trusted area (e.g., secure peripheral, trusted module), and (ii) sensitive data is generated by a component in the untrusted area (e.g., untrusted application, kernel subsystem). In general, sensitive data should be generated in the trusted area whenever possible either by the use of run-time security primitives or secure peripherals. Otherwise, sensitive data should be assumed to be *compromised on creation*. Let us explore these two cases independently: the problems, the solutions we propose, and how we implement them.

8.2.1.1 Trusted Sensitive Data Generation

This is the most favorable scenario. If data is generated within the trusted area, there is no risk in that it remains in clear both in main (secure) memory and in secure secondary storage (*Secure Element (SE)*). Sensitive data in the trusted area can be generated by a software component executing in the secure area or by a hardware component assigned to the trusted area as a secure peripheral. Let us use encryption key generation as an example. When TSM generates a new encryption key, it is placed in secure memory and afterwards stored in the SE. Note that if the hardware platform counts on a Random Number Generator (RNG), this is configured as a secure peripheral before it assists to generate the encryption key. Since all communication with the RNG is protected by the NS bit, the security of the operations is defined by TrustZone's level of tamper-resistance (Chapter 10.1). In this case, since the RNG could in principle be used by an untrusted application, any sort of cache needs to be *zeroized* as part of the process of assigning the peripheral to the untrusted area. In either case, the encryption key would remain in the trusted area, preventing access from untrusted components. This would also be the scenario covering Global Platform use cases were sensitive user information is captured trough secure peripherals (e.g., pin-entry-keyboard, touch screen) in the secure area.

Since tamper-resistant storage is a limited resource, TSM can make use of untrusted secondary storage[14] to store encrypted sensitive data. The underlying assumption, as discussed in Section 6.3.2, is that we trust the encryption scheme (Section 8.1.2.1). In order to guarantee the confidentiality of the encrypted data, encryption keys never leave the trusted area in clear. Seeking to maximize the flexibility of our trusted storage solution we use TSM to create encrypted containers in the untrusted file system. This way, TSM can use the storage scheme that most suit the data being stored (e.g., block-based, key-value, object-oriented). We denote these encrypted containers *Secure Containers*. Note that the use of encrypted containers to provide trusted storage has been used in popular academic and commercial solutions such as VPFS [285], DepSky [45], and TrueCrypt [30]. In the case of the Trusted Cell, and giving its architecture, the remaining question is: How to create a secure container in untrusted storage without having access to it? As we discussed in Section 6.3.2, we chose

[14]Note here we refer to the software stack (I/O stack in this case) that is untrusted. As discussed in Section 6.3.2, we assume that hardware is trusted and operates correctly. However, the fact that the software handling secondary storage is untrusted makes the storage layout (e.g., file system, inodes, metainformation) untrusted. Thus the operations over stored data are also untrusted.

to re-use the existing infrastructure in order to minimize the TCB. A consequence of this decision is that access to secondary storage has to necessarily be done through the untrusted I/O stack. In fact, the trusted area does not implement an I/O stack, hence it has no access to untrusted secondary storage.

In order to create secure containers in the secure area and store them in untrusted storage using the I/O stack in the commodity OS, we make use of secure shared memory. As described in Chapter 2, TrustZone supports the creation of shared memory. Our approach, formed by Open Virtualization TEE and our generic TrustZone driver implements it. The process of creating a secure container is depicted in Figure 8.3.

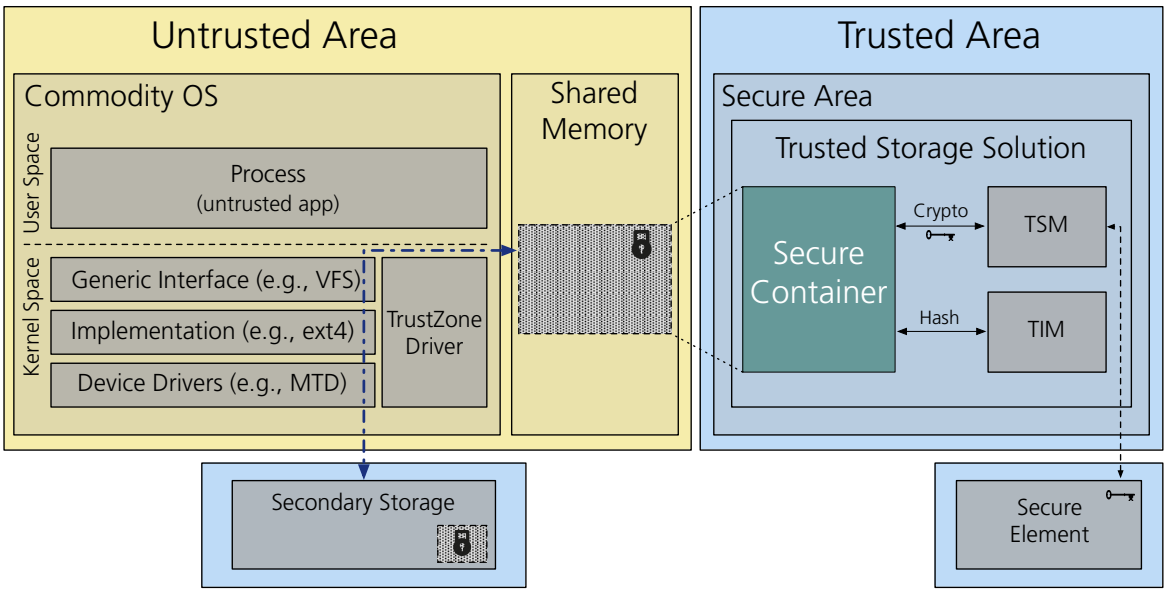


Figure 8.3: Trusted Storage for Trusted Data Generation. Use of *Secure Containers* to provide a trusted storage solution for sensitive data generated in the trusted area using the untrusted I/O stack in the commodity OS.

The trusted storage service allocates a memory region in *secure memory*, which is not accessible to the untrusted area. It formats it to create the secure container, and then stores sensitive data in it. When the secure container is ready for storage, TIM calculates the hash of each stored component independently, places it in the container, and calculates the hash of the container. The hash is then stored in the SE. Afterwards, TSM generates an encryption key, encrypts the secure container, and stores the encryption key in the SE. At this point, a shared memory region is allocated in the untrusted area, and the memory region containing the secure container is copied to that newly allocated shared memory region, while secure memory is freed. The secure area then *requests* the commodity OS to store that piece of memory in secondary storage. We implement this by having a preallocated piece of memory that is used by the REE Trusted Cell and the TEE Trusted Cell to communicate. This technique is also used in frameworks such as OpenAMP [137] for communicating two different software stacks. After receiving the request, the REE Trusted Cell takes care of issuing the necessary I/Os and acknowledging completion to the TEE Trusted Cell when the secure container is completely stored in secondary storage. As part of the completion,

the REE Trusted Cell sends to the TEE Trusted Cell the necessary metadata to identify the file representing the container (i.e., inode). Note that the secure container is a file for the commodity OS. In order to retrieve the secure container, the TEE Trusted Cell issues a request to the REE Trusted Cell, which reads the file in untrusted memory. The secure area coverts that piece of memory into shared memory and copies the secure container to the secure memory space. Memory in the untrusted space is released. At this point, the TEE Trusted Cell can decrypt the secure container and use its contents.

Since this solution relies in the correct operation of the REE Trusted Cell, which is untrusted, the satisfaction of I/O requests is not guaranteed. In fact, an attacker could compromise the REE Trusted Cell by injecting code in the kernel with the objective of misusing the communication interface with the TEE Trusted Cell in different ways: from not attending the storage request at all, to returning incorrect data (e.g., secure container, metadata, acknowledgment), or even using it as an attack vector. Here, we provide a complete analysis of the attacks that could be launched against this communication interface in Section 10.1. While these attacks might affect the availability or durability of sensitive data in our trusted storage solution, **they do not affect integrity nor confidentiality**. Secure containers are always encrypted and neither hashes nor encryption keys ever leave the trusted area in clear. Availability and durability could be improved by means of redundancy, distribution, or recoverability mechanism based on periodic backup as in [285]. This is a topic for future research.

One of the features that this design supports is intrusion detection. The fact that a class of attacks does not compromise the proper operation of the secure area in itself, or the sensitive data stored on it, but do compromise the services in the secure area, allows the secure area to use secure containers as *bait*. This way, the TEE Trusted Cell could detect some attacks launched against it. More specifically, those attacks targeting the REE Trusted Cell. We explored this idea in the context of both intrusion detection, and self-recovery; if the Trusted Cell detects an attack it could (a) inform the user, (b) lock itself down to prevent future attacks and protect sensitive data, or (c) try to fix itself. In our current implementation, we can detect when a secure container has been compromised and follow (a) or (b) based on the device usage policy. We explore (c) in future work (12.2) in the context of *antifragile* [268] trusted storage and run-time security in general.

Finally, note that since data is generated in the trusted area, there is in reality no need to store it in a secure container. We implement this functionality thinking on a scalable way to store secure logs and metadata. In principle, the SE has enough space[15] to store several encryption keys, certificates, and signatures. In this way, we do not recommend that encryption keys leave the trusted area. Still, if and advanced encryption scheme is used, where different keys are used for different blocks (or other division of the physical storage device), encryptions keys and other encryption metadata can be stored in secure containers.

[15]Commercial Secure Elements feature different sizes. From some hundred KB to a few MB.

8.2.1.2 Untrusted Sensitive Data Generation

Another possibility is that sensitive data is generated in the untrusted area. This is less favorable since in this case sensitive data would be, as mentioned above, *compromised on creation*. Interestingly, this is the typical case in commodity OSs: untrusted applications collect sensitive user data that they store, in the best case encrypted, on secondary storage. Here encryption keys are also generated and stored in the untrusted environment. While this might change in newer untrusted applications, which outsource sensitive data collection to secure tasks when secure hardware is available, this is the case for legacy applications. In this section, we present a trusted storage solution that, even when it assumes that sensitive data is *compromised on creation*, it stores it securely without compromising encryption keys. In this way, this technique provides better guarantees than the current state of the art. Still, we would always recommend to generate sensitive user data using the trusted area.

This exact scenario is the one we used to introduce split-enforcement in Section 6.3.2, represented by Figure 6.3. Here, encryption occurs directly in the hardware (i.e., physical storage device). If we look at flash memories, any Solid State Drive (SSD) nowadays comes with a dedicated cryptoprocessor so that data is better distributed throughout flash memory; this same cryptoprocessor can be used in principle to encrypt data blocks for our purpose. In fact, as we discussed in depth in Section 6.4, next generation I/O devices (e.g., SSD) will be driven by software defined storage (SDS) techniques. This will allow to move part of the TEE Trusted Cell down to the storage device, thus enabling more secure encryption schemes (e.g., use different keys that are stored in the SE, use of a RNG in the trusted area). In this case, encryption is transparent to the untrusted area. Moreover, since encryption occurs on hardware belonging to the trusted area, encryption keys are not compromised, which sets our proposal apart from existing trusted storage approaches integrated in commodity OSs (Section 3.2). Here, we require a management component in the storage device that allows to communicate with the secure area both at boot-time and run-time. This allows to implement an asymmetric encryption scheme where encryption occurs on the hardware, but decryption on the secure area. Hence, giving the TEE Trusted Cell the possibility to enforce usage policies by means of split-enforcement. Such management component is far from being an I/O stack; it provides a narrow communication interface to exchange storage meta-information, guaranteeing a low TCB.

Note that this configuration responds to an *encrypt everything* scheme, where every block is encrypted before being written to disk. Configurations exhibiting larger encryption granularity become viable when the TEE Trusted Cell can place a trusted module in the storage device. Current hardware does not support programming flash storage devices directly; they are black boxes implementing proprietary algorithms for data placement and garbage collection[16]. The advent of *Software Defined Storage (SDS)* represents a huge possibility for trusted storage solutions implementing split-enforcement techniques. We consider this a fascinating topic for future work.

This approach in itself does not suppose a big contribution, since sensitive data can be directly leaked or manipulated, and the only protection comes from the assumption that

[16]This is normally implemented in flash devices inside of the Flash Translation Layer (FTL).

data reaches untampered the storage device. However, if we combine this technique with the memory protection mechanism we will introduce in Section 8.2.2.1, this technique becomes a novel approach to protect sensitive data directly generated by an untrusted component (e.g., untrusted application) without outsourcing sensitive data production to the secure area. In next section, we will extend this approach with memory protection techniques provided by the reference monitor in order to guarantee that sensitive data in the untrusted area is *protected on creation*.

8.2.2 Reference Monitor

A reference monitor is a component in a system that mediates access to sensitive assets based on a given policy. As mentioned in the state of the art (Section 3.3), reference monitors have been used to protect operating system assets from illegitimate access in the form of MAC [12, 166].

We implement a reference monitor that enforces access and usage control policies using split-enforcement. The reference monitor architecture corresponds to Figure 6.2, and follows the requirements that we defined before (Section 6.2). Usage policies are generated by TUM, which gets its inputs from the untrusted area through the REE Trusted Cell. As we discussed when we introduced split-enforcement (Chapter 6), the remaining question is: How do we lock sensitive assets in order to force the untrusted application's workflow to go through the usage decision point in the secure area?

We have identified three assets for which locking primitives must be provided in order to implement a reference monitor: sensitive data in motion (i.e., in memory), sensitive data at rest (i.e., in secondary storage), and peripherals that either generate or manage sensitive data in any way. Combining these three locking primitives, the reference monitor can authorize a piece of unencrypted sensitive data to be referenced by untrusted applications in the commodity OS, and at the same time **enforce** that this data cannot be transmitted over a network peripheral, or even obfuscated using memory operations. In this way, service providers can use untrusted applications to implement innovative services on top of untrusted commodity OSs. Users can define usage policies that are actually enforced.

8.2.2.1 Memory Locking

In order to keep track of sensitive data used from the untrusted area in unencrypted form, it is necessary to monitor how untrusted memory regions are used. The goal is to mediate the memory operations involving these memory regions through the secure area, in such a way that they are subject to the usage policies in TUM. In this way, sensitive data can be manipulated in the untrusted area, without the risk of being leaked or compromised.

In TrustZone, the memory space is divided between the trusted and the untrusted area. In order to support tracking memory operations involving sensitive data, we need a mechanism that allows to lock regions of untrusted memory in such a way that memory accesses are forced to be forwarded to the secure area in order to unlock these regions. When a memory

operation reaches the secure area, the memory region is only unlocked if the operation satisfies the usage policies in TUM. We denote these memory regions *Trusted Memory Enclaves*. As long as sensitive data is kept in trusted memory enclaves, the Trusted Cell can verify its integrity, and enforce that any operation affecting it will be mediated by the usage policies in TUM. In order to implement trusted memory enclaves, we need two mechanisms: (i) a way to represent and keep track of how a specific region of memory is used, and (ii) a way to mediate accesses to that region of untrusted memory from the secure area.

In order to keep track of regions of untrusted memory, we investigate Code-Pointer Integrity (CPI) [182]. As described in Section 3.3.3, CPI provides the necessary mechanisms to guarantee the integrity of code pointers in a program (e.g., function pointers, saved return addresses), thus allowing to verify the integrity of a memory region. In order to prevent attacks against these mechanisms, CPI proposes a ***safe region*, which is trusted**, where pointer metadata is stored. An implementation of CPI depends then on the resilience of such safe region. This design suits indeed the paradigm of a execution environment divided between a trusted and an untrusted area; the safe region naturally belongs to the trusted area, while the monitored memory region resides in the untrusted area.

Implementing CPI in TrustZone is not a simple task. While x86 does support memory protection instructions (e.g., Intel TXT [219], Intel SGX [158], AMD SVM [7]), which are the base technology for all the work in memory integrity presented in Section 3.3.3, ARM does not. In fact, TrustZone-enabled processors such as Cortex-A9 or Cortex-A15 do no support a Memory Protection Unit (MPU), and it is only optionally available for the Cortex-M line [23]. Moreover, TrustZone does not support full virtualization: TrustZone does not support trap-and-emulate primitives (Section 2.4). As a result, it is impossible to virtualize the non-secure MMU and arbitrate its use from TrustZone's secure monitor (Section 2.4). While FIQ and IRQ interrupts can be configured to always be trapped by the secure monitor, the MMU is a special case. TrustZone enables a MMU per world, which means that each world can freely manage its address space without intervention from the secure monitor. As a consequence, memory operations cannot be trapped and forced to go through the secure area to enforce usage policies. This same problem appears when trying to emulate devices using TrustZone. The creators of Genode make a very good description of this issue in [123]. Here, the Genode team argue how device emulation is not supported, imposing the necessity of patching the kernel with hypercalls in their TEE. Again, given our attack model, this is not an option.

The only solution we can devise using current ARM hardware is using ARM virtualization extensions to virtualize the MMU. Such extensions are mandatory in the ARMv8 architecture, and are present in AMRv7 processors such as the Cortex-A15 [22]. Using these virtualization extensions, a thin hypervisor can be used to either (i) redirect memory operations to the secure world, or (ii) directly implement a TEE Trusted Cell module to enforce memory policies. This way, the untrusted area does not need to be modified, therefore invalidating the attack vector where an attacker bypasses calls to the secure area. While this solution allows to implement trusted memory enclaves, it augments the size of the TCB; a hypervisor - even when an extremely thin one, since we are not interested on virtualization capabilities such as multiguest support -, is needed. This solution makes the attack surface larger. However, it does allow to reuse the untrusted I/O stack to avoid code replication,

which was one of our primary goals. In any case, introducing a hypervisor in the TCB is always better than replicating complex, time-tested, and community maintained pieces of software such as the Linux I/O stack. Until a TrustZone revision is presented that supports trap-and-emulate, it is necessary to implement a hypervisor based on hardware virtualization (i.e., trap-and-emulate) to support trusted memory enclaves.

By defining trusted memory enclaves, we force memory operations involving sensitive assets in the untrusted area to go through the secure area and feed the policy engine. The performance overhead is the same that with any other context switch to the secure area. The overhead introduced by CPI in itself seems also minimal in the results presented in [182], but we need to implement CPI in our Trusted Cell prototype before drawing further conclusions. This is a necessary next step for future work.

At the moment of this writing memory locking is indeed a useful mechanism that can bring memory protection features similar to the ones present in Intel SGX to ARM-powered processors featuring virtualizations extensions. What is more, such memory protection mechanisms will be paramount when persistent memory becomes the norm. When this happens data at rest and in motion will be the same, and memory operations need to encapsulate data locking (i.e., encryption) as I/O operations do today. We expect trusted memory enclaves to become a general practice. In this section, we present a design for ARM architectures supporting full virtualization.

8.2.2.2 Data Locking

Locking data at rest relies on cryptographic operations, and it is supported by the mechanisms that we have already described for our trusted storage solution (Section 8.2.1). In fact, trusted storage is the locking mechanism for sensitive data at rest in our Trusted Cell prototype. As a short recapitulation, encryption and decryption are decoupled in split-enforcement's first and second phase respectively, allowing to place encryption and decryption primitives in different software stacks. In this way, encryption becomes the locking mechanism, and decryption the necessary unlocking one. Since data generation can take place in either the trusted or the untrusted area, two types of data locking emerge. In both cases, encryption is transparent to the untrusted area, making decryption a necessary step in order to make sensitive data usable. As in other examples of split-enforcement, before unlocking the sensitive asset (decrypting), a usage decision is generated and evaluated.

Data locking for sensitive data generated in trusted area. In this case, data locking occurs directly on generation and encryption keys directly reside in the SE, which is only accessible to the secure area. Unlocking necessarily takes place in the secure area. Any attempt to bypass the secure area would result on the secure container in which sensitive data is being stored to be unavailable (Section 8.2.1.1). The underlying assumption is that we trust the encryption scheme. In this way, unlocking is forcedly mediated by the usage policy engine in the reference monitor, thus complying with split-enforcement.

Data locking for sensitive data generated in untrusted area In this case, encryption takes place in hardware in order to guarantee that encryption keys are not compromised (Section 8.2.1.2). I/O operations are mediated in hardware, which is unaccessible to the untrusted area. Thus, data locking is guaranteed. What is more, data unlocking can take place on different places depending on the trust assumptions specific of a device: (i) sensitive data is directly decrypted in hardware and usage decisions are mediated by a TEE Trusted Cell component in-built in the hardware; or (ii) sensitive data is not decrypted on hardware and decryption must necessarily take place in the secure area. In either case, we trust the encryption scheme. This possibility requires a key exchange between the secure area and the storage device, which takes place at boot-time when the trusted area is initialized, but the untrusted area is not. This exchange can be maintained at run-time using the same management component. As mentioned, managing keys requires a minimal increase of the TCB.

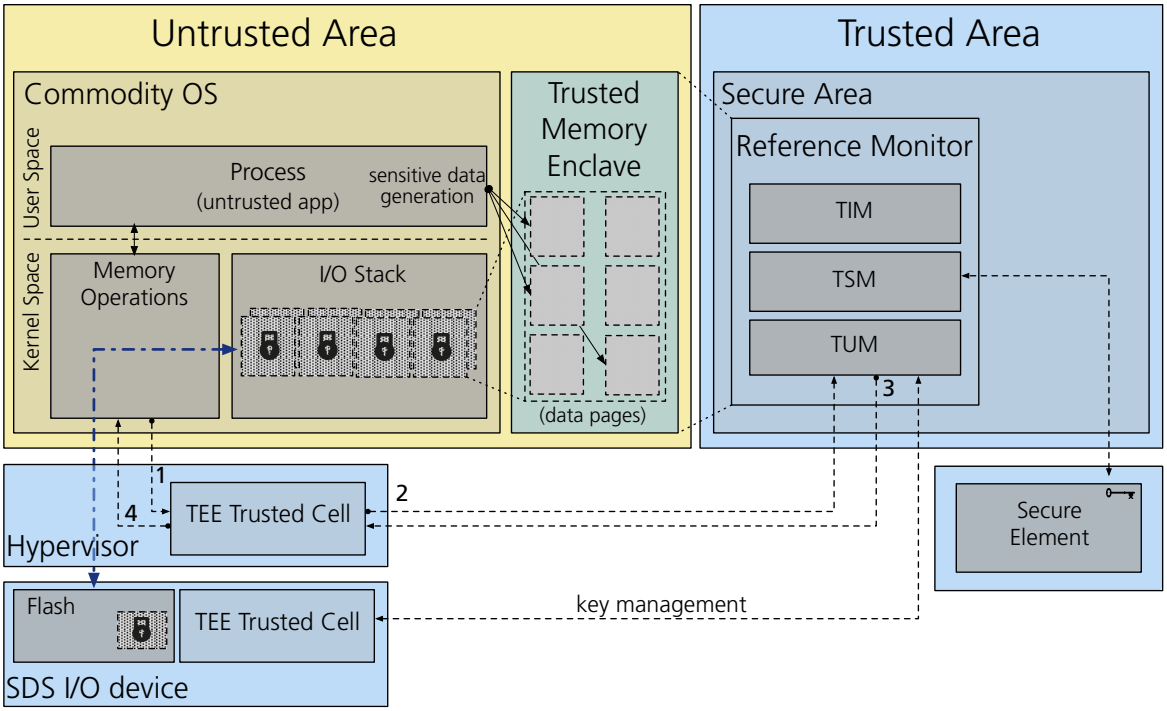


Figure 8.4: Trusted Storage for Untrusted Data Generation using *Trusted Memory Enclaves*. An untrusted application can use a trusted memory enclave to produce sensitive data. Data pages in the trusted memory enclave are encrypted before leaving it and follow their path down the I/O stack to be stored. This guarantees confidentiality and integrity of sensitive data. If pages reach the SDS I/O component, it can guarantee data durability. The encryption scheme is decided between the different TEE Trusted Cell components.

If this data locking technique is combined with the memory locking mechanism we have just presented, based on trusted memory enclaves, data would no longer be *compromised on creation*. Indeed, trusted memory enclaves guarantee that memory operations are mediated through TUM, which allows to guarantee integrity and confidentiality of pages containing sensitive data. Since in this case, data pages leave the trusted memory enclave to reach sec-

ondary storage, we encrypt them. Encryption is carried out by TSM, and integrity metadata collected by TIM, thus confidentiality and integrity are guaranteed. This technique is similar to the one used on Overshadow [62].

This allows for a new class of innovative services that do not rely on secure tasks to generate sensitive data; sensitive data can be directly generated in the untrusted area and protected by trusted memory enclaves provided by the trusted area. The motivation for this technique is to increase the security premises of legacy untrusted applications generating sensitive data. However, since memory protection requires more interaction with the untrusted commodity OS, the attack surface is larger. Also, even though confidentiality and integrity are still guaranteed (pages are encrypted when they reach the I/O stack), availability and durability are not; the untrusted I/O stack could still prevent data pages to reach secondary storage. While it is true that durability can be guaranteed once data pages have reached the SDS I/O device (note that it mediates I/O requests through TUM), it is not full solution. We discuss this in detail in our security analysis (Section 10.1.4.4).

This technique improves the security of sensitive data produced in the untrusted area significantly, however the premise is that trusted memory enclaves are not compromised. When possible, sensitive data should be generated in the trusted area. Figure 8.4 depicts this process.

8.2.2.3 Peripheral Locking

As described in Section 2.4, one of TrustZone's most relevant security features is the possibility of securing peripherals at run-time. This is in essence the locking/unlocking mechanism that we propose. A peripheral can be configured to be accessible or non-accessible to the untrusted area by writing to a specific register (*TrustZone Protection Controller (TZPC)*) from the secure area. In our split-enforcement architecture the workflow would be the following. At boot-time all the peripherals that are marked as sensitive assets in the device usage policy are mapped to the secure area, and are therefore not accessible to the untrusted area. This corresponds to an *a priori* enforcement in the first phase. From this moment, whenever the untrusted area makes use of a sensitive peripheral, a call to secure area using the REE Trusted Cell is issued. The call, which maps to a secure system call, feeds the policy engine (TUM), which produces a usage decision based on the sensitive asset, context, and internal state, as described in Section 8.1.2.3. If the usage decision is positive, the reference monitor assigns the peripheral in question to the untrusted area by writing to the TrustZone register that modifies the TZPC for that peripheral. Otherwise, a permission error is returned. In either case, the control flow is returned to the untrusted area. Note that the untrusted area can bypass the call to the secure area. In this case, access to the peripheral is rejected by the AXI-to-APB bridge (Section 2.4).

In order to preserve the locked status of a secure peripheral, it is mapped back to the secure area when the untrusted application that was authorized to make use of it is preempted. This is possible by capturing the IRQ interrupts (Section 7.1.1.2) in the secure monitor before the interrupt handler in the untrusted area processes them. In order to avoid having to maintain a state in the secure area with the peripherals that have been enabled in the

Subsystem	Lines of Code Modified
TSM	+ 510
TIM	+ 158
TUM	+ 1624
TSPM	+ 580
LSM Framework	+ 23
Total	+ 2895

Table 8.1: Lines of coded (LOC) modified to implement the trusted storage solution and reference monitor in our Trusted Cell prototype. It is important to mention that some operations are delegated to libraries in the secure area, which make that the number of LOC is reduced. For example, crypto operations are outsourced to the OpenSSL instance executing in Open Virtualization. Note that the LOC include header files. We use *cloc*[18] to count LOC, and we omit comments or blank lines in our numbers.

secure area, IRQs reset all peripherals to their initial state. If a higher granularity is needed, state can be maintained in the secure monitor. An example use case is mapping a peripheral to the untrusted area until a specific interrupt occurs. Since the interrupt is captured in a component in the secure area, the chain of trust is maintained.

In our performance evaluation (Section 11.2), we will show the performance overhead of triggering a context switch to the secure area. In our experiments, where we issue a call to the secure area for every system call in the untrusted area, we show that the overhead is less than 16% in the worst case, when using different workloads. The cost of mapping a peripheral to the secure area is that of an average instruction, where a register in memory is set to a new value.

8.3 Prototype

We build a prototype for the Trusted Cell using Xilinxs Zynq-7000 as a hardware platform, Open Virtualization as a TEE (Section 7.1.1), Linux Kernel with generic TrustZone support (Section 7.2) as kernel space, and a light command-base version of Ubuntu[17] as user space.

We have implemented the *REE Trusted Cell* completely as described throughout this chapter. We implement the *LSM Trusted Cell* as a LSM module in the Linux kernel, a REE Trusted Cell manager in user space that communicates with the REE Trusted Cell kernel part through *sysfs*, and a portion of Global Platform's Client API that communicates with the generic TrustZone kernel driver in kernel space using the *tz_device*. Our prototype implementation follows the modular design described at the beginning of this chapter, thus adding new TEE APIs to the REE Trusted Cell is significantly simplified if we compare to the state of the art in TrustZone TEEs (Section 7.2.1).

We have also implemented the *TEE Trusted Cell* in the secure area. Here, we implement

[17] `http://javigon.com/2014/09/02/running-ubuntu-on-zynq-7000/`
[18] `http://cloc.sourceforge.net`

TSM, TIM, TUM, and TSPM as we have described them in Section 8.1.2. We use then these four trusted modules two provide the two trusted services we have just described: a reference monitor, and a trusted storage solution. We have fully implemented the trusted sensitive data generation solution based on secure containers. However, we encounter two fundamental limitations in our hardware platform when implementing untrusted sensitive data generation: the lack of PCI-express connectors in Zynq ZC702 that allow to connect a SDS I/O device to it, and the lack of hardware virtualization extensions neither in TrustZone as a technology, nor in the Zynq-7000 Cortex-A9 processors. It is important to mention that at the time we started our research on TrustZone (back in 2012), Zynq was the only hardware platform that allowed for TrustZone experimentation: TrustZone was enabled in Zynq SoCs, and TrustrZone registers where accessible [298]. Moreover ZC702 was the only Zynq-7000 board supported by Open Virtualization (in fact, one of the first being supported at all), which at the same time was - to the best of our knowledge - the only non-commercially TEE framework available to the public at that time. This limited our options in terms of choosing an experimentation platform[19]

In our prototype, we have taken the Zynq ZC702 and Open Virtualization to the end of their possibilities, both in terms of software and hardware utilization. This has allowed us to better understand TrustZone and virtualization, since we had to overcome these limitations with creative solutions. It has also allowed us to understand the hardware requirements that a full implementation of split-enforcement requires. This will help us in future iterations of this work. In fact, putting the research and solutions present in this thesis together with hardware fully supporting split-enforcement we are in the point where we are describing a commercial prototype to support trusted storage and usage control.

We now present the two pending shortcomings in the implementation of our current prototype and how we moved forward to overcome them. The first shortcoming is related to hardware-assisted encryption; the second is related to memory locking. Table 8.1 shows the LOC necessary to implement our Trusted Cell prototype.

Hardware-Assisted Encryption. The first shortcoming comes from the fact that Zynq does not provide read/write non-volatile physical storage (NVRAM). This means that any form of secondary storage is accessed as a peripheral. We use an SD Card in our prototype. There are two alternatives if we want to attach a Solid State Drive (SSD) to Zynq: (i) we could interface the flash memory through the network using I2C, Samba, etc.; (ii) we could use Zynq's Low-Pin Count (LPC) and High-Pin Count (HPC) interfaces to implement a PCI Express controller and, through an adapter, attach the flash memory to the board, treating it as any other peripheral. Note that all these interfaces have their representative trusted registers to map them to the secure area in Zynq [298]. An extra inconvenient is that, even though any SSD nowadays features a cryptoprocessor, the firmware controlling the SSD (FTL) is typically closed and proprietary. This prevents us from directly using features in the storage device. Another possibility would be using Zynq's Programmable Logic (PL), which is equivalent to an FPGA, to implement a dedicated cryptoprocessor based on Microblaze[20] so that blocks are encrypted in a trusted component before they are stored. However, this

[19]Note that others have reported having the same issues [291].
[20]`http://www.xilinx.com/tools/microblaze.htm`

solution would depend very much on the hardware we are using, preventing us from providing a hardware-agnostic solution as we intend. Still, OEM could investigate this path to provide better trusted storage solutions in their designs. In this context, we are working together with Xilinx, to incorporate our work to their next generation silicon.

All in all, the engineering effort for implementing hardware-assisted encryption would have forced us to deviate decidedly from out research interests. We would have had to (i) implement support for NVRAM in Zynq (e.g., a PCI Express driver on top of LPC), (ii) adapt the initialization and termination of the implemented devices (e.g., PCI Express device) in the Linux kernel so that they can be assigned to the secure world[21], and (iii) implement a simple FTL to allow us experiment with real hardware. From our perspective now, we believe that we made the right choice; implementing hardware-assisted encryption would not have resulted in a better understanding of the implications of using split-enforcement. We consider this a separate problem that we can address better now that we understand the architectural impact of our design. For our prototype we have implemented the cryptographic locking in a specific file system, ext4. At a file system level, encryption is done at page level, which matches flash blocks. This means that decryption primitives in TSM would not need to change when hardware-assisted encryption is supported. While from a security perspective, the solution we have implemented is not as resilient as hardware-assisted encryption, it has allowed us to experiment with split-enforcement. In fact, the advent of next generation storage I/O devices supporting *Software Defined Storage (SDS)* is an ideal framework for trusted storage experimentation. The lessons learned from our prototype implementation will definitely facilitate the process of implementing hardware-assisted encryption on this new class of devices.

Memory Locking. The second shortcoming comes from the fact that TrustZone does not support full virtualization of the non-secure world. As we have described above, the non-secure world freely manages all its virtual address space, and it is not possible to virtualize the MMU to mediate I/O operations. The proposed solution is using ARM virtualization extensions to implement a thin hypervisor that allows to mediate memory accesses.

While these extensions are present in all ARMv8 architectures (e.g., A-53 [27], A-57 [28]) and in ARMv7 processors such as the Cortex-A15 [22], they are not present in the Cortex-A9 [21]. This is a problem since Zynq only features Cortex A-9 processors. There are two solutions to this problem. First, we can change development board. Indeed, most our contribution is platform independent; and the only platform specific fragments relate to the use of TrustZone registers and APB peripherals. Porting these parts to a different target platform does entail an engineering effort. However, the gained experience when implementing support for Zynq will certainly serve to move faster in a new port. In this way, we do not consider that porting to a new target platform would be a big issue. On the other hand, finding one that offers as much TrustZone support as Zynq is difficult. TEE frameworks such as SafeG TOPPERS and Open Virtualization directly support it (thus, providing reference implementations that are very useful to port drivers to other frameworks), and TrustZone configuration registers are public [298]. This is not the case for other target platforms at the time of this writing: NDAs

[21] Note that a device such as PCI Express presents the same issues than the ones described in Section 7.2 for other secure devices

are necessary, and collaborations with educational institutions is not always prioritized[22].

Second, we could make use of the FPGA in the Zynq Zc702 to implement a soft processor supporting virtualization to implement the untrusted area. However, Microblaze[23], the soft processor designed for Xilinx's FPGA, does not support at the moment trap-an-emulate instructions. Besides, this solution would limit our work to a very specific target platform, and a very specific architecture.

At the time of this writing, the most sensible solution is to advocate for the necessity of full virtualization in a future revision of TrustZone. This will benefit the embedded community, where TrustZone is already used today as a thin hypervisor, and it would specially benefit the security community, since approaches like split-enforcement could be applied to provide better run-time security. In a more practical sense, if we had to move to a different platform, probably Nvidia Tegra 4[24], featuring 4 + 1 Cortext-A15 CPU cores would be our choice. The fact that Nvidia has publicly released their own TEE framework, Nvidia TLK, points to a compromise to openly support TrustZone in their boards. Here, we also expect next generation silicon coming from all manufacturers to feature ARMv8 processors, thus allowing us to experiment with the presented hypervisor approach.

Regardless how memory operations are trapped, a common need is to redirect memory accesses to a trusted module that enforces memory policies. Such module would support CPI's *safe zone* in order to guarantee the integrity of *Trusted Memory Enclaves*. At the time of this writing however, CPI is still very new and we are still in the process of understanding the implications of incorporating it to our Trusted Cell architecture. Still, we have defined the framework to experiment with CPI and memory locking in the future.

In the secure area, we use shared memory regions, such as the ones described for storing *Secure Containers* in Section 8.2.1.1, in order to give the untrusted area access to unencrypted sensitive data. Each shared memory region defines a trusted memory enclave. In this way, the secure area allocates a trusted memory enclave and maintains its address pool in a mapping table to identify them. Shared (virtual) memory is allocated continuously to facilitate the mapping procedure. At this point, sensitive data is unencrypted and copied to the trusted memory enclave, which is directly accessible to the untrusted area. This part of the design follows the security requirements established for split-enforcement. Pages are encrypted whenever leaving the trusted memory enclave.

In order to experiment with memory locking, we implement a set of hypercalls in the commodity OS to trigger calls to the secure area before a memory access is served. These calls are equivalent to the ones that a hypervisor in any of the two approaches described above would issue. In the secure area, when the memory access is forwarded, the memory address is evaluated in TUM: if the memory access satisfies the device usage policies, the request is returned to the untrusted area. Otherwise, an exception is issued. In order to test this procedure, we enforce a simple policy where memory copies are not allowed.

[22]Here, we would like to highlight that our experience with ARM, and specially Xilinx has been very positive. Both have acknowledged our work and helped us with technical support, direct contact, and also hardware. We are thankful, and do not take such level of support for granted.

[23]`http://www.xilinx.com/tools/microblaze.htm`

[24]`http://www.nvidia.com/object/tegra-4-processor.html`

It is clear that this proof-of-concept implementation does not align with split-enforcement requirements. First, a formal method to verify point integrity such as CPI is not implemented. Second, the hypercalls added to the commodity OS can be bypassed if the kernel is compromised. In other words, no mechanism to lock memory is enforced, as it is the case for data at rest and peripherals. Still, implementing this early prototype for trusted memory enclaves has allowed us to understand the challenges that we need to address in order to provide a split-enforcement implementation for memory locking. Namely, (i) we need a virtualized environment that allows to trap-and-emulate memory accesses, (ii) we need to design a thin hypervisor that is able to mediate memory accesses by either forwarding to the secure area or directly implementing the part of TUM that contains memory policies, and (iii) we need an efficient way to represent trusted memory enclaves and their memory pools in order to minimize latency when evaluating memory addresses stemming from the untrusted area. Implementing memory locking to enable split-enforcement for sensitive data in motion in the untrusted area is a necessary step for future research.

8.4 Conclusion

In this Chapter we have presented the design and implementation of a Trusted Cell, i.e., a distributed framework that leverages the capabilities of a given TEE to provide trusted services. We implement four trusted modules to support these trusted services: *Trusted Security Module (TSM)*, *Trusted Integrity Module (TIM)*, *Trusted Usage Module (TUM)*, and *Trusted System Primitive Module (TSPM)*, and implement two trusted services on top of them: a trusted storage solution and a reference monitor.

We believe that our design and implementation of a reference monitor and a trusted storage solution using the Trusted Cell framework raise the bar for the current state of the art in run-time security. Moreover, it proves that it is possible to relax some of the trust assumptions that have persisted in the security community over time. While we have not implemented all the components for the locking mechanisms in split-enforcement, we do have argued the different alternatives, presented their design - both conceptually and technically -, and outlined realistic expectations. In fact, we have taken the hardware platform and TEE framework combination that was available at the time we started this research to the limit of its possibilities. Whenever our experimental platform has been a limitation, we have implemented a proof-of-concept mechanism in order to evaluate our hypothesis. In these cases, security premises where not satisfied, but the process allowed us to understand system limitations to choose a better environment in future iterations. Altogether, our work have exposed problems that affect the embedded, virtualization, systems, and security communities equally, and defines a solid roadmap for future research.

The solution we propose for trusted storage focuses on guaranteeing the confidentiality and integrity of sensitive data while reusing an untrusted I/O stack. What is more, it integrates with the commodity OS, allowing service providers to make use of trusted storage primitives without restricting the scope of their services. Indeed, with the aid of our reference monitor, sensitive data can leave the trusted area and directly be processed by these services. This contrasts with VPFS [285] - which we consider the most comprehensive solution for trusted

storage at the moment -, where sensitive data is processed uniquely in the (equivalent to the) trusted area. In our approach, complex untrusted applications can make use of our trusted storage capabilities while having the same level of confidentiality as in VPFS[25]. Integrity is provided by the usage policies that control actions to sensitive assets. We have left availability and durability out of the scope of our research given the security constrains we have set, which prevents us to rely on the I/O stack in the commodity OS. Here, the advent of I/O devices supporting *Software Defined Storage (SDS)* represents huge opportunity. Since these new I/O devices feature a complete execution environment based on ARM SoCs, they can support a TrustZone-based TEE Trusted Cell component dedicated to trusted storage. Such a component would not only guarantee confidentiality (TSM) and integrity (TIM) of sensitive data, but could also provide availability and durability by enforcing usage policies to mediate I/O operations, implementing redundancy techniques, or generating continuous backups.

The solution we propose for run-time security assumes that an attacker can gain full control of the untrusted software stack, including the kernel. This sets our approach apart from existing work on run-time security. Recall that our split-enforcement solution locks sensitive assets so that the untrusted area requires the reference monitor to unlock them. In this way, we reuse the existing untrusted stack for accessing peripherals and maintain a low TCB. It is worth mentioning that we cannot find any system primitive used by existing approaches that we cannot provide by separating the policy engine from the monitored environment. An interesting topic for future work is to explore how existing run-time solutions (such as the ones described in Section 3.3) could be implemented using split-enforcement.

All in all, these two trusted services together provide the necessary run-time security primitives to protect sensitive data both at rest and in motion, while allowing it to be directly managed by untrusted applications in the commodity OS. This enables service providers to implement services based on sensitive data, while it allows users to enforce usage control policies to ensure the confidentiality and integrity of their data.

[25] We also use AES in CBC mode with unique initialization vectors (IVs).

Chapter 9

Certainty Boot

At this point, we have presented all the run-time security mechanisms that allow to implement split-enforcement. Indeed, the combination of the Linux support for TrustZone and the TEE presented in Chapter 7, and the Trusted Cell presented in Chapter 8 form the hardware and software framework that can leverage run-time security as we presented it in our hypothesis (Chapter 6). In this Chapter, we use this framework and the concept of split-enforcement and apply to it a concrete use case: increasing software freedom in mobile devices. While the mechanisms we discuss in this chapter can be directly applied to other ARM power devices we have discussed throughout this thesis (e.g., next generation I/O devices, ARM servers), we consider that mobile devices exhibit some unique characteristics that allow us to center the discussion on privacy, and the role that technology plays on protecting it.

9.1 Context

Unlike other personal devices such as laptops, mobile devices are designed today to run a single Operating System (OS). Typically, Original Equipment Manufacturers (OEMs) lock their devices to a bootloader and OS that cannot be substituted without invalidating the device's warranty. This practice is supported by a wide range of service providers, such as telecommunication companies, on the grounds that untested software interacting with their systems represents a security threat[1]. In the few cases where the OEM allows users to modify the bootloader, the process is time consuming, requires a computer, and involves all user data being erased[2]. This leads to OEMs indirectly deciding on the functionalities reaching the mainstream. As a consequence, users seeking the freedom of running the software that satisfies their needs, tend to root their devices. However, bypassing the security of a device means that these users lose the capacity to certify the software running on it. This represents a risk for all parties and services interacting with such a device [189, 237].

This topic has been widely discussed by Cory Doctorow in his talk *Lockdown: The Coming*

[1] `http://news.cnet.com/8301-17938_105-57388555-1/verizon-officially-supports-locked-bootloaders/`

[2] `http://source.android.com/devices/tech/security/`

Civil War over General Purpose Computing [95]. Here, he argues that hardware security platforms such as Trusted Platform Module (TPM) (Section 2.2) have been misused to implement what he calls the lock-down mode: *"Your TPM comes with a set of signing keys it trusts, and unless your bootloader is signed by a TPM-trusted party, you can't run it. Moreover, since the bootloader determines which OS launches, you don't get to control the software in your machine."*. Far from being taken from one of his science fiction dystopias, the lock-down mode accurately describes the current situation in mobile devices: users cannot always modify the software that handles their sensitive information (e.g. pictures, mails, passwords), control the hardware peripherals embedded in their smart-phones (e.g. GPS, microphone, camera), or connect to their home networked devices (e.g. set-top boxes, appliances, smart meters). This raises obvious privacy concerns.

In the same talk, Doctorow discusses an alternative implementation for hardware security platforms - the certainty mode -, where users have the freedom to choose the software running in their devices, and the *certainty* that this software comes from a source they trust. What is more, he envisions the use of context-specific OSs, all based on the user's trust. The issue is that the trust that a user might have in a given OS, does not necessarily extended to the third parties interacting with it (e.g., private LANs, cloud-based services, etc.).

In this chapter we present a novel approach [134] to allow users choosing the OS they want to run in their mobile devices while (i) giving them the certainty that the OS of their choice is effectively the one being booted, and (ii) allowing running applications to verify the OS at run-time. Put differently, we extend Doctorow's certainty mode idea to all the parties interacting with a mobile device. We denote this approach *Certainty Boot*. While modern Unified Extensible Firmware Interface (UEFI) x86-powered platforms support the installation of new OSs and their corresponding signing key by means of the BIOS, this is not the case for mobile platforms powered by ARM processors, where neither BIOS support, UEFI[3], nor a TPM[4] are present. The same applies for user space applications doing integrity checks on the running system. In order to address this issue in mobile devices we propose a *Two-Phase Boot Verification* of the boot process: in the first phase, boot components are verified and logged in the same fashion as trusted boot; in the second phase, the boot trace can be checked by running applications in order to verify the running OS. Enforcing policies based on these checks is done by means of split-enforcement as implemented in the Trusted Cell. Note that both logging the boot sequence boot-time and OS verification at run-time are a *Run-Time Security Primitives* leveraged by the Trusted Cell. They are indeed *Trusted Services*. The key, as we have argued several times already, is guaranteeing that these run-time security primitives are necessary for the correct operation of the untrusted area, in order to invalidate attacks targeting the communication interface leveraging trusted services.

We base the security of our design in hardware security extensions present on a wide range of mobile devices: ARM TrustZone as a Trusted Execution Environment (TEE), and Secure Element (SE) as a tamper-resistant unit. This is the same combination we have used for the Trusted Cell architecture (Section 8).

[3]Linaro is working on porting UEFI to ARM. However this work is still very much in progress (`https://wiki.linaro.org/ARM/UEFI`)

[4]As mentioned in Section 2.2, there have been attempts to build a TPM based on TrustZone capabilities. However, given that TrustZone is not tamper-resistant, a general solution is far from being adopted.

9.1.1 Example Scenario

The solution we propose in this chapter enables service providers with security concerns to adapt an OS to meet their particular security requirements. This customized OS is offered by a trusted party; therefore we call it a **certified OS**. How trust is defined here is up to the service provider: certification, formal verification, legal responsibility, etc. For the trusted area, the certified OS is still untrusted; thus it is considered as any other commodity OS. Such a certified OS could restrict the installation of applications, filter network usage, or arbitrate access to certain hardware. We consider this is not a violation of user's freedom since it is explicit that such a certified OS is used to enable some specific services, most probably in an enterprise context. The idea is that users are able to change to the OS of their choice to handle their personal data. This is a *clean cut* between what is personal data and what is not. By enabling users to exchange OSs that can be certified according to their criteria (again, users decide what they trust), we enable services and organizations to establish restrictions concerning the software that interacts with their systems, while preserving the user's right to choose - and certify - the software handling their personal information.

Bring Your Own Device (BYOD). One clear practical application for a certified OS is the BYOD problem. This refers to users wanting to interact from their personal mobile devices with their company IT services. Examples include VPN connections, access to Enterprise Resource Planning (ERP) systems, or company email. Even when companies put limits to BYOD as a way to prevent internal, and probably unintentional attacks against their infrastructure, research shows that employees force their devices into their enterprise services [87]. Given the heterogeneity of the current mobile device OS landscape, this introduces a big overhead for system administrators. Concrete challenges include porting services to different platforms, having to deal with platform-specific security threats, or increasing the complexity of the enterprise's LAN. This is a product of the devices connecting to these services not being trusted. By using one customized OS that is preconfigured to interact with all company services, enterprises could safe time and money while increasing the security of their hosted services. When employees are not using their company services, they can switch to an OS they trust to handle their personal information. In this way one single device can be used in different contexts without forcing trade-offs on either side. The limitation is that it requires the user to switch OS.

9.2 Architecture

Figure 9.1 depicts how hardware and software components interact with each other in the two-phase boot verification. While in this case we specify the components present in the two-phase boot verification, the underlying architecture is the same as in the Trusted Cell (Figure 8.1). In normal operation both the (*Rich Execution Environment (REE)*) and the (*Trusted Execution Environment (TEE)*) are booted. This is, the signatures of the mobile OS and the OS bootloader (OSBL) have been validated by the SE. Note that the verification process is carried out in the secure area (i.e., TEE), consisting of its bootloader (TEEBL)

and an OS (TEE OS), and the SE as root of trust. Additionally, we assume the manufacturer bootloaders, first (FSBL) and second stage (SSBL), also as root of trust components. Applications running in the REE can be installed and executed without restrictions just as we are used to see in current mobile OSs. We do not make any assumptions regarding the trustworthiness of these applications; they are untrusted applications. However, we consider that the secure tasks running in the secure area are trusted. We describe the communication between the trusted and the untrusted areas in more detail in Section 9.2.3.

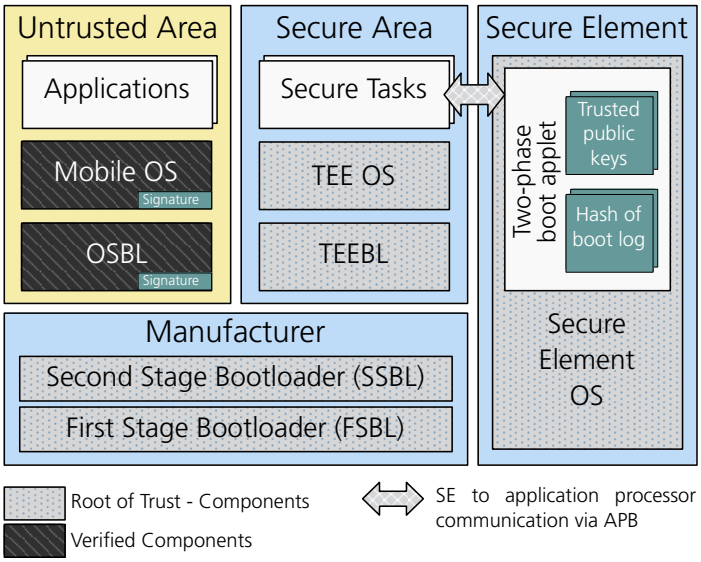


Figure 9.1: Architecture for *Two-Phase Boot Verification*.

As described in the Trusted Cell architecture (Section 8.1), the SE is configured as a secure peripheral and therefore only accessible from the secure area. Note that while the SE is not a peripheral in itself, it is connected via the APB (e.g., I2C, etc.), and therefore treated as such. In this way, untrusted applications make use of secure tasks to access the SE. As a consequence, if the secure area becomes unavailable, untrusted applications would be unable to communicate with the SE. The secure area becoming unavailable could be a product of a failure, but also a defense mechanism against unverified software. If the OS image cannot be verified, the OS would still be booted, however the TEE will not respond to any attempt of communication. In this case we give up on availability in order to guarantee the confidentiality and integrity of the sensitive data stored in the SE (e.g., corporate VPN keys stored in the tamper-resistant hardware).

Finally, the secure area enables the SE to makes use of untrusted storage (e.g., SD card, flash memory), as enabled by the trusted storage solution presented in Section 8.2.1. One utility is to allow the SE to use untrusted storage to handle OS certificates, while preserving their integrity and confidentiality. This gives users the option to add additional trusted public keys to the SE when the first-phase verification failed (see Section 9.2.3 for details). It also enables to store several OS images in a common SD card, not running into space issues with the SE. The trade-offs are the same as we discussed in Section 8.2.1.

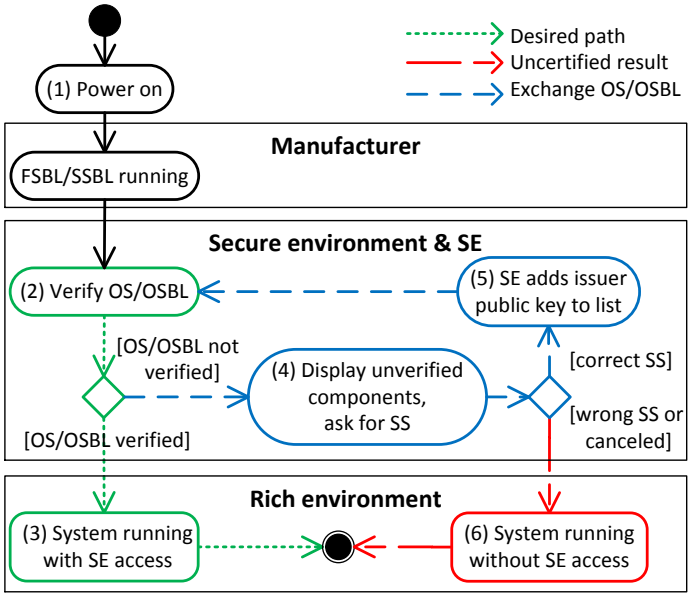


Figure 9.2: Activity Diagram for *Two-Phase Boot Verification*. Transitions show the desired path (dotted), an uncertified result (long-dashed) and steps used for exchanging OS (dashed). "SS" stands for shared secret.

9.2.1 First Phase Verification

The first phase verification is responsible for verifying and starting the TEE[5] and the commodity OS. Besides this, the first phase also enables the user to install a customized OS and flag it as trusted. All steps necessary to verify the boot process, are depicted in Figure 9.2. The numbers in the diagram correlate with the numbers in the following enumeration. Borders indicate which component is responsible for each step.

1. The user presses the power button. FSBL, SSBL, and TEE are started.
2. The TEE attempts to verify OS/OSBL by sending their signatures and the hash of the images to the SE. The applet in the SE attempts to verify the signatures with all trusted public keys of OS/OSBL issuers and compares it to the received hash. The result of this operation is reported to the TEE. In addition, the hash of the booted OS for the second-phase is stored in the SE.
3. Once the OS/OSBL are verified, the OS is booted with full access to the SE through the TEE. This is the end of the expected course of actions.
4. If the verification of OS/OSBL (step 2) fails, a message, explaining which component is unverified, is displayed. Now the user can choose to either certify the OS/OSBL by

[5]In this description we will refer to the secure area as TEE since we want to emphasize the process of booting the TEE framework that supports the secure area

entering the shared secret and provide an issuer certificate, or continue to boot the uncertified OS/OSBL. For further details see Section 9.2.3.

5. If a legitimate user (authenticated by shared secret) flags an OS/OSBL as trusted, the SE adds the public key of the issuers' certificate to the list of trusted keys and continues to verify the newly added OS.

6. If the user enters a wrong shared secret or cancels the operation, the uncertified OS/OSBL is booted without access to the SE and a message informs the user about the uncertified state. This is the end of the uncertified course of actions, where the user can still use the device (uncertified OS/OSBL is booted), but due to missing verification, access to the secured data is denied by the TEE.

This architecture ensures that only a combination of verified OS and OSBL are granted access to the SE. If one of the components is not verifiable, the device can still be used (uncertified OS is booted), but access to the SE is denied. The secure area guarantees this. Throughout the whole first phase all executed commands are logged in the SE by maintaining a hash chain. This approach is similar to trusted boot (see Section 4) and enables the second phase verification. The process of installing custom OS is explained in Section 9.2.3.

9.2.2 Second Phase Verification

In the second phase verification, untrusted applications can verify the system in which they are running before executing. To do this, untrusted applications make use of secure tasks to validate the running OS by checking the boot traces in the SE. These secure tasks are not application-specific, but *Run-Time Security Primitives* that any untrusted application running in user space can request to use. As mentioned, for the commodity OS verifying the running system is a trusted service.

As in the case of our reference monitor (Section 8.2.2) or our trusted storage solution (Section 8.2.1), trusted services are exposed to the untrusted area by means of the *REE Trusted Cell*. At boot-time, the secure area generates a secure device tree (Section 7.2) where it explicits which trusted services it supports, as defined by the trusted modules, secure tasks, and specifications it implements. Note that in this case however, split-enforcement is slightly different: it is the untrusted application that requires that service to work properly, since it executes in a certified OS. This enables trusted providers to place certificates in the secure area to verify the device that connects to their services, thus defining the enforcement part; untrusted applications necessarily need to go through the secure area to operate correctly. All the benefits of split-enforcement are maintained (Chapter 6): Low TCB, not code replication by reusing untrusted software stacks, fast dissemination of security critical operations, etc.

By enabling untrusted applications to verify the running OS using run-time security primitives, they can pass the list of hashes they trust and a parameter defining the hash algorithm used to calculate them (e.g., sha, DJB2, md5). This allows for several hashing algorithms being supported and new ones being easily introduced, satisfying different application requirements. When the secure task executes, it communicates with the SE to obtain the

hash of the OS image calculated at boot time, and compares it with the hashes trusted by the untrusted application calling it. Applications can also verify the boot traces to check that all the components they require (e.g., biometric sensor) have been correctly initialized at boot time. As a result, untrusted applications are able to make decisions at run-time depending on both the OS and the available peripherals. Usage policies emerge naturally from this architecture. But what is more interesting, these usage policies have nothing to do with the *Device Usage Policy*; they modify how the untrusted application interacts with remote services. How service providers implement this is not defined by us. A possibility is that they install their own trusted modules to leverage these trusted services and make use of the usage device policy and *Application Contract* to enforce the usage policies the defined at run-time, but this is not necessary. The trust model that service providers implement is theirs to enforce; we provide the framework. Note that here we take the perspective of the mobile device, where services are remote, thus external. From the point of view of the device providing those remote services, the Trusted Cell architecture ensures the enforcement of its own device usage policies. An obvious integration of these two Trusted Cells working together is assuming mutual distrust between the devices, and using the Trusted Cells to provide mutual certification and execute remote trusted services. This is out of the scope of this thesis, and an interesting topic for future research.

Since the SE is configured as a trusted peripheral, untrusted applications cannot directly communicate with it; they need to do it through the secure area. Additionally, the SE signs the retrieved hashes using its own private key in a similar manner as the TPM using the Attestation Identity Key (AIK) (Section 2.2). To distribute the corresponding public keys of the SE, an infrastructure similar to other public-key infrastructures is required (e.g., openPGP).

9.2.3 Exchange OS and Boot Loader

We share the concern pointed by Doctorow that current secure boot implementations necessarily lock devices to a specific OS chosen by the OEM. In order to avoid this in our architecture, we propose the **configuration mode**. TEEs are suitable candidates to implement this mode, since they support features for secure user interactions (e.g., biometric sensors attached to the trusted area). The sequence of actions for the configuration mode starts by the user flashing a new mobile OS or OS bootloader (e.g., using uboot) with a signature that is not trusted by the platform (public key of the OS issuer is not in the list of trusted keys). As depicted in the transition from step 2 to 4 in Figure 9.2, the OS will not be booted in that case.

The user will now be given the possibility to either manually verify the signature of the new mobile OS or cancel the process within a secure task of the TEE. In case of a requested manual verification, the user will be asked to point to the certificate of the OS issuer on the untrusted memory (e.g., SD card) and enter a shared secret within the secure UI of the TEE (step 4 in Figure 9.2). This shared secret could be a PIN or password that has been shipped together with the SE. With an appropriate secure channel protocol, the user will be authenticated to the SE and a secure communication between TEE and the applet will be established. If the user does not want to verify the OS, the system would still be

booted without access to the sensitive data in the SE (step 6 in Figure 9.2). After successful authorization, the secure task sends the public key of the OS issuer to the applet, where it will then be added to the list of trusted keys (step 5 in Figure 9.2). If users do not have access to the certificate, or do not want to completely trust the issuer, they can also exclusively sign the specific OS instance with the private/public key pair of the SE. This implies that not all releases of the issuer are trusted. In this case, the secure task will create the hash of the OS image and let the applet sign it with its own private key. After this, the SE will verify the signature of the mobile OS again (step 2 of Figure 9.2). This time, the applet can verify the signature with the public key and therefore confirm the validity of the image (if it was not tampered). Attacks attempting to flag an OS as trusted will fail as long as the shared secret remains unknown to the attacker. An adversary could also try to manipulate the certificate which is stored in the untrusted memory. However, as the TEE has full network capabilities, it can verify the certificate validity with a correspondent public-key infrastructure, such as the web of trust from PGP.

While the configuration mode gives the user the possibility to easily change and manually verify the source of the mobile OS, we are aware that exchanging and verifying the TEE is not supported within our architecture. We consider the TEE as a system-dependent software to support hardware security extensions. In this way, we can see the benefits of exchanging it to provide security updates, however, we cannot see the benefits it brings with respect to user's freedom. Exchanging TEEs is not considered and we leave it as a topic for future research. Here, the repercussions of limiting the root of trust and adding an additional security layer are a significant impact to out architecture.

9.3 Conclusion

In this chapter, we describe the concept of a *Two-Phase Boot Verification* for mobile devices. Our goal is to give users the freedom to choose the OS they want to run in their mobile devices, while giving them the certainty that the OS comes from a source they trust. We extend this certainty to running applications, which can verify the OS in which they are executing. This run-time verification is a trusted service that allows untrusted applications to define their own usage policies. We believe that this is a first step towards an architecture for mobile devices where security is leveraged without locking devices to specific software. This allows users to switch OSs depending on their social context (e.g., work, home, public network), which we contemplate as a necessary change in the way we use mobile devices today. One device might fit all sizes, but one OS does definitely not.

One of the core limitation of our approach is the fact that the OS must be switch to provide a *clean cut* between the trusted services leveraged by each OS available to a device. We believe that our two-phase boot verification would be a natural complement to multi-boot virtualization architectures like *Cells* [15], leveraging an architecture where multiple verified OSs could run in parallel, therefore facilitating the process of switching OSs. Moreover, we believe that two-phase boot verification together with the Trusted Cell in a virtualize environment can allow a more secure sandboxing architecture, where security-critical applications can virtualize their own environment and provide their own trusted services. This requires

that problems such as run-time installation and attestation of trusted modules, as pointed out in Section 8.1.1, are solved. Still, the combination of the run-time security proposals found in this thesis can enable such services.

Finally, since our approach is based on off-the-self hardware, it can be implemented in currently deployed mobile devices, and other ARM-power devices where a tamper-resistant unit can be attached to the Advanced Peripheral Bus (APB).

Part III

Evaluation

In Part III, we present our evaluation. We split this presentation into an analytical and an experimental evaluation. In the analytical evaluation (Chapter 10) we provide an exhaustive analysis of split-enforcement and its materialization in the Trusted Cell in terms of security and requirement compliance. This analysis allows us to evaluate split-enforcement using the experience of building the generic TrustZone driver and the Trusted Cell, only without the distraction of considering the platform limitations we have encountered in the process of building them.

In the experimental evaluation (Chapter 11) we focus on our experimental platform (hardware and software) and how it performs when implementing split-enforcement with the generic TrustZone driver and the Trusted Cell. The objective is to discuss about the overhead introduced by mediating system primitives through a decision point in the trusted area. Since the cost of associated with the decision point is orthogonal to split-enforcement, this experimental evaluation will help us, and others, to make better design decision when new hardware appears that can better support split-enforcement.

Chapter 10

Analytical Evaluation

In this Chapter we provide an analytical evaluation of our contributions (Chapters 7, 8, and 9). We first provide an exhaustive security analysis for each of them. Then, we look at the design requirements that we established in the exposition of *Split-Enforcement* (Chapter 6), and study how they are met in the Trusted Cell.

10.1 Security Analysis

We organize our security analysis in a bottom-up fashion; from the hardware to the upper software layers. In this way, we present (i) the secure hardware that we have used in our experimentation (i.e., ARM TrustZone and SE), (ii) the generic TrustZone driver we implement in the Linux kernel, (iii) our Trusted Cell prototype, and (iv) our proposal for certainty boot. Since the bottom line of *Split-Enforcement* is that the untrusted area can be fully compromised, an attacker could design intricate attacks against the communication interfaces between the trusted and the untrusted areas. Our design advocates for a high integration between the two areas in order to support innovative services, and this inevitably comes at the cost of exposing components in the trusted area. Still, we will see that we maintain the assumption that the untrusted area (i.e., untrusted applications and commodity OS) is fully untrusted, and the fact that it is compromised does not affect neither the confidentiality nor the integrity of sensitive assets. We start by giving an overview of types of attacks and attackers.

10.1.1 Types of Attacks

Defining tamper-resistance in a quantitative way is difficult. Not only because there are no obvious metrics for such definition, but because the level of tamper-resistance of a given design or technology is tightly coupled with *known* attacks against it. In this way, we will describe the security of the different designs we have presented in this thesis as a function of the attacks - we know - they can resist against. This makes our security analysis perishable;

if a new attack is presented tomorrow against TrustZone for example, the base for our security analysis would instantly change. Tamper-resistant promises should then be consequently adapted. Note that this is unavoidably for any technology and proposal related to security. For this reason, we avoid the unnecessary experimental evaluation with current user and kernel space *root-kits* [52, 58, 232], since, not expecting the locking mechanisms present in split-enforcement, they will fail to compromise sensitive assets. This result would be misleading, and therefore we avoid including it in this thesis. Note that we have indeed experimented with root-kits and built split-enforcement around known kernel root-kit techniques: system call modification, overload of system call return values, side-channel operations. etc.

In order to position the attacks that we will describe in this chapter, we classify attacks in four categories: hack attacks, shack attacks, lab attacks, and fab attacks. While there is no reference classification for types of attacks in the security community (different works present their security analysis in different formats, using different metrics and roots of trust), we believe that this classification, inspired on other security analysis [29, 165], covers the whole spectrum of possible attacks to a target design.

- **Hack Attacks** are attacks limited to software. Typically, these attacks are triggered by an user approving the installation of a piece of software that then executes the attack. This is either because the malware pretends to be a piece of the software that the user does want to install, or because the user does not understand the warning messages displayed by the operating environment. Examples of hack attacks include viruses and malware.
- **Shack Attacks** are a low-budget hardware attack. Attackers have physical access to the device, which mean that they can attempt to connect to device using the *JTAG*, boundary scan I/O, and built-in self test facilities. but they lack the knowledge or equipment to carry out an attack at an integrated circuit level. Examples of shack attacks include forcing pins, reprogramming memory devices, and replacing hardware components with malicious alternatives.
- **Lab Attacks** are comprehensive and invasive hardware attacks. Attackers have access to laboratory equipment and the knowledge to perform unlimited reverse engineering of a given device (e.g., reverse engineering a design, performing cryptographic key analysis). The assumption should be that a device can always be bypassed by a lab attack given enough time and budget.
- **Fab Attacks** represent the lowest level of attack. Here, malicious code and/or logic is inserted into the layout of an integrated circuit in the fabrication plant. Lab attacks also include intended bad designs and embedded backdoors. Circuitry fabricated in the chip cannot be easily detected, not even using chip validation techniques.

In order to put attacks in context, we also provide a classification for attackers. We adopt the taxonomy proposed by IBM [89], which distinguish between three classes of attackers: intelligent outsiders, trained insiders, and funded organizations.

- **Intelligent Outsiders** are remote attackers who lack specific knowledge on the system. They might have access to moderately sophisticated equipment. These attackers try

to reproduce hack and simple shack attacks published in the Internet by a technical expert, rather than attempt to create new ones.

- **Trained Insiders** are technical experts, highly educated, with experience and access to sophisticated tools. They are considered trusted, possibly employed by the company developing the device subject of the attacks. Their knowledge of the system varies, but it can be assumed that they have access to the information describing it (e.g., secret information, detailed designs). They seek to discover and share new class attacks.
- **Funded Organizations** represent organizationally founded teams of trained attackers. They are capable of carrying out sophisticated attacks by means of advanced analysis tools. They are capable of designing new and innovative attacks exploiting the most insignificant weaknesses. Examples of such organizations include organized cybercrime, cyberactivist groups, or governments.

10.1.2 Hardware Protection Baseline

Before we provide the security analysis for our contributions, we will state where the protection baseline lays for the hardware that we are using, in terms of the attacks it can protect against. Understanding hardware limitations was key to choose the secure hardware composing our Trusted Cell prototype, so that we could implement split-enforcement while maintaining our trust assumptions. Moreover, the secure hardware combination that we have chosen: TrustZone + Secure Element defines the protection upper bound for our Trusted Cell prototype independently from the software we have built on top of it.

Given that we are using commercially available secure hardware, fab attacks are out of scope. If the device implementing TrustZone's security extensions, or the SE are compromised at design- or fabrication-time then split-enforcement and the security-oriented implementation of our generic TrustZone driver, and our Trusted Cell prototype should also be assumed compromised. In this way, we trust Xilinx to provide a non-compromised TrustZone design in their Zynq-7000 ZC702 SoC, and we trust NXP to provide a non-compromised A7001 chip, which we use as Secure Element.

10.1.2.1 ARM TrustZone

According to ARM [29] *TrustZone technology is designed to provide a hardware-enforced logical separation between security components and the rest of the SoC infrastructure.* In terms of TrustZone's specific mechanisms, this means that TrustZone enforces in hardware a separation between the components extended by the NS bit (Section 2.4) and the rest of the components in the device.

Since TrustZone does not provide hardware mechanisms to protect the NS bit neither in terms of size, tamper counter-measurements, nor tamper-evidence it is safe to state that TrustZone is not tamper-resistant. This means that lab attacks are out of scope. Moreover, since the NS bit is an extension to the AMBA3 AXI system bus, advanced shack attacks targeting to probe or force pins, trigger specific sequences of interrupts (IRQ, FIQ) and

external aborts, or combinations of them could compromise the communication with secure peripherals and even enable access to secure memory and secure code execution.

The only realistic assumption is that an attacker with access to a TrustZone-enabled device can (i) bypass the TrustZone security extensions and take full control of the software in the secure area, (ii) steal the sensitive data (e.g., secrets, encryption keys) that are stored in it, and (iii) misuse other sensitive assets to get further access to other services (e.g., mobile banking), devices (e.g., by reusing certificates or encryption keys), or information (e.g., by superseding the device owner's digital identity).

10.1.2.2 Secure Element

Secure Elements, as other smart cards, are considered tamper-resistant. This means that they can resist hack, and shack attacks, as well as a number of lab attacks. While attacking the interface is always possible, the fact that the SE is only accessible from the trusted area gives us an extra security protection in terms of exploiting the SE interface by means of hack attacks; an attacker would have to succeed in attacking TrustZone before targeting the SE. Moreover, since SEs have a narrow interface to access sensitive data, inputs are easy to verify. Still, the SE does not have a standard way to authenticate where the instructions it receives come from. This is an open issue in the security community.

The assumption should be that with enough time and money, a trained attacker with physical access could steal all the secrets in the SE by means of a lab attack. Still, no known hack and shack attacks are known that can systematically be used to target SEs.

10.1.3 Attacks against TrustZone Driver

There are two TrustZone components that are exposed to the untrusted area, and therefore are subject to being compromised: the generic TrustZone driver and the secure monitor. While these two components are tightly related, we explain them separately, since their location in the untrusted area responds to two different attacks vectors. A third attack vector that we cover is directly compromising the secure area without using the interfaces exposed to the untrusted area.

10.1.3.1 Attacks against User Space Support

An attacker that can execute code on the device - even without root privileges - might have access to the *REE Trusted Cell* and the *tz_device* (Figure 8.2). While these two components only contain the interface, an attacker could still try to force erroneous inputs. The objective would be to exploit errors in the kernel, specially in the TrustZone driver, such as buffer overflows, buffer over-reads, or other code bugs that trigger undefined behavior in the kernel. As showed in Table 7.1, the generic TrustZone driver is fairly small (4030 LOC in its current status), which allows us (and the rest of the kernel community) to inspect the code manually.

We have also made use of *splint*[1] a static analysis tool for C that allows to prevent bugs and bad practices, and *checkpatch* [167], a Linux kernel patch to improve code style. Still, we assume that the interface can be exploited in the untrusted area. In our Trusted Cell prototype, as we will detail later in this chapter, we rely on TUM to enforce the *Device Usage Policy* and prevent sensitive assets to be leaked or misused. However, since the interface in the REE Trusted Cell is untrusted, there is no policies that can be enforced there. A full analysis of attacks against TUM is provided later in this chapter.

If an attacker obtains root privileges through a hack attack, she could compromise support both for user space applications and kernel submodules. If such support is compromised, a Denial of Service (DoS) attack against the trusted area would succeed. This is part of our assumption; the most obvious attack in a fully compromised area is to disable security extensions. Such an attack would affect the availability of sensitive assets but would not affect their confidentiality or integrity; encryption keys and integrity measurements never leave the secure area unencrypted.

10.1.3.2 Attacks against Secure Monitor

If an attacker obtains root privileges through a hack attack, or forces pins through a shack attack, she could compromise the secure monitor. The secure monitor is indeed TrustZone's most vulnerable component since it is in charge of switching between kernel and secure spaces. If compromised, illegitimate code could run while the processor is executing in secure mode. This involves prioritized access to all peripherals and memory.

For ARMVv7 architectures, the secure monitor is an implementation-specific software component. This has allowed bad designs reaching the market. An example is the attack reported in October 2013, affecting Motorola devices running Android 4.1.2. The attack reached the National Vulnerability Database and scored an impact of 10.0 (out of 10.0)[2], and it can be found in the attacker's blog[3]. The lack of a standard secure monitor implementation makes it impossible to make general comments about its level of tamper resistance. While we are not aware of any successful attack against Open Virtualization's secure monitor, that does not mean that such attacks are not possible. Formal verification, static analysis, or third party certification could increase the trustworthiness of those components, but again, it would only cover specific implementations. As in the case above, in our Trusted Cell prototype, we rely on TUM to enforce the *Device Usage Policy* and prevent sensitive assets to be leaked or misused. Still, this measure depends on the actual implementation of the secure world, and cannot be argued as a good defense for the TrustZone interface in the untrusted area.

A solution to this general issue is coming for ARMv8 processors. Here, ARM is providing an open source reference implementation of secure world software called ARM Trusted Firmware (ATF) [4]. This effort includes the use of ARMv8 Exception Level 3 (EL3) software [135] and a SMC Calling Convention [24], as well as a number of various ARM interface standards,

[1] `http://www.splint.org`
[2] `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-3051`
[3] `http://blog.azimuthsecurity.com/2013/04/unlocking-motorola-bootloader.html`
[4] `https://github.com/ARM-software/arm-trusted-firmware`

such as the Power State Coordination Interface (PSCI) and Trusted Board Boot Requirements (TBBR). This makes it possible that the secure monitor can be reused in different TEE frameworks, which reduces the secure monitor to a single point of failure. Here techniques such as formal verification or static analysis can help to improve the secure monitor's resilience, knowing that it is a system primitive, not a framework-specific component. Also, the fact that ATF executes in EL3 limits the scope of hack and shack attacks significantly. Organizations such as Linaro, Xilinx, and Nvidia are already contributing to its development and pushing for its adoption. In this way, we expect the secure monitor to become a common system primitive to different TEE frameworks.

10.1.3.3 Attacks against Secure World

Direct attacks against Trustzone's secure world are attacks against the TrustZone architecture, and are beyond our TrustZone driver in the untrusted area. These attacks could target (i) accessing secure memory, and (ii) executing code in the secure world. Given our architecture, and providing that the secure monitor is not compromised (see above), these attacks require physical access to the device. As also mentioned above (Section 10.1.2.1) TrustZone cannot protect against a lab attack, or a complex shack attack. Thus, if an attacker can force the NS bit, she could directly access secure memory and execute code in the highest privilege mode at her will. This would compromise all the sensitive assets that the secure world has direct access to. However, note that if TrustZone is assisted with other secure hardware such as a SE to store secrets (e.g., encryption keys, certificates), these would not be directly accessible to the attacker. In order to steal these secrets, and attacker would need to targeted the tamper-resistant unit (i.e., SE) specifically since compromising TrustZone does not entail any advantage.

10.1.4 Attacks against Trusted Cell

We analyze now the Trusted Cell. Note that while we have also maintained a low TCB when designing and implementing the Trusted Cell (Section 8.1), and used development tools such as *splint* or *checkpatch* to improve the quality of our code, we will keep our security analysis at the design level. In oder words, we analyze split-enforcement. We will still argue about the trusted services we have implemented, but we will not argue for our actual implementation. The fact that we have not used formal verification techniques to validate the Trusted Cell invalidate any argument that we can provide in terms of code correctness. Still, we can provide a complete analysis of what we have built, and how split-enforcement can be exploited[5].

In the Trusted Cell architecture (Chapter 8) we described how we intended to increase the level of tamper-resistance of the Trusted Cell by including a SE in our design. The reason behind this decision is that we want to guarantee the confidentiality and integrity of sensitive

[5]Note that this argument cannot be used for the generic TrustZone driver. Since it is an untrusted component that interacts with a whole untrusted stack, the correctness of our code (even if completely bug-free) would not make a difference in terms of trust. The Trusted Cell executes in secure space, which means that the compactness (i.e., low TCB) and correctness of this code is paramount.

assets at least against shack attacks and some classes of lab attacks. In this way, in the security analysis for the Trusted Cell we will look at the different types of attacks that can be designed against its different protection layers (i.e., untrusted area + TrustZone-enabled trusted area + SE), and what their impact would be on sensitive assets. We also introduce a set of counter-protection mechanisms that the Trusted Cell implements to limit its attack surface.

10.1.4.1 General Attacks against REE Trusted Cell

All communications with the *TEE Trusted Cell* are made through the *REE Trusted Cell*, which is untrusted. Attacks against the communication interface in itself involve the Linux LSM framework through the *LSM Trusted Cell*, which forwards secure system calls to the TEE Trusted Cell and through it to TSM, TIM, TUM, and TSPM. Hence, attacks to the LSM framework - and to the untrusted kernel in general - could bypass all (or specific) trusted modules forming the TEE Trusted Cell. Here, we do not make any assumptions regarding the privileges of an attacker. More explicitly, we assume a malicious *root* that can compromise any part of the untrusted area, e.g., using applications such as *ptrace* to control the execution of other legitimate applications, accessing kernel and physical memory (*/etc/kmem* and */etc/mem* respectively), or injecting kernel code by means of a loadable kernel module (LKM). We classify and describe these attacks in two classes: secure data path manipulation and DoS attacks. Here we discuss how these attacks affect trusted services in general. In the following subsections, when we analyze specific attacks against our trusted storage and reference monitor solutions, we come back to these attacks, especially secure data path manipulation, and concrete them for each trusted service. Note that in the case of targeting a specific trusted service we assume that the attacker is familiar with the accepted inputs taken by a trusted module and the expected outputs associated with them.

Secure data path manipulation. Examples of possible attacks include: bypassing the secure area, superseding the return of a secure task or trusted service, manipulating the internal state and/or memory of an untrusted application by using developing and debugging tools (e.g., by using ptrace), attempting a return-oriented programming (ROP) attack, buffer overflows, etc. As we have argued throughout the whole thesis, resisting these attack vectors is the motivation for split-enforcement. An attacker could prevent that secure system calls are forwarded to the secure area, modify an entry in the system call table, or overwrite the usage decision returned to the untrusted area. However, none of these attacks could unlock a sensitive asset. In order words, bypassing the secure area brings no benefit to the attacker since sensitive assets are locked (Section 8.2.2). This sets our work apart from other solutions, where the security of the whole proposal relies on a trusted component being executed. Split-enforcement locks all sensitive assets at boot-time so that using the secure area is a necessary step to access them. Indeed, it does not matter how much the untrusted kernel is modified, split-enforcement's first phase occurs in the trusted area, before the untrusted area is booted, and therefore it is out of its scope. All unlocking mechanisms reside in the secure area, which is also outside of the scope of the untrusted area.

An attacker could also attempt to hijack or compromise the secure data path by means of

ROP, return-to-libc, or Iago attacks [60]. The success or failure of these attacks depends strictly on the concrete trusted service and the level to which it reuses the untrusted software stack. We analyze these attacks separately for the two trusted services we have implemented in next section.

DoS against the secure area. Since TEEs are essentially based on a client - server architecture DoS attacks will always be possible. The design we propose guarantees integrity and confidentiality, but does not guarantee availability. These guarantees can only be assured if following a secure driver approach (Section 6.3.1). They can also be statistically guaranteed by means of redundancy, distribution, or recoverability mechanism based on periodic backup as in [285]. These techniques are out of scope in these thesis; we focus on confidentiality and integrity of sensitive assets. Still, they would be a valuable contribution to our work.

10.1.4.2 Attacks against Reference Monitor

An attacker with knowledge on the locking mechanisms that force untrusted applications to go through the secure area could target them to leak sensitive data or misuse other sensitive assets. There are two primary attacks that can be designed against the reference monitor: attacks against the locking mechanisms (i.e., memory locking, data locking, and peripheral locking), and attacks against the usage policy engine in TUM. We analyze each locking mechanism separately.

Memory Locking In this case, an attacker would try to directly compromise *Trusted Memory Enclaves*. This is probably the attack vector that makes most sense for an attacker to invest time and money in. As a reminder, trusted memory enclaves support that untrusted applications access unencrypted sensitive data. Thus, in case of a successful attack, an attacker could dump the contents of a trusted memory enclave and steal all sensitive data in it. If combined with an attack to hijack the patch of different legit applications, an attacker could systematically steal sensitive data protected by the secure area. This would expose all sensitive data protected by our trusted storage solution.

If we assume an implementation based on virtualization extensions besides TrustZone, where the MMU is virtualized and memory accesses are mediated by a thin hypervisor that redirects them to a TEE Trusted Cell component in the secure world (Section 8.2.2.1), an attacker could design two attack vectors: (i) attacks againsts the hypervisor, and (ii) attacks against the address space managed by a legit untrusted application.

With regards to the hypervisor, the attacker could try to either (i) bypass the call emerging from the hypervisor to the TEE Trusted Cell, (ii) modify the usage decision returning from the TEE Trusted Cell, or (iii) disable the components providing a virtualized MMU to break memory isolation. Any of these attacks requires tampering with the hypervisor. Given that we propose a thin hypervisor (without support for multi-virtualization), whose only purpose is to virtualize the MMU for the untrusted area, these attacks are very unlikely. Note that our proposal depends on ARM processors providing trap-and-emulate, so entry points to

the hypervisor are provided by the hardware, not by hypercalls. Even if the hypervisor was compromised, TUM offers an extra protection layer, minimizing the amount of leaked sensitive data by means of usage policies.

With regards to the untrusted application's address space, an attacker could attempt side-channel [178, 168] attacks to leak encryption information. In this case, TSM relies on OpenSSL, which is designed to resist these kinds of attacks. Randomization comes from the trusted area, not from the untrusted kernel, so attacks against in-kernel randomization sources such as */dev/random* are canceled. Also, an attacker could attempt Iago attacks [60] to modify the return value of system calls and corrupt the trusted memory enclave. Memory locking does not depend on the commodity OS, nor on its system call, therefore in principle Iago attacks should not be possible. Also, CPI guarantees against return-to-libc and ROP attacks should apply to Iago attacks. Still, since these attacks can tamper with concurrency and process identification kernel mechanisms (in which the page cache relies), we cannot completely remove the possibility for these attacks against trusted memory enclaves. Note that ARM does not count on *paging* to assign permission to specific pages and page cache as in x86. In this case, we rely on CPI to detect the attack based on memory accesses mediated by the hypervisor. While this is in principle enough to prevent Iago attacks, the fact that we do not count on CPU instructions to protect pages as in HyperSafe [282], opens for intricate attacks similar to Iago attacks. In the case that these attacks succeed, they would only leak specific sensitive data portion present in a trusted memory enclave; in no case can these classes of attacks compromise the TEE Trusted Cell to launch a systematic release of secrets in the SE. Investigating mechanisms against Iago attacks such as the ones that Criswell et al. propose in Virtual Ghost [79] are a topic for future research.

Data Locking As described in Section 8.2.2.2, data locking is entirely based on our trusted storage solution. We refer the reader to the security analysis that we provide below on trusted storage (Sections 10.1.4.3 and 10.1.4.4).

Peripheral Locking Since we build peripheral locking directly on top of TrustZone capabilities, we completely rely on TrustZone protection mechanisms for it. We refer the reader to the analysis provided above for TrustZone as a technology (Section 10.1.2.1). In summary, if an attacker can force the NS bit, peripheral locking would be bypassed. Also, if an attacker can manipulate interrupts, she could prevent a peripheral to be locked again after a legit untrusted component makes use of it (Section 8.2.2.3). In general, any attack vectors targeting secure peripherals rely on the AXI-to-APB bridge, or on secure memory isolation (also governed by the NS bit) if DMA is used. In general, lab attacks and complex shack attacks can compromise peripheral locking.

Usage Policy Engine Assuming that an attacker knows implementation details about the usage policy engine in TUM, she can design attack vectors to avoid usage restrictions. Under this scenario, an attacker could hijack the data path of a legit untrusted application and send a concrete set of inputs to steal sensitive data or misuse other sensitive assets. As we have discussed before, a complete usage control model should detect these attacks

through memory analysis and access pattern analysis. Still, if we assume the worst case, where an attacker can trick the usage policy engine, the sensitive assets it would have access to are limited. This is, an attacker cannot have access to all sensitive asset by means of this mechanism; it would be necessary to launch an independent attack per sensitive asset. Here, we believe that it is safe to assume that a usage policy engine containing such a number of vulnerabilities is very unlikely to reach the secure area undetected.

Also, an attacker could benefit from side channel attacks to (i) detect access patterns to sensitive information by exploiting a legit untrusted application, and (ii) misuse such application to exploit the concrete usage policy engine in TUM. In this way, an attacker could reconstruct some meaningful pieces of sensitive through several independent attacks. We rely on OpenSSL to prevent this attacks, and CPI to detect memory operations necessary to carry out a software-base side-channel attack. Hardware side-channel attacks (shack attacks and lab attacks) are out of scope.

In general, we provide the framework to support trusted services in the commodity OS, but we cannot respond for the quality of concrete components in the Trusted Cell. One of our design principles is that we follow a modular approach so that components can be changed and evolve with time. If a poorly implemented component replaces a part of the proposed Trusted Cell (e.g., usage policy engine), we are not in a position to anticipate a rigorous security analysis. Here, trusted memory enclaves featuring CPI (Section 8.2.2.1) are a very promising solution. In a fully virtualized environment it is not unrealistic to envision a trusted memory enclave being used for communication purposes between untrusted and trusted areas. In this way, attacks similar to Iago attacks would be canceled by CPI. As we have mentioned, fully implementing CPI as a guard system for trusted memory enclaves to provide memory protection in ARM-powered devices is a necessary step for future work.

10.1.4.3 Attacks against Trusted Storage (Trusted Generation)

In this section we cover attacks against the trusted storage solution where sensitive data is generated in the trusted area (Section 8.2.1.1). Here, an attacker with knowledge about how this trusted storage solution uses shared memory to reuse the untrusted I/O stack and store encrypted containers of sensitive data in untrusted secondary storage (Section 8.2.1.1) can compromise the storage process. Indeed, since we rely on the untrusted I/O stack to access secondary storage, an attacker could easily design a DoS attack where either the whole I/O stack is blocked, or targeted pages are prevented from being stored. An intricate attack in this class could specifically target secure containers and make it appear that storage primitives work properly by only blocking those I/O requests involving secure containers. Moreover, an attacker could fake I/O completions to make the TEE Trusted Cell believe that I/O operations are executing properly. Under this circumstances, and attacker could simply prevent that secure containers are stored or modify these containers with malicious data. We analyze different types of attacks and see how these attacks only can compromise availability and durability but never confidentiality and integrity of sensitive data.

Sensitive data leakage. The creation of trusted memory enclaves guarantees that secure containers never leave the trusted area unencrypted; the shared memory region enabling the enclave is only created when the secure container is encrypted and the encryption key is safely stored in the SE. In general, while an attacker could use the mentioned mechanisms to compromise the availability and durability of secure containers, these attacks cannot affect the confidentiality or integrity of the data stored in them. Assuming the resilience of our encryption scheme (Section 8.2.1.1), data cannot be leaked by attacking REE Trusted Cell components.

An alternative to reusing the untrusted area I/O stack would be the use of secure drivers 6.3.1. However, as we have already discussed, this would entail a dramatic increase of the TCB, which would result on a larger attack surface against the TEE Trusted Cell. Such solution would provide higher levels of availability and durability at the cost of confidentiality and integrity of sensitive assets. We state it again: For us split-enforcement is about providing higher levels of confidentiality and integrity. Thus, any compromise that results on confidentiality and integrity being lowered is not an option.

Secure data path manipulation. Secure containers are already encrypted when they reach the untrusted area. Moreover, the process of placing a secure container in a trusted memory enclave does not rely on untrusted operations; the secure area allocates shared memory region and places an encrypted secure container in it before notifying the REE Trusted Cell. This makes impossible any form for replay attack such as ROP, return-to-libc, Iago attacks, or side channel. These attacks are funded on the assumption that the trusted area relies on primitives provided by the untrusted area (e.g., system call return values). This is not the case for secure containers. The only possible attacks are against the I/O operations but since these do not interact with the trusted area to establish the secure data path, all attack vectors targeting sensitive data it are invalid.

Data destruction Access to secondary storage is allowed to the untrusted commodity OS. In this case, where sensitive data is strictly generated in the secure area, mediating I/O requests in the untrusted area is not necessary. This makes our trusted storage solution immediately portable to devices that do not count on a programmable I/O device supporting *Software Defined Storage (SDS)*. However, this comes at the cost of durability and availability. In this way, the untrusted area would have direct access to the files containing the secure containers and therefore could destroy them or tamper with them in any way. As mentioned above, extending this solution with redundancy, distribution, or recoverability mechanism would be a valuable contribution to this solution. In next section, we argue how it is possible to provide a solution with better durability properties by means of the mentioned new generation of SDS I/O devices.

10.1.4.4 Attacks against Trusted Storage (Untrusted Generation)

In this section we cover attacks against the trusted storage solution where sensitive data is generated in the untrusted area (Section 8.2.1.2). Here, data is *compromised on creation*.

Thus, an attacker can (i) steal or tamper with sensitive data as it is being produced, or (ii) insert pieces of malicious code in sensitive data pages intending to exploit code bugs in the secure area. Since the untrusted area is assumed to be fully compromised, the only way to protect against these attacks is through trusted memory enclaves that encrypt pages before they are sent down the I/O stack. This technique, as mentioned in Section 8.2.2.1, is similar to the memory encryption scheme presented in Overshadow [62]. Since we have already discussed the attack vectors against memory locking when we presented the reference monitor, we now assume that memory protection is in place. If memory protection is bypassed or not implemented[6], this solution cannot guarantee confidentiality and integrity of sensitive data.

Sensitive data leakage and manipulation. If data is generated in a trusted memory enclave, all memory operations are mediated by the secure area. Thus, data is *protected on creation*, with regards to the kernel. This enables legit untrusted applications to produce sensitive data and guarantee their confidentiality and integrity while pages are managed by the untrusted kernel. Since pages are encrypted when sent down the I/O stack, confidentiality and integrity are maintained until the pages reach the I/O device. Note that the I/O device can verify the integrity of these pages by communicating with the secure area. Once a page reaches the I/O device durability is guaranteed as long as the usage policy engine in TUM can correctly discern between legit and illegitimate I/O requests.

If the untrusted application is malicious, trusted memory enclaves cannot protect against sensitive data being leaked. Note that this use case is out of scope; we conceived this mechanism as a way to increase the security of legacy applications generating sensitive data; if the application is malicious then it can directly leak sensitive data as it produces it. However, TUM would still be able to enforce usage device policies and application contracts. Assuming that these are well define sensitive data leakage would be prevented. Attacks against this mechanism are attacks against the reference monitor (Section 10.1.4.2).

Secure data path manipulation. Since secure data path in this case rely on trusted memory enclaves, we refer the reader to the security analysis on memory protection Section 10.1.4.2.

Data destruction Access to secondary storage is mediated by the storage device, which is trusted and implements the subset of TUM that deals with I/O usage policies. In this way, the untrusted area has no direct access to data pages; TUM protects pages containing sensitive data. The policies installed in TUM are the ones regulating page access. In this case, durability of data can be guaranteed from the moment the data reaches the storage device. This is, an attacker cannot destroy data that is already stored. However, since data pages need to be transmitted to the storage device by the untrusted I/O stack, an attacker could target these specific pages and impede that they reach the storage device. Since these pages are encrypted as guaranteed by the trusted memory enclave, an attacker

[6]Note that trusted storage and the reference monitor are orthogonal. Given their modularized design, one can be implemented in a Trusted Cell instance without the other.

can prevent them from reaching the storage device (i.e., a form for data destruction), but cannot compromise their confidentiality or integrity.

10.1.4.5 Counter-Protection Techniques

We finish the security analysis for the Trusted Cell by providing a set of counter-protection techniques we have explored using Trusted Cell components. Since the Trusted Cell can guarantee the integrity and confidentiality of sensitive data, it can use this root of trust to detect attacks against trusted services. We present the three counter-protection mechanisms we have explored:

Secure Containers. As presented in Section 10.1.4.3, the Trusted Cell re-uses the untrusted I/O stack to store secure containers in secondary storage. Using this same procedure, the Trusted Cell can generate a false secure container that it uses as *bait*. The Trusted Cell stores it and reads it periodically; if the secure container can be acquired and preserves its integrity, the trust that the Trusted Cell has in the untrusted area augments. Since the attacker has not means to know which secure containers are used for these purpose, the Trusted Cell can statistically verify that the untrusted has not been compromised. A compromised secure container is detected by the Trusted Cell with a 100% guarantee.

I/O Completion Acknowledgment. If a SDS I/O device is used as secondary storage, the same management component that allows for key exchange with the secure area can be used to acknowledge I/O completion. In order to lower the number of context switches to the secure area, thus minimizing performance overhead, a set of I/O completions can be send to the secure area. In this way, TUM can guarantee that all secure containers have reached secondary storage. As described in Section 10.1.4.4, this allows to provide durability of sensitive data. If a set of I/O completions are consistently not acknowledged, the untrusted area is likely to have been compromised.

CPI. CPI is in itself a counter-protection mechanisms since it does not prevent an attack; it detects when an attack has taken place. In this way, the Trusted Cell can take measures to protect the rest of the sensitive data it manages based on CPI outputs.

When the Trusted Cell detects that the untrusted area has been compromised, we have explored two protection mechanisms: (i) alerting the user, and (ii) locking the secure area and stopping communication with the untrusted area. In the second case, secure memory is *zeroized* and all secrets stores in the SE. An attacker would then have to target the SE to store secrets; all attack vectors using hack and shack attacks targeting TrustZone are invalidated in this case. In this case, we give up on availability to provide confidentiality and integrity of sensitive data.

10.1.5 Attacks against Certainty Boot

The two-phase verification mechanism that supports certainty boot relies on the same primitives that enable split-enforcement. More specifically, it relies on (i) our trusted solution to store the hash of the kernel to be booted; and (ii) run-time security primitives to communicate with the secure area and verify the kernel image at boot-time, and the boot traces at run-time. We analyze attacks against the two-phase verification. Then we analyze attacks against the *First Stage Boot Loader (FSBL)* and *Second Stage Boot Loader (SSBL)*, since they represent our root of trust for certainty boot.

10.1.5.1 Attacks against Two-phase verification

Attacks against the hashes in the SE are directly attacks against our trusted storage solution. Hence, we refer the reader to Section 10.1.4.3, where we present the security analysis for sensitive data generated in the trusted area.

Attacks against the run-time security primitives that allow to verify boot traces are wider, since these run-time security primitives enable direct communication with the TEE Trusted Cell. Without the reference monitor in place, such run-time security primitives would be exposed to side-channel and Iago attacks. All the responsibility relies then on TUM to enforce usage policies. Still, superseding the boot trace information provided by the secure area is an attack out of control of TUM, which would succeed.

If the reference monitor is in place, trusted memory enclaves would be used to protect the communication between untrusted applications and the secure area. In this case, the communication relies on the resilience of the trusted memory enclave. We refer the reader to Section 10.1.4.2, where we provide the security analysis for all locking mechanisms in the reference monitor.

DoS are always a possibility since an attacker could prevent calls to the secure area. In this case, the boot sequence would be aborted. In case of untrusted applications, these would be terminated. This is indeed the locking mechanism protecting two-phase verification.

10.1.5.2 Attack against FSBL and SSBL

Since the root of trust begins with the FSBL and SSBL, a sophisticated software attack that supplants the SSBL could prevent the boot of a legitimate TEE. Thus, preventing the verification and logging of booted components depicted in Figure 9.2. While the attacker would gain control of the device and the communication to the SE, the secrets stored in the SE would remain inaccessible at first. Indeed, the SE applet is configured to wait for a trusted system state, thus it will not reveal sensitive information if the correspondent boot hashes are sent. The attacker would need to modify the SSBL so that it is sending the hashes of a normal trusted boot. As the SE is only a passive component, it does not have methods to verify the trustworthiness of the source of a received message. Signing the messages would also not prevent these attacks due to the inability to securely store the private key on the

mobile device. While intricate, the attack is theoretically possible. However, we assume that the SSBL is locked by the OEM and additionally verified by the FSBL, as it is the case in current devices. Still, the lack of source verification capability of the SE applet remains an open challenge for the research community.

If an attacker only substitutes the OS bootloader, an untrusted OS would be booted without access to the SE. This is already one of the scenarios contemplated as normal operation (i.e. step 6 in Figure 9.2).

10.2 Design Requirements Compliance

When we presented split-enforcement in Chapter 6, we established the hypothesis that an untrusted application could be represented by a state machine, where states represented the sensitive data being used, and transition represented actions being taken. Also, we provided a reference monitor design to mediate untrusted applications and system resources through a policy decision point. Then, in Chapter 7, we provide a number of design principles for TEE Linux support in terms of flexibility, genericity, and functionality. Finally, in Chapter 8 we put everything together to design and build a Trusted Cell.

In this section we analytically evaluate how the requirements we established in Chapters 6 and 7 are materialized in the Trusted Cell (Chapter 8).

10.2.1 State Machine and Reference Monitor Abstraction

In our hypothesis (Section 6.2), we argued that a state machine abstraction together with a reference monitor would be sufficient to enforce usage policies. As we have shown with the the Trusted Cell, our hypothesis is confirmed. The state machine representing an untrusted application is located in TUM (Section 8.2.2). By means of a policy engine, also in TUM, a decision can be generated based on the state of an untrusted application, and the actions it takes. This is represented by states and transitions in the state machine, respectively.

By using encryption and integrity measurements in TSM and TIM, we guarantee the resilience of the states and the veracity of the transitions. Indeed, TSM, TIM, and TUM allow to implement the trusted storage solution and the locking mechanisms in the reference monitor that guarantee such veracity. Moreover, since these same locking mechanisms prevent the use of sensitive assets without an untrusted application passing the control flow to the secure area, the enforcement of the usage policies governing transitions is also guaranteed.

All in all, such state machine abstraction allow to keep track of the use that an untrusted application makes of sensitive assets. Together with a reference monitor and trusted storage, transitions in the state machine can be monitored and usage decisions governing them enforced. Split-enforcement is then supported.

10.2.2 Low TCB

Our generic TrustZone driver is in total 4030 LOC in its current status (Table 7.1). The Trusted Cell prototype is less than 3000 LOC (Table 8.1). Even when following programming conventions as in *checkpatch*, and giving a clean *split* analysis, we are in the range of the 10K LOC that can be formally verified [176]. If extending our Trusted Cell design with a thin hypervisor to virtualize the MMU, we would be around the 10K LOC - SecVisor's hypervisor is ∼3000 LOC [255]. While Open Virtualization is ∼30K LOC most of its code is redundant due to lack of internal abstractions: dealing with the stack to pass parameters, or supporting Global Platform specific functions which could be refactored (Section 7.1.1). The core components that allow to implement the secure monitor are below 4000 LOC. Finally, OpenSSL, which is ∼400K LOC is a mismatch for Open Virtualization. Our analysis of OpenSSL shows that porting the encryption algorithms that we use for TSM and TIM we would be in the range of 3000 LOC. This suits other work porting encryption libraries such as Overshadow [62] or VPFS [285].

10.2.3 Protection vs. Innovative Applications

From a protection perspective, in the security analysis we have seen that the only type of attacks that can leak portions of sensitive data are Iago [60] attacks. These attacks can compromise trusted memory enclaves since data is allowed unencrypted into the untrusted area. Sensitive area *inside* the secure area is not affected by these attacks. In fact, only lab attacks (Section 10.1.1) targeting the SE could expose secrets in the secure area.

While we limit the TCB to maintain a small attack surface and offer such strong security guarantees, untrusted applications can be as complex (and compromised) as developers wish, just as the commodity OSs underneath them. This allows for innovative services in the untrusted area. By means of run-time security primitives, untrusted applications can make use of sensitive data to provide these services. The presented trusted storage and reference monitor solutions support these run-time security primitives in the secure area. In this way, a balance between innovation and protection is provided with the Trusted Cell. A further analysis on how to better protect trusted memory enclaves against Iago attacks would increase protection while not limiting untrusted applications. As mentioned, studying proposals such as Virtual Ghost is a topic for future research.

10.2.4 Untrusted Commodity OS

As we have discussed throughout the whole thesis, the untrusted area is fully untrusted. This includes our generic TrustZone driver and the REE Trusted Cell. All protection mechanisms are provided by the trusted modules in the secure area, which are out of the scope of the untrusted area. The assumption that the commodity OS and untrusted applications are fully untrusted is then maintained.

10.2.5 TrustZone Driver Genericity

One of the main requirements we imposed for the *generic* TrustZone driver was not to impose policy for neither kernel submodules not user applications. The simple *open/close*, *read/write* interface we proposed for the *tz_device* has allowed us to implement Open Virtualization's OTZ API and a subset of Global Platform's API with almost no modification[7] on top of it.

Moreover, it has allowed us to implement the communication between the REE Trusted Cell and the TEE Trusted Cell, which require much more integration than Global Platform's API. Given the genericity of *open/close*, *read/write*, which follow the Linux design pattern *everything is a file*, we cannot see a case where it cannot be re-used to implement more complex interfaces.

From the other side of the interface, i.e., new specific TEE drivers, we have followed Linux kernel design conventions to (i) define an interface in a structure, and (ii) implement it in a specific driver. As a proof, we have ported Open Virtualization to be fully compatible with the defined interface. Other kernel device drivers such as I2C or TPM, the VFS layer, or the LSM framework use this design patter to define an interface that several components implement concurrently. This allows for genericity while maintaining one unique interface.

10.3 Conclusion

In this analytical evaluation we show how our contributions, i.e., generic TrustZone Driver, Trusted Cell, and Certainty Boot (i) resist a large percentage of the attacks vectors that we know of today, and (ii) comply with the requirements we had established for them in our design. Indeed, we have satisfied our main objective: increasing the security of complex applications without killing innovation. The untrusted area, where innovative applications execute, can be fully compromised. However, by means of a series of run-time security primitives, these applications can access trusted services that serve them while guaranteeing the confidentiality and integrity of sensitive data. More importantly, these trusted services not only enable the outsourcing of secure tasks to a trusted area protected by hardware, they also allow sensitive data to leave such trusted area and be directly accessible to innovative, untrusted services, while still guaranteeing its confidentiality and integrity.

In conclusion, we have analytically argued how our contributions successfully implement split-enforcement to allow innovative services to use sensitive information without compromising it.

[7] The only modifications required where substituting file descriptor instructions triggering *ioctl* calls in the kernel to our interface in the tz_device.

Chapter 11

Experimental Evaluation

As we discussed in Chapter 8, when discussing the implementation of the Trusted Cell, we encountered two fundamental limitations in our evaluation platform: The lack of PCI-express support for Zynq ZC702, and more importantly, the lack of trap-and-emulate virtualization extensions in TrustZone and in the two Cortex-A9 processors powering the Zynq ZC702. This has prevented us from fully implementing memory locking and support for *Software Defined Storage (SDS)* I/O devices (Chapter 6). Still, we decided to explore the design space entirely and take our evaluation platform to its limits. Note that these two limitations emerged in the process of implementing the Trusted Cell. By completely exploring this design path our intention is to (i) expose further limitations in currently available hardware and software supporting a TEE, and (ii) gain a better understanding of the hardware characteristics that a platform should have to fully support split-enforcement.

Our approach is to complete a proof of concept implementation for the trusted service fragments that are not supported by our platform. While this implementation does not comply with split-enforcement, it allows us to experiment with trusted services and explore non-functional requirements such as performance. Note, that this proof of concept implementation lowers resilience to attacks stemming from the untrusted area, but trusted modules in secure area are directly portable to a platform supporting split-enforcement. Therefore, our experimental results can be directly applied to other platforms using TrustZone to leverage a TEE. Moreover, we design the proof of concept implementation in such a way that calls to the secure area are identical to the ones that would have come from a split-enforcement compliant implementation (Section 8.3). We believe that exhausting this path is a contribution in itself since it exposes concrete limitations of ARM TrustZone security extensions in terms of virtualization, which given their secrecy, have not been fully addressed before in the security community. Our experience is too that these limitations are not fully understood in the industry either, given the predominance of Global Platform use cases.

In this Chapter, we provide an experimental evaluation of our Trusted Cell prototype implementation, focusing on the performance overhead of executing secure tasks in the context of split-enforcement locking mechanisms: memory operations (memory locking), I/O operations (data clocking), and system bus accesses (peripheral locking) from a micro and macro perspective. We also look at the performance impact of CPU usage in the secure world.

In this way, we provide a complete benchmark for the ARM TrustZone security extensions. Given the obscurity that still today surrounds TrustZone and the increasing popularity of ARM processors, understanding what is the impact of using TrustZone is important for system designers. For us, this experimental evaluation is paramount in order to (i) understand the viability of mediating system resources through the secure area as we propose with split-enforcement in terms of overhead, and (ii) choose a platform that can support the next iteration of our work.

11.1 Experimental Setup

As we have mentioned, when we started experimenting with TrustZone, options where limited both in terms of hardware and software. Zynq was one of the few platform fully supporting TrustZone, and the only one where TrustZone registers where available. Open Virtualization was to the best of our knowledge the only non-commercially TEE framework available to the public at that time. In fact, options stayed for a long time very limited. As an example, Johannes Winter explains in [291] how the lack of access to TrustZone configuration registers on Samsungs S3C6410 SoC - a platform also enabling TrustZone - delayed their work significantly. Today, academic experimentation with TrustZone, specially since we first presented our platform in [131], takes place almost entirely on Zynq. Examples include [300, 188, 300, 228]. In this section, we provide a full description of our experimentation platform.

We rely on a Xilinx's Zynq-7000 SoC, since it is one of the few development boards that provides full support for ARM's TrustZone security extensions. Zynq is a System-on-Chip (SoC) composed by a Processing System (PS) formed around a dual-core ARM Cortex-A9 processor, and a Programmable Logic (PL), which is equivalent to that of an FPGA. More concretely, we use the ZC702 development board which features the Zynq-7000 SoC described above clocked typically at 666MHz, and comes with 1GB of DDR3 RAM typically clocked at 533 MHz. The ZC702 is equipped with a number of interfaces that facilitate programming: One Gigabit Ethernet interface, one USB-to-UART Bridge, JTAG, HDMI, and a SDIO Card Interface[1]. Moreover, it also provides I2C (x2), SPI (x2), USB (x2) bus interfaces, and a GPIO. Different headers allow to interface these: a 2x6 Male Pin I/O, 2x6 and 1x6 PMOD I/O, and two LPC connectors. The ZC702 is depicted in Figure 11.1. Detailed information about interfaces and headers can be found in the Zynq Book [80], and in the Xilinx Zynq-7000 SoC ZC702 Evaluation Kit product page[2]

Unlike other commercial SoCs, TrustZone support in the dual-core ARM Cortex-A9 processor is not disabled. Moreover, documentation introducing the TrustZone configuration registers for Zynq are public [298]. Also, Xilinx maintains a set of public git repositories[3] containing patched versions of the Linux Kernel, u-boot, qemu, etc., as well as a Zynq base targeted

[1] Note that these are already incorporated interfaces. The Zynq-7000 features 2 pairs for each I/O interface (CAN, UART, SD, and GigE)

[2] `http://www.xilinx.com/ZC702`

[3] `https://github.com/xilinx`

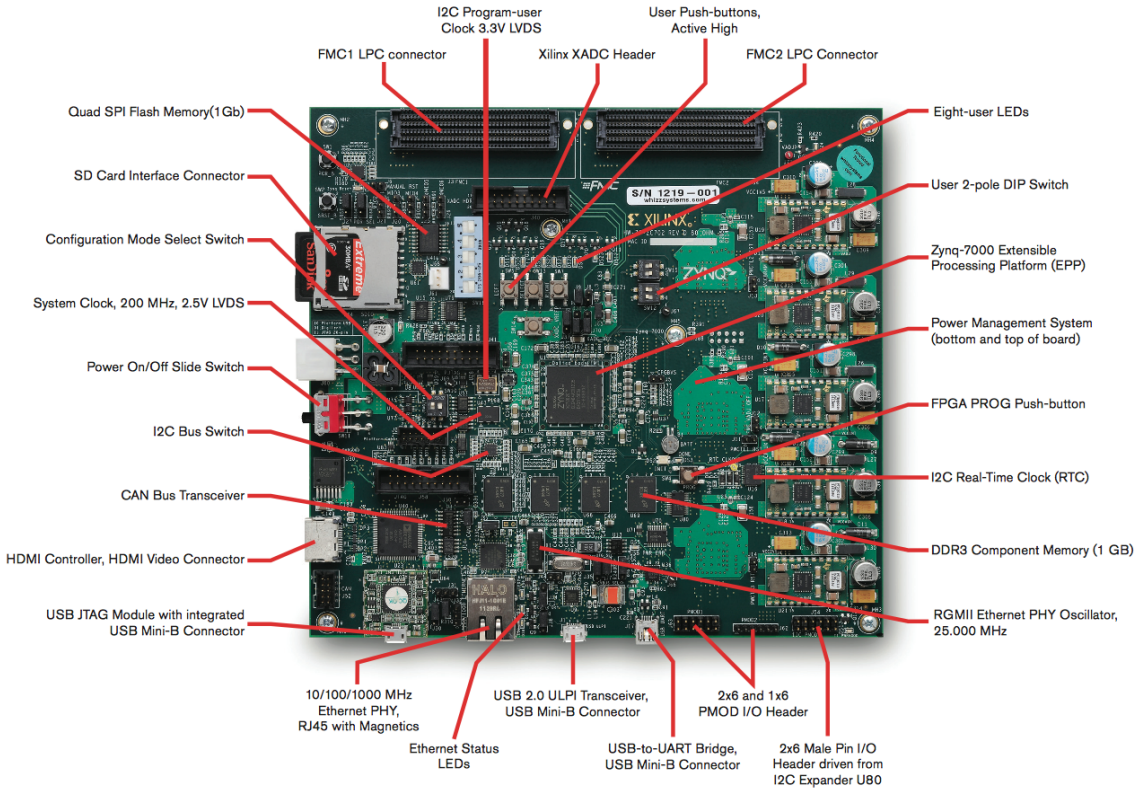


Figure 11.1: Zynq-7000 All Programmable SoC: ZC702 Evaluation Board. Image credit to Xilinx (`http://www.xilinx.com/publications/prod_mktg/zynq-7000-kit-product-brief.pdf`)

reference design (TRD)[4], which eases the implementation and engineering effort. A number of wikis[5] and forums[6] complement support with a community around Zynq. We have been a part of this community to support newcomers with Zynq and TrustZone documentation.

From a software point of view, any design of a trusted service using TrustZone should rely on four main components:

1. A TrustZone operating system, which represents a specific way to organize TrustZone's secure world.
2. A TrustZone driver that enables interactions between secure and non-secure worlds;
3. A commodity OS that supports the execution of complex untrusted applications (i.e., innovative services);
4. A set of trusted modules that implement the trusted services in the TrustZone secure world.

[4] `http://www.wiki.xilinx.com/Zynq+Base+TRD+14.5`
[5] `http://www.wiki.xilinx.com/Zynq+AP+SoC`
[6] `http://forums.xilinx.com/t5/Zynq-All-Programmable-SoC/bd-p/zaps`

In our experiments, the TrustZone operating system is Sierraware's GPL version of Open Virtualization[7]. We use Xilinx's patched Linux Kernel (version 3.8.0)[8] as the operating system running in the non-secure world, together with a light command-based version of Ubuntu[9]. These systems respectively manage the the secure space, kernel space, and user space.

In order to organize the communication between kernel and secure spaces, we incorporate the generic TrustZone driver and the REE Trusted Cell to Xilinx's Linux 3.8.0 kernel. Software for both the secure and non-secure worlds is cross-compiled using CodeSourcery toolchain 2010q1[10] in order to avoid misleading results due to different compiler optimizations[11]. Finally, we add user space support in Ubuntu to allow communication with the TrustZone drivers by means of the *tz_device* user space interface. Besides, we implement user space support as a set of commands that are used for configuration, testing, and debugging (Section 7.2). Most of our tests are unittests (CuTest) that execute in user space and use the tz_device to communicate with the TrustZone driver to trigger calls to the secure area (Section 7.1.1.4). Details about the generic TrustZone driver status are given in Section 7.3; details about the Trusted Cell prototype are given in Section 8.3.

It is relevant to mention that we configured Open Virtualization to execute in asymmetric multiprocessing (AMP) mode. This means that one of the two Cortex-9 cores is dedicated to kernel space, and the other to secure space. This means that Zynq is seen as an uniprocessor machine from kernel space. The main reason for this configuration is that symmetric multiprocessing (SPM) support is experimental in Open Virtualization. This could add a misleading performance overhead due to bad synchronization between the two cores, thus we have avoided it. While AMP is typically considered to perform better, with only two cores the benefit is expected to be minimal [96]. Note that the cost of context switching to the secure area is the same for AMP and SMP configurations. Hence, for our experiments the core is not sacrificed, but used differently. We have also experimented with a SMP configuration, and can support this claim experimentally. However, this configuration is not stable, and the reason why we are more confident providing experimental results under AMP. We count on fully implementing support for SMP in the future. This is indeed a necessary step to reach the level of a commercial prototype.

11.2 Performance Overhead

We conduct our performance experimental evaluation through an *application benchmark* and a set of *microbenchmarks*. This allows us to look at the performance impact from two different perspectives. The goal of the application benchmark is to present the system with different heavy untrusted applications representing innovative services and evaluate the impact of

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

[8] `https://github.com/Xilinx/linux-xlnx`

[9] `http://javigon.com/2014/09/02/running-ubuntu-on-zynq-7000/`

[10] `https://sourcery.mentor.com/GNUToolchain/release1293`

[11] At the time of this writing, we have ported to Linaro's cross compiler since it is a maintained, open source tool. CodeSourcery stopped having open source support when it was acquired by Mentor Graphics. Still, the results we show here are made with CodeSourcery's cross compiler.

triggering a context switch per system call, where access to system resources is mediated by the policy engine (TUM) in the secure area. Understanding the overhead created by generating (and enforcing) these usage decisions is important in order to understand the impact of mediating access to system resources through the Trusted Cell. We also conduct a set of microbenchmarks where we compare the performance of kernel and secure spaces when executing workloads that are equivalent in number of instructions, complexity, and outcome (e.g., encryption).

Every time that a secure task (or trusted module) is called from kernel space, a context switch takes place between kernel and secure space. Even when this process is implementation specific, it at least involves: saving the untrusted state, switching software stack, loading the secure state, dispatching the secure task, and returning to kernel space (save secure state, change software stack, load untrusted sate). We denote this double context switch *Secure Round Trip*. From a system design perspective this is the real overhead introduced by the Trusted Cell into the untrusted software stack. Understanding the overhead of calling a secure task in relation to the cost of calling a function locally in kernel space is necessary when designing the communication protocol between the trusted and untrusted areas. We will calculate this overhead both in the application benchmarks and the microbenchmarks and then use the results to present a discussion on the viability of split-enforcement and the Trusted Cell.

The metric we use is the overhead introduced by the secure space, defined as:

$$Overhead = \frac{(T(secure) - T(kernel))}{T(kernel)}.$$

11.2.1 Application Benchmarks

Modern systems are almost uniquely multi-process and multi-threaded. Established Linux applications such as *tar* or *make* issue from tens to even thousands of system calls per second [284] while running simultaneously and probably managing several threads. We pick such applications as representatives of untrusted applications, whose workloads represent innovative services.

In this set of experiments we take a wide range of applications that exhibit different workloads, and execute them in our experimental setup. The idea is to observe the impact of triggering a context switch to the secure area for each system call, when executing a heavy workload. Since each system call triggers a call to the TEE Trusted Cell in the secure area, we study the degradation of the approach as the number of instructions executed per call to the secure area augments. These instructions represent the cost of generating and enforcing a usage decision. Other work done in the area of reference monitors [108, 85] and system call evaluation [283, 145] shows that the overhead can be substantial.

Table 11.1 contains the different workloads. We assume that these Linux applications approximate upper bounds for the characteristics of innovative services. For example, compiling the Linux kernel in the ZC702 (*gcc through make*) requires 100% use of memory and CPU for

more than 50 minutes on average, without the overhead of the mediating system resources in the secure area.

Workload	Description
make	Build of the Linux 3.8.0 (Xilinx) Kernel.
bzip2	bzip2 on the Linux kernel source.
tar / untar	Tarball of the Linux kernel source.
dd	Create a 200MB file using the Unix dd utility.

Table 11.1: Array of applications to evaluate the performance overhead of using the Trusted Cell to mediate system resources.

Figure 11.2 shows the overhead of triggering a context switch to the secure area per system call, compared to a "normal" execution, where no context switch happens. We show the performance degradation as the number of instructions executed in the secure area per system call increases. What is interesting is that the overhead varies depending on the workload. For example, applications like *tar* or *make* show a significant overhead, reaching 16% in the latter. However, *dd* shows almost no overhead. This performance gap is due to the different number of system calls issued by each applications. Using *strace* we can see that *dd* issues a total of 106 calls, spending a great amount of time in each (e.g., 2.66 seconds per read call), while *tar* issues a large number of short system calls. Since the overhead is introduced per system call, the resulting overhead is obvious. In Table 11.2 we present the outcome of *strace* for the *tar* command.

Looking at the different application benchmarks we obtain the cost of a secure round trip from the application perspective. This is represented in Figure 11.2 by the overhead created by executing zero instructions (*0 ins.*) in the secure area (TUM). Note that this measurement is more valid from a systems perspective than just measuring the cost of the SMC call in terms of interrupts in the Cortex-A9 processor; the secure monitor, the logic loading the secure stack, and the secure dispatcher forwarding the call to the correct secure task need to be taken into account. Based on our experiments, the overall cost of completing a secure round trip oscillates in the order of the few microseconds (5.3μs on average) with peaks in the order of 20μs[12]. We obtained these access times by analyzing the *strace* outputs for each application benchmark[13]. We explain the variation in microseconds through the combination of caches and buffers present in the Cortex-A9 processor, Open Virtualization, and the Linux kernel.

Experimentally evaluating that a secure round trip is in the order of a few microseconds is very relevant in terms of evaluating the applicability of split-enforcement and allows us to argue for which types of operations its use is acceptable and for which it is not; memory operations are much cheaper than I/O operations, thus the performance impact of a few microseconds to perform a secure round trip is not the same for these two cases. We will take this discussion further in Section 11.2.3 once we have presented the microbenchmarks

[12]Note that we omit the *wait4* system call since it represents the time when a function suspends execution of its calling process until status information is available for a terminated child process, or signal is received. This time is irrelevant in terms of overhead.

[13]Note that we obtain *strace* outputs in a separate experiment from the benchmark not to affect it with time measurements

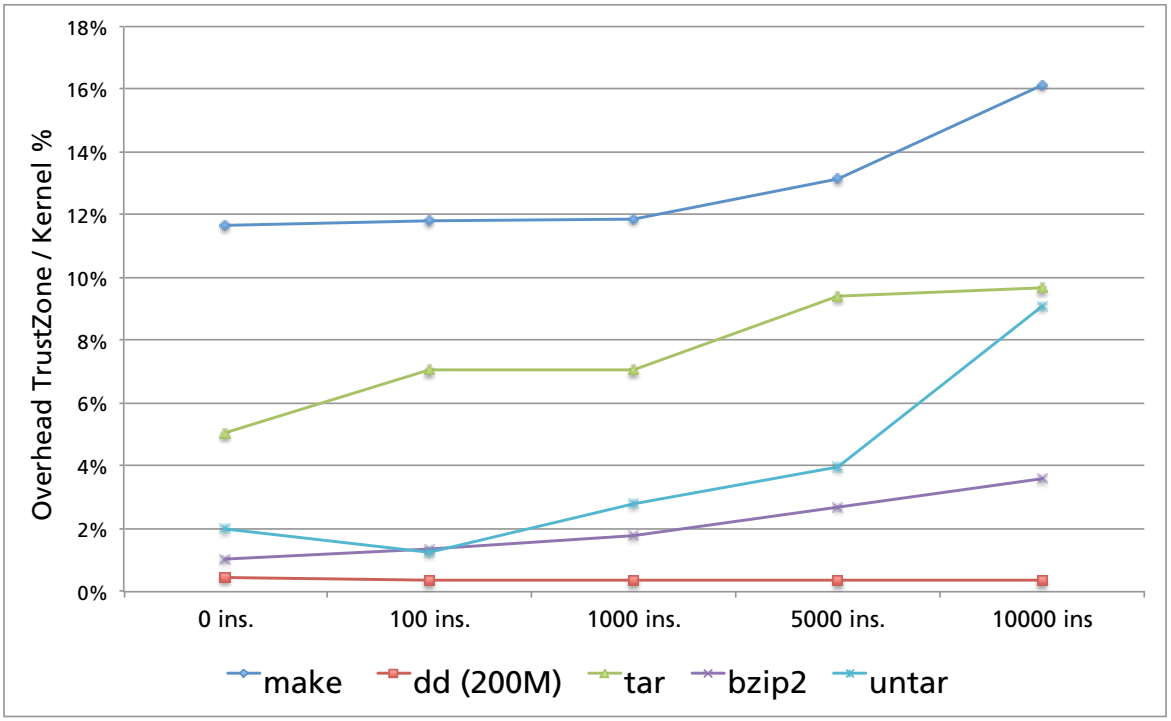


Figure 11.2: Overhead of mediating each system call through the secure area for a range of applications exhibiting different workloads. Y axes show the overhead. X axes show the number of instructions executed per system call in the secure area.

for different CPU, memory operations, and also secure round trip in next subsection.

As we mentioned when we presented the process of porting Open Virtualization to the generic TrustZone driver in Section 7.2.3, when executing these experiments we experienced that Linux kernel crashed in kernel space due to a lack of memory. Further investigation exposed two memory leaks of 4 and 32 KB respectively in Sierraware's TEE implementation. The 32KB leak could be found using *kmemleak*[14] since memory ended up being dereferenced. The 4KB one on the other hand was a consequence of storing a bad memory reference, which prevented an internal list from being freed. Since the amount of memory lost per call was that small, the leak has most probably not been significant in TEE typical use cases. We have reported the issue to the interested parties, as well as a patch fixing it. The memory leak is now fixed for the open source and commercial version of Sierraware's TEE framework.

11.2.2 Microbenchmarks

n this set of experiments we observe the difference in performance when a workload is executed in secure and kernel space respectively. This is specially relevant for the pervasive OS support operations (e.g., memory operations). The microbenchmarks presented here are designed to answer what is the cost of using secure space services instead of kernel ones.

[14] `http://lwn.net/Articles/187979/`

syscall	T. (%) M	sec. M	μs/c. M	T. (%) N	sec. N	μs/c. N	calls
read	31.54	2.632572	30	37.51	2.541182	29	87969
newfstatat	18.77	1.566603	34	21.93	1.485474	32	46323
write	18.74	1.563877	16	18.10	1.226000	13	96003
openat	15.04	1.255282	28	14.72	0.997364	22	45014
fstat64	8.66	0.722392	8	3.47	0.234838	3	93211
close	4.84	0.403546	9	2.99	0.202657	4	45037
getdents64	1.84	0.153971	24	0.98	0.234838	11	6333
fcntl64	0.43	0.035874	6	0.00	0.000000	0	6337
wait4	0.15	0.012196	12196	0.29	0.019667	19667	1
open	0.00	0.000000	0	0.00	0.000080	2	43
...	...	...	...	...	...	...	...
total	100.00	**8.346313**	-	100.00	**6.773886**	-	426467

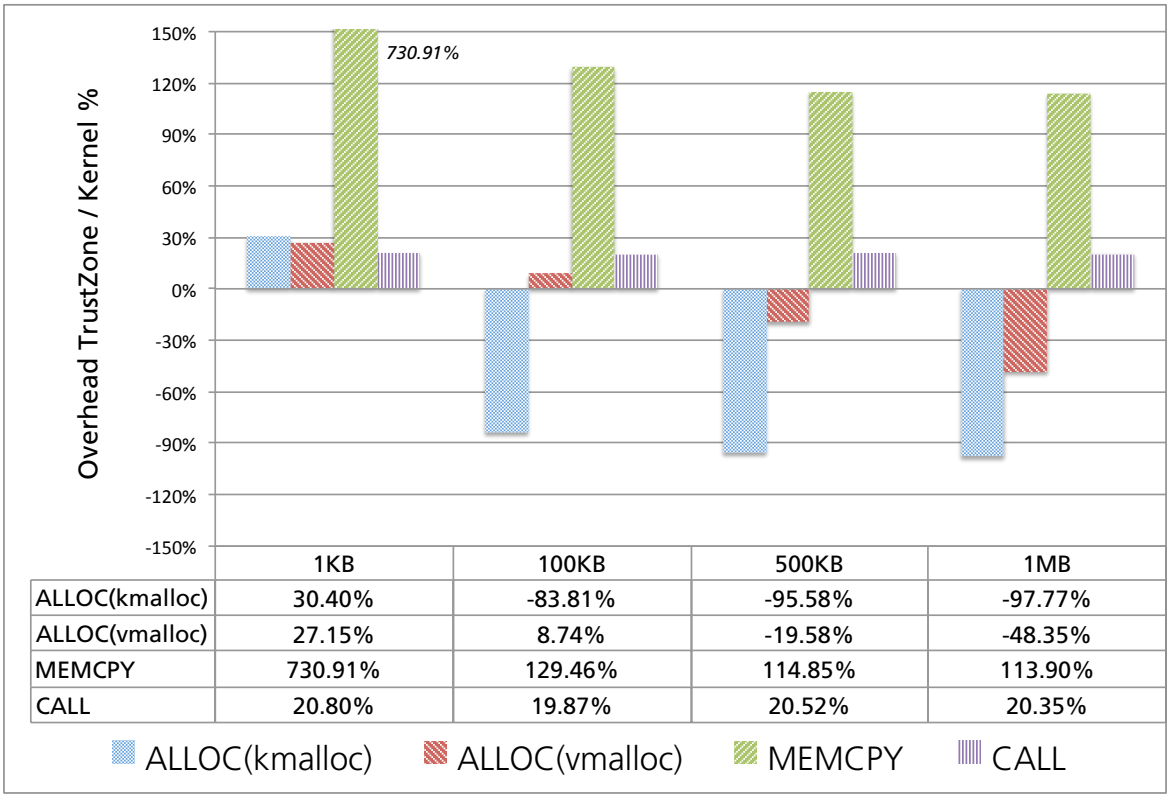
Table 11.2: *strace* output for *tar* application when syscall mediation through the secure area is enabled (M = mediated execution) and 10000 instructions are executed in the secure area per system call, and when syscall mediation through the secure area is disabled (N = normal execution). Columns respond to typical *strace* output. T = Time in percentage (%), sec. = seconds, μs/c = microseconds per call. First and last columns are common since they represent syscall name and number of calls respectively. System calls whose output for μs/c. is *0* are omitted (e.g., lseek, access, uname).

Here we not only calculate the overhead of a secure round trip, but we also include the cost of executing different operations in the secure area. This allows us to (i) validate the application benchmarks from a micro perspective, and (ii) compare how kernel and secure spaces relate to each other. Since the operations offloaded to the secure area would otherwise be carried out in kernel space, all microbenchmarks refer to the overhead of using secure space relative to kernel space.

Memory Operations Figure 11.3 shows the overhead of executing memory operations in secure space. The figure presents (a) the overhead of allocating memory in secure space (compared to kernel space) using *kmalloc* and *vmalloc*, (b) the overhead of copying memory with *memcpy*, and (c) the overhead of executing a *call* to secure space.

In Open Virtualization, when the secure area initializes at boot time, a memory pool is created using a technique similar to the buddy system [141]. Since the secure area has a limited amount of memory, it is relatively cheap to pre-allocate and manage this pool. As a result, secure tasks can obtain dynamic memory at constant cost. We verify that, in our experimental setup, this cost is indeed fairly constant ($\sim$200μs).

The design principle followed in Open Virtualization is that secure tasks will not need to allocate big amounts of memory, and allocations will not occur as often as in a general purpose OS. In kernel space however, Linux uses slab allocation [54], which is designed to allocate large amounts of memory while minimizing external fragmentation. This explains that secure area is faster at allocating memory as the requested size increases.



	1KB	100KB	500KB	1MB
ALLOC(kmalloc)	30.40%	-83.81%	-95.58%	-97.77%
ALLOC(vmalloc)	27.15%	8.74%	-19.58%	-48.35%
MEMCPY	730.91%	129.46%	114.85%	113.90%
CALL	20.80%	19.87%	20.52%	20.35%

Figure 11.3: Overhead of using memory operations in secure space (Open Virtualization's TEE) with respect to using them in kernel space (Linux Kernel v.3.8).

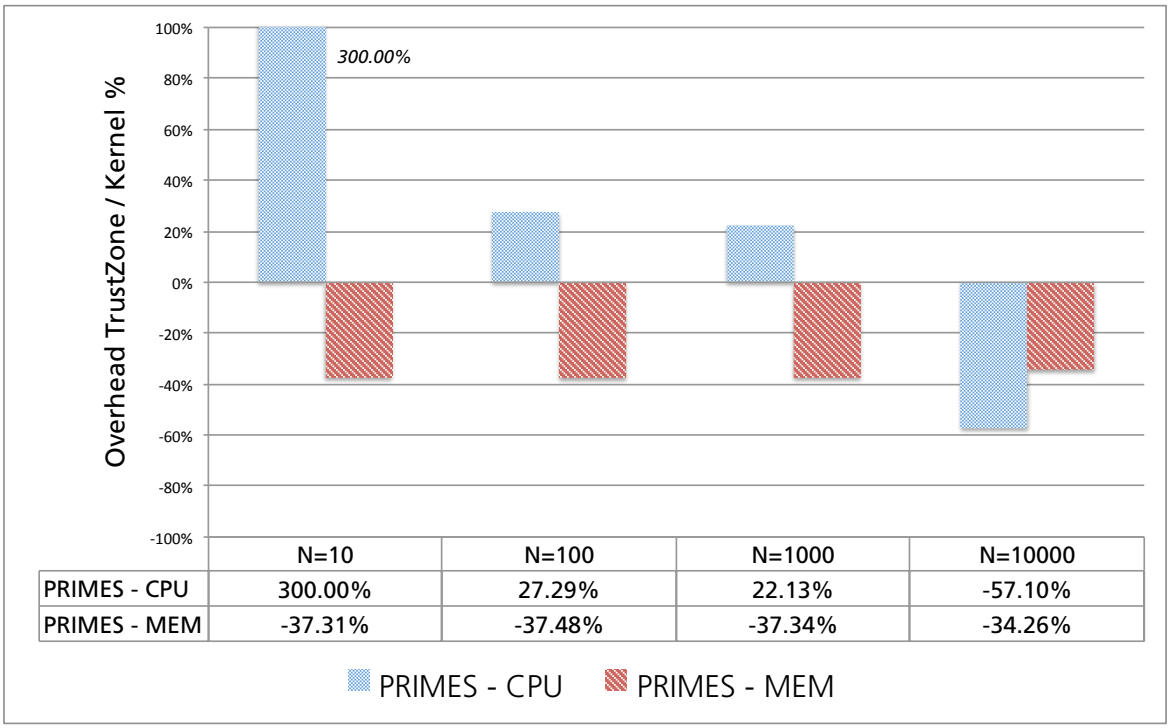
When using *vmalloc*, the kernel allocates virtually contiguous memory that may or may not be physically contiguous; with *kmalloc*, the kernel allocates a region of physically contiguous (also virtually contiguous) memory and returns the pointer to the allocated memory [77]. While *kmalloc* is normally faster that *vmalloc* it depends very much on the requested allocation size and how fragmented the memory is. In our tests *kmalloc* clearly deteriorates faster than *vmalloc*; we execute each memory operation a million times in order to avoid misleading results due to scheduling, locking, etc. When looking at individual allocations, *kmalloc* does perform better for low allocation sizes. We believe that this workload is representative of the intensive memory usage that results from switching contexts between kernel and secure spaces in innovative, complex applications.

The secure area cannot exploit this advantage when copying memory though. While, theoretically, a secure space built on top of TrustZone should have the same resources as kernel space, in practice TEEs are designed to work with few resources under low duty-cycles. Sierraware's TEE starts to show problems when the secure task heap size is increased over 64MB. What is more, the secure software stack is designed with compactness (low TCB) and verifiability in mind, not performance. Sierraware's TEE exhibits a high overhead when carrying out memory copies.

The overhead of making a call to a secure function is a sub-product of our measurements and is included as reference, since it is an overhead that must be paid every time that a secure

task is invoked. As expected the overhead is constant (∼20%). More interestingly, this experiments confirms that the overhead of issuing a call to the secure area is indeed in the range of a few microseconds, as we had previously inferred from the application benchmark.

CPU-hungry Operations Trusted modules in the secure area combine all sorts of operations, including CPU-hungry computations. Figure 11.4 shows the overhead of executing algorithms in secure space. We experiment with two different algorithms to calculate the first n primes: a brute-force algorithm (CPU-hungry), and the Sieve of Eratosthenes [214] (memory-hungry).



	N=10	N=100	N=1000	N=10000
PRIMES - CPU	300.00%	27.29%	22.13%	-57.10%
PRIMES - MEM	-37.31%	-37.48%	-37.34%	-34.26%

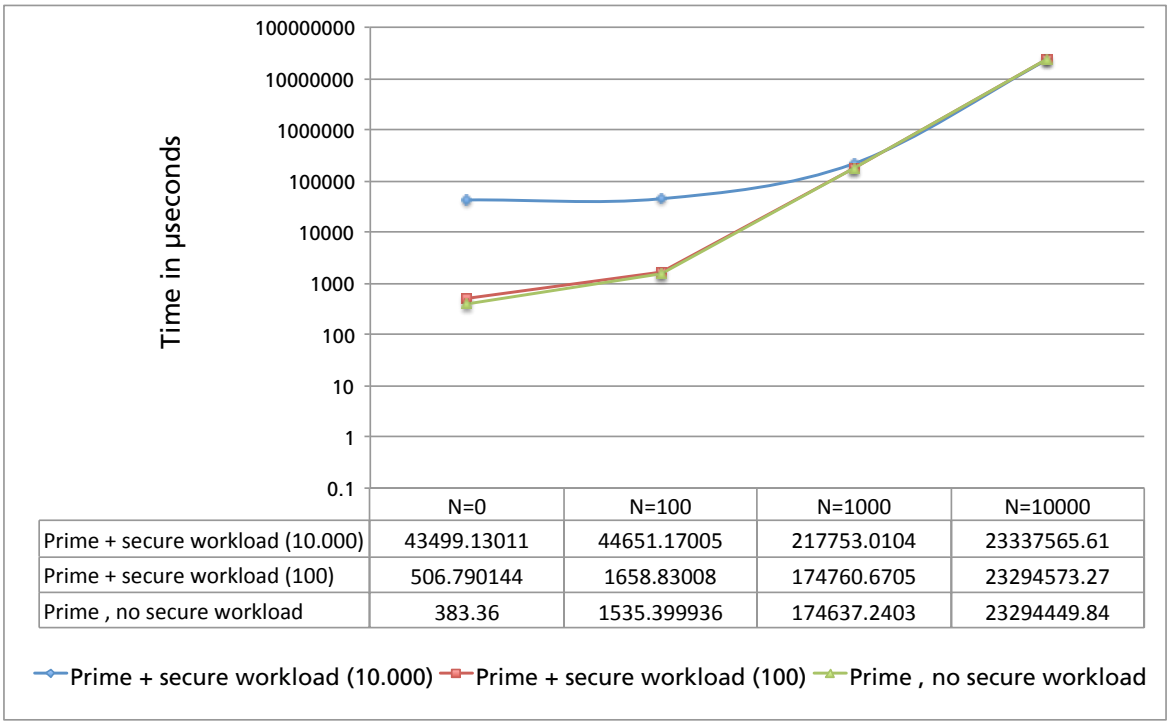
Figure 11.4: Overhead of calculating the first N primes using a CPU-hungry algorithm and a memory-hungry one.

We observed that at low CPU usage, execution in secure space is much more expensive than execution in kernel space. When exposed to a high CPU usage, though, secure space outperforms kernel space. This is because the secure area is never preempted. One of the traditional TEE design principles, as defined by Global Platform, is that secure space has higher priority than kernel space. This is (erroneously) achieved by disabling interruptions when secure tasks are executing (see Section 7.1.1.2 for details). In summary, on ARMv7 there are two types of interrupts: external interrupts with normal priority (IRQ), and fast external interrupts with high priority (FIQ) [21]. In TrustZone, FIQs are normally associated with secure space, and IRQs with kernel space [29]. Open Virtualization disables IRQs when entering secure space. As a result, long-running secure tasks are not scheduled out of execution. Coming back to the prime experiment, when performing the calculation of first 10000 primes using a brute force algorithm, the secure area takes advantage of the TEE not

being preempted, therefore outperforming kernel space, at the cost of missing IRQs.

The Sieve of Eratosthenes on the other hand, is a memory-hungry algorithm which has a memory requirement of $O(n)$, where n is the size of the buffer and the square of the largest prime we are looking for. In our experiments we use a constant buffer, because we want to study how memory operations compare to CPU operations in terms of overhead. The results shown in Figure 11.4 follow the trend in Figure 11.3. Understanding the decisions behind the design of the TEE supporting the secure area is crucial in order to design trusted services. In the case of Open Virtualization, memory - while a limited resource -, has a beneficial impact on performance, as opposed to intensive CPU operations.

The consequence of secure tasks not being preemptable can be better seen in Figure 11.5, where a kernel space task, and a secure space task compete for resources. Here we can see that, when a secure task executes a heavy workload, the execution of a rich application is delayed proportionally to the time that it takes the secure task to finish its execution. TEEs are not designed to run heavy workloads for significant periods of time. This is indeed a strong limitation that affects the design of our trusted modules.



	N=0	N=100	N=1000	N=10000
Prime + secure workload (10.000)	43499.13011	44651.17005	217753.0104	23337565.61
Prime + secure workload (100)	506.790144	1658.83008	174760.6705	23294573.27
Prime , no secure workload	383.36	1535.399936	174637.2403	23294449.84

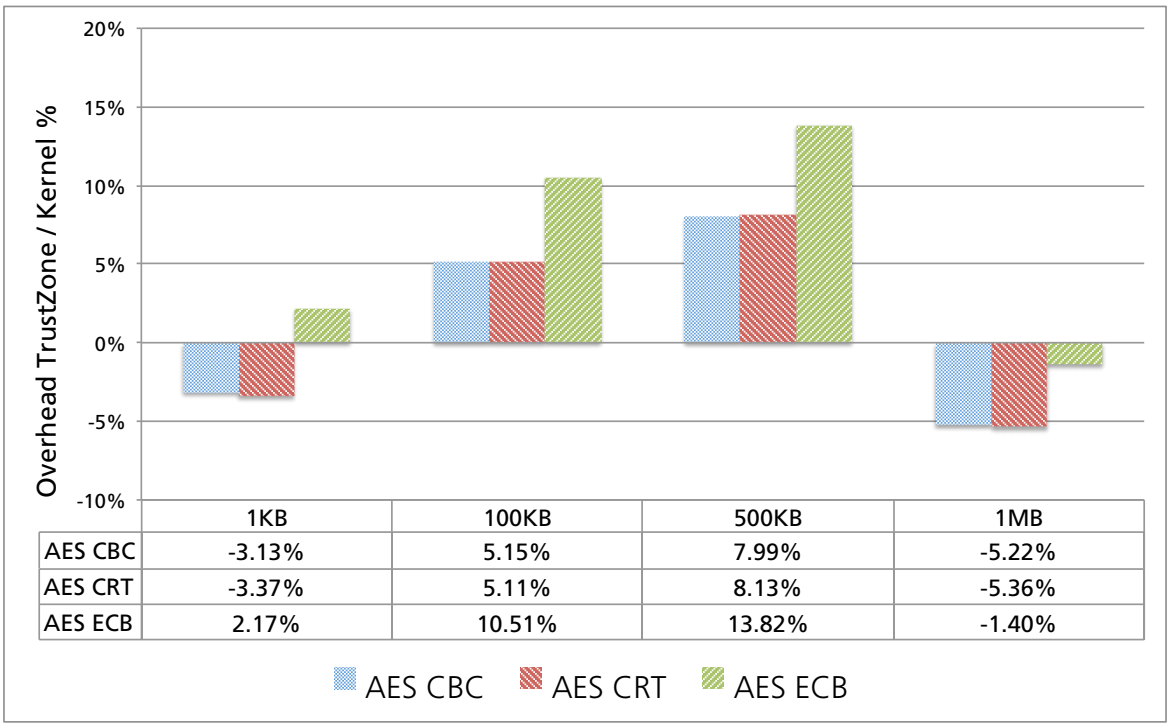
Figure 11.5: Impact of Open Virtualization's TEE disabling IRQ interrupts when executing a secure task.

Cryptographic Operations The last microbenchmark that we present targets cryptographic operations. These operations are relevant in two different ways: (i) they mix memory- and computationally-hungry operations, and (ii) they represent a workload that must be carried out in the secure area by TSM.

For these set of tests we use AES-128, which is the default encryption algorithm in TSM. As

we mentioned in the Trusted Cell design, we choose this algorithm since there is no known attack which is faster than the 2^{128} complexity of an exhaustive search [46]. AES-192 and AES-256 have shown more vulnerabilities even when they use longer keys [47, 46]. In kernel space we use the Linux Crypto API [209]; in secure space, we use the *libcrypto* EVP API available in OpenSSL[15] through TSM interfaces.

Figure 11.6 compares the overhead of using AES with a 128-bit key and CBC, CRT, and ECB cipher modes in secure space. There are two interesting results coming out of this test. First, the trusted-untrusted overhead follows the same trend independently from the cipher block mode used. Second, ECB performs worse than CBC and CRT in the secure area.



	1KB	100KB	500KB	1MB
AES CBC	-3.13%	5.15%	7.99%	-5.22%
AES CRT	-3.37%	5.11%	8.13%	-5.36%
AES ECB	2.17%	10.51%	13.82%	-1.40%

Figure 11.6: Encryption overhead of using the AES algorithm with a 128-bit key and different block ciphers modes. Kernel space relies on Linux Kernel crypto API; secure space uses the *libcrypto* EVP API available in OpenSSL.

The first result might seem confusing since the secure area outperforms the rich area both when encrypting a block of 1KB and a block of 1MB (around 5% faster), while it exhibits an overhead of up to 13% when encrypting a block of 500KB. For every encryption there is a number of structures that need to be allocated (transformation block cipher, scatterlist, initialization vector, ciphertext buffer, etc.). For encrypting 1KB, at least another 1148 bytes are needed - besides the input buffer. This applies for both the Linux Crypto and EVP APIs.

As shown in Figure 11.3, allocating and freeing memory is much cheaper in the secure area. When encrypting 1MB, the amount of computation begins to be significant. In this case, the secure area has the advantage of not being preemptable, as seeing in Figure 11.5.

[15] `https://www.openssl.org/docs/crypto/evp.html`

Regarding the difference exhibited by ECB, ECB is a block cipher that operates on one block at a time. It does not use an initialization vector to kickstart the encryption, and it uses the same algorithm to encrypt each block [265]. This results in ECB using less memory and being more efficient [230], at the cost of being less secure [265, 128][16].

It is difficult to compare *libcrypto* EVP and the Linux Crypto APIs, mostly because they are designed for different contexts. This might explain the lack of a performance study including them both. Even when a performance benchmark is presented, the outcome depends largely on the context (i.e., payload, keys, hardware used, etc), which results in opposite conclusions [153, 212, 106] over time. There might be some aspects of *libcrypto* EVP that outperforms the Linux Crypto API, and viceversa. However, we believe that the tendency depicted in Figure 11.6 responds to the design principles that form a traditional TEE as Open Virtualization (Section 7.1.1). We want to make clear that our results are not meant to serve as a reference for comparing both libraries.

11.2.3 Applicability of Split-Enforcement

Mediating system resources through the Trusted Cell in the secure area comes inevitably at the cost of performance. This cost is a function of (i) the frequency of the calls to the secure area, and (ii) the type and extent of the computation that takes place per call. In the presented benchmarks we measure separately the cost of a *Secure Round Trip* to the secure world, and the overhead of executing a CPU-hungry, memory-hungry, and combinations of them in the secure area. What is more, we present an application benchmark where we take the worst case and mediate *every* system call through the TEE Trusted Cell in the secure area. The results of our experiments show that:

1. The cost of a secure round trip is in the range of the few microseconds (at 666MHz.). Note that a secure round trip not only includes the cost of setting the processor in privilege mode, but also the cost on saving the untrusted context, loading the trusted context, dispatching the secure tasks, and the return to the control flow to the untrusted area (i.e., save trusted context and load the untrusted context).

2. The overhead of 10000s context switches, executing 1000s of instructions in secure space is below 20%. It is thus possible to envisage that the TEE Trusted Cell captures the state of untrusted applications running in user space through a LSM that redirects (some of) their system calls to evaluate their actions and unlock system resources.

3. The Trusted Execution Environment provided by Open Virtualization is not well suited for long-running tasks in secure spaces. Indeed, secure tasks run to completion without being pre-empted. During that time, kernel tasks are delayed, and risk missing IRQs that are disabled. The consequence is that the option where sensitive data is stored and processed in secure space (i.e., secure drivers Section 6.3.1) cannot be supported on

[16]Note that we include ECB for showing, once again, the impact of TEE design decisions, such as preallocating memory in the case of Open Virtualization. We do not consider using ECB cipher mode in our design, since increasing its security, requires using different encryption keys for each block. CBC, which is the default cipher in TSM, does not exhibit this shortcoming.

this platform. In fact, since Open Virtualization follows a traditional Global Platform design, we can state that there is a mismatch between current TEEs that are designed to offload simple secure tasks to the secure area, and the needs of the run-time security community in ARM-powered devices.

4. The performance of encryption in secure space is not dramatically slower than in kernel space; TSM being in charge of crypto operations in the TEE Trusted Cell seems like a good option.

With the results we have obtained, we are now in position to analyze their applicability to the three locking mechanisms forming split-enforcement:

Data locking. Even in datacenters, I/O device latencies today are in the range of few hundreds of microseconds [284]. This means that mediating *each* I/O request through the Trusted Cell - which as mentioned is in the order of a few microseconds -, would represent an overhead of <10%. I/O devices that are not used in edge data centers, are still counted in milliseconds. This is specially true for general purpose personal devices such as smart phones, laptops and set-top boxes. In this last case, the cost of mediating I/Os is one order of magnitude below current I/O device latencies. The trusted storage solution we present is then directly applicable to personal devices at an insignificant cost. What is more, in its actual form, our solution is viable for datacenters with focus on security and privacy.

Memory locking. Memory operations are measured in nanoseconds, therefore a microsecond overhead is not acceptable. Note that this measurements respond to our proof of concept implementation, where a context switch to the secure world is required *per* memory access. The alternative design we propose, based on a hypervisor can lower the impact of mediating memory accesses significantly. Indeed, work on hypervisors show how virtualization overhead can be minimized significantly using current ARM virtualization extensions [81, 152].

Peripheral locking. Access to the ARM's Advanced Peripheral Bus (Section 2.4) is in the order of a few dozens of clock ticks, which at 666MHz corresponds to nanoseconds [225]. However, communication with a peripheral itself depends very much on the peripheral, and in cases where humans are involved (e.g., identification) time is measured in hundreds of milliseconds in the best case. Note that access to I/O devices and memory is also handled through the APB bus; TrustZone's NS bit depends on the proper operation of the AXI-to-APB bridge. A general claim cannot be made since latency depends very much on the peripheral. For us, the peripherals that we secure are limited to sensitive data acquisition since we maintain a low TCB; securing peripherals requires porting secure drivers (Section 6.3.1). Moreover, the fact that in ARMv8 an exception level is dedicated to the secure monitor, allows to explore the possibility of adding simple usage rules directly to the secure monitor to avoid a context switch to the secure area. This would lower latency to the nanosecond scale. Investigating how TUM can be distributed among different devices and exception levels is a topic for future research.

11.3 Discussion

This experimental evaluation covers the design space for the first non-commercial framework available for TrustZone. While our proof of concept implementation does not fully comply with split-enforcement (because of the limitations of the framework), it has helped us (i) expose further limitation in this portion of the design space, and (ii) understand which properties that future hardware and software providing TEEs should exhibit.

From a hardware perspective, the lack of trap-and-emulate instructions in TrustZone, and the lack virtualization extensions in popular TrustZone-enabled processors such as the Cortex-A9 prevent the implementation of trusted memory enclaves. Moreover, the overhead of switching to the secure area per memory access introduces a significant overhead.

From a software perspective, we have seen that traditional TEEs such as Open Virtualization are designed for Global Platform use cases: low duty-cycled, non-preemtable, with non-interruptible secure tasks, and preallocating a memory pool for the secure task heap. Such frameworks are not well suited to provide run-time security as we have defined it. We can expect much less overhead from a secure space implemented on top of a largely independent, high duty cycled secure microkernel. Examples implementing this paradigm include Genode [115], and TOPPERS SafeG [249]. The security community has much to learn from the real-time community and the work they have done with hardware extensions originally conceived for security such as TrustZone. The performance results we have presented should serve as a reference when evaluating these, and next generation TEEs.

Throughout our experiments we have observed that the overhead of mediating I/O operations and peripheral in the Trusted Cell is very acceptable, even when the TEE we are using is not well suited for real-time response. Note that a secure round trip is in the order of a few microseconds. This means that the presented trusted storage solutions and trusted data generation and acquisition with secure peripherals are directly applicable to today's platforms relying on direct attached storage. This type of trusted storage meets in fact all the requirements established by split-enforcement in terms of data confidentiality and integrity (Section 8.2.1.1).

Memory operations on the other hand, suffers more the few microseconds that it takes to context switch to the secure area. This backs our initial thoughts on the need for MMU virtualization in the untrusted area; if TrustZone were to support trap-and-emulate memory accesses, a trusted thin hypervisor could implement memory usage policies and directly mediate memory accesses. This would not only lower overhead, but it would give the possibility to implement memory locking as envisioned in split-enforcement. Such advances in secure hardware are also necessary to support trusted storage techniques in next generation I/O devices, as described throughout Chapter 8. Until TrustZone gets a revision, combining it with virtualization extensions such as the ones presents in ARMv8 processors (e.g., Cortex-A53, and Cortex-57) is a relevant topic for future work.

To conclude, we list the properties in terms of hardware and software that an ARM-powered platform should implement in order to support split-enforcement, and run-time security as we have envisioned in this thesis. We limit ourselves to commercially available hardware,

and omit obvious components such as RAM (note that split-enforcement does not have an impact on memory usage since it does not introduce large structures that should fit in main memory).

- Hardware:
  - A processor that implements both TrustZone and virtualization extensions. This is necessary to mediate memory accesses without killing performance, and providing high security guarantees (Chapter 6). Examples of commercially available processors with this properties include Cortex-A15 (ARMv7), Cortex-A53 (ARMv8), and Cortex-A57 (ARMv8).
  - Several PCI-express interfaces that allow to communicate with next generation I/O devices natively.
  - Internal non-volatile flash memory that can either be virtualized or secured from TrustZone. This allows to implement different levels of trusted storage.
  - A tamper-resistant unit to store encryption keys and other secrets (e.g., Secure Element).
  - Different cache levels that can also be secure. This allows the secure area to cache sensitive data in cleartext.
- Software:
  - A TEE that is largely independent from the untrusted area, high duty cycled, preemptable, and preferably with real-time guarantees to minimize context switch overhead to the secure area. Examples implementing this paradigm include Genode [115], and TOPPERS SafeG [249].
  - A thin hypervisor that fully virtualizes the MMU in the untrusted area. Such hypervisor does not need support for multi-guests, but should implement the part of TUM that corresponds to memory accesses in order to minimize overhead. Related work on secure hypervisors show that such design can be achieved with a small TCB [282].
  - A distributed framework that allows to implement trusted services using different TEEs working concurrently. The Trusted Cell is an example of such framework. This allows, for example, to have a secure area in the I/O device and secure area the host device that collaborate to enforce usage policies. Even when the hypervisor mentioned above is not a secure area, it is trusted. Collaboration between the hypervisor and the host device secure area is also supported by the Trusted Cell framework.

Chapter 12

Conclusion

12.1 Conclusion

In the beginning of this thesis, we argued that one of the main factors enabling cyberattacks was the increasing complexity of software. Our argument was that complexity hides vulnerabilities in the code, causing software to occasionally behave nondeterministically. In our view, cyberattacks are indeed about detecting unspecified behaviors and finding ways to exploit them. The question that we asked, and that has served as motivation for our work, was: *How do we handle the security of complex software without killing innovation?*

In this thesis we answered this question by focusing on run-time security. While we hope that future developers with the help of better development tools will produce honest, bug-free code, we assume that innovative services will, for the time being, exhibit unspecified behavior. In order to protect the *Sensitive Assets* that motivate cyberattacks, we propose the use of *Run-Time Security Primitives* to enable *Trusted Services*. These allow sensitive assets to be directly used by untrusted services, but mediate actions involving them in order to guarantee their integrity and confidentiality. Enabling trusted services in commodity operating systems has been the focus of this thesis. In the process of addressing these problem, we have produced three major contributions that we summarize next. We present their design and implementation in Part II and their analytical and experimental evaluation in Part III. All in all, this thesis provides an updated look at the run-time security landscape from boot protection mechanisms to run-time policy enforcement. Part I is dedicated to covering the state of the art, but we extend it with comparisons and discussions through the whole thesis.

Split-enforcement. Some work have proposed to protect sensitive assets by dividing the execution environment in two areas: one trusted and one untrusted. We follow the same approach, but require that these areas are separated by hardware. We denote them *Trusted Area* and *Untrusted Area*. Also, instead of restricting the use of sensitive assets to components in the trusted area, we use these trusted components to mediate actions carried out directly by innovative services in the untrusted area which involve sensitive assets. This way

we do not limit the scope of innovative services. We denote the mechanism we use to achieve this *Split-Enforcement*. This is our first contribution (Chapter 6).

Generic TrustZone driver for Linux Kernel. We implement split-enforcement using ARM TrustZone security extensions. While work on TrustZone-enabled TEE frameworks has gained traction in the last years, there is still no generic support for the Linux kernel. Apart from contributing to Open Virtualization, the TEE framework that we have used, we have proposed the first generic TrustZone support for the Linux operating system. At the time of this writing, we have submitted a first set of patches with the generic TrustZone driver to the *Linux Kernel Mailing List (LKML)* with the intention of contributing our work to the mainline Linux kernel. This is our second contribution (Chapter 7).

Trusted Cell We finally implement a distributed framework that leverages the capabilities of a given TEE to provide trusted services using TrustZone. We denote this framework *Trusted Cell*. Using this framework, we implement two trusted services that we consider pivotal to run-time security: (i) a trusted storage solution that ensures the confidentiality and integrity of sensitive data while reusing legacy infrastructure to minimize the *Trusted Computing Base (TCB)*; and (ii) a reference monitor that allows to enforce *Usage Control* policies targeting a wide range of sensitive assets in a device. While building the Trusted Cell, we identified a number of deficiencies in our platform affecting the hardware, software, and also TrustZone's design. We explored the design space we initiated despite these limitations. This has allowed us to better reason about the hardware and software combination that can support future iterations of our work. This is our third contribution (Chapters 8, and 9).

12.2 Future Work

Based on our experience designing and building support for trusted services, we now propose a roadmap for future work. As we have argued during Part III, split-enforcement is in fact a good solution to provide run-time security in commodity OS without making assumptions on their trustworthiness. Moreover, our experimental results show that it introduces an affordable overhead in terms of performance. We propose future work based on our initial hypothesis that a state machine can capture utilization of sensitive assets, and be serve as a basis for usage policies enforcement by means of split-enforcement(Section 6.2.

We divide this roadmap for future work in three sections. First, we look at what we can improve in the Trusted Cell framework in terms of OS support and trusted modules. Second, we describe the necessary components required to improve the two trusted services that we have already implemented. Here, we look at the reference monitor and trusted storage solutions separately. Finally, we propose a number of trusted services that we had in mind when implementing the Trusted Cell, and that have, in a way, shaped the Trusted Cell framework.

12.2.1 Linux TEE support and Trusted Modules

While we believe that our design and implementation of the generic TrustZone driver and the Trusted Cell framework (i.e., trusted modules) are very complete, there is a number of improvements that should be done before these components reach a commercial level.

In terms of the generic TrustZone driver, there are three issues that we need to addressed. First, we need to work on the feedback we obtained from the Linux kernel community and resubmit the patches. The process of making a new driver part of the mainline Linux kernel is long and tedious, and requires to improve the code in terms of correctness and understandability. Put differently, our driver requires some "plumbing" before it gets accepted in the Linux kernel. As we have mentioned, we are in the process of collaborating with Linaro to push this effort and make the driver mainstream. Second, once the generic interface is par of the mainline Linux kernel, we need to port more TEE drivers to support as many TEE frameworks as possible. Since we are working with Linaro, support for OP-TEE is at the top of the list. Finally, third, we need to divulge our work both in the Linux and research communities so that people interested in experimenting with TrustZone can focus their efforts in one place. The idea is to work on Linux-specific presentations, kernel documentation, and LWN articles, as well as to write academic publications. We are indeed in the process of writing a paper to present the design and implementation of the TrustZone driver at the moment.

In terms of the Trusted Cell framework there are two areas that require improvement:

Open Virtualization. We would like to continue with the improvements we have already done to Open Virtualziation and contribute to the TEE community. We still need to implement an abstraction layer to liberate developers from dealing with the stack to pass parameters between TrustZone's secure and non-secure world (Section 7.1.1.1). Also, we would like to improve the SMP mode (Section 11.1) to take advantage of the two processors in the untrusted area. Still, this work depends on the license under which Sierraware plans to release future version of Open Virtualization (Section 7.1.1.6).

Trusted Modules. There are also minor improvements that can be done to TSM, TIM, and TUM. As we mentioned when we presented TSM, we intend to make cryptographic operations library-agnostic, borrowing the design of the Linux kernel Crypto API [209]. In this way, other trusted modules would not depend on OpenSSL directly. In relation to TIM, we would like to apply our experiences with calculating file hashes using Merkle trees [205] to improve IMA [75] in the Linux kernel. In TUM we would like to substitute our policy engine with a full implementation of UCON$_{ABC}$. Finally, in terms of the framework in itself, we intend to work on a LSM stacking solution so that the Trusted Cell could coexist with current access control frameworks such as AppArmor [216] or SELinux [263].

12.2.2 Current Trusted Services

With regards to out reference monitor and trusted storage solution, future work focuses on addressing the platform deficiencies that we have encountered when implementing them, which have prevented us form fully implementing split-enforcement. Let us look at each trusted service independently.

Reference Monitor. A necessary step towards a full implementation of split-enforcement is providing memory locking (Section 8.2.2.1). In the process of exploring the whole design space with our hardware and software platform we identified that one of its major deficiencies is the lack of virtualization trap-and-emulate instructions. Counting with this hardware support is necessary to implement memory locking. We have two strategies to achieve this. The first one is to explore ARMv8-powered processors featuring both TrustZone and virtualization extensions. Examples of such processors include Cortex-A53 and -A57. This would allow is to add a thin hypervisor to the TCB and support trusted memory enclaves. The second option is to use our work to push for virtualization extensions being included as part of TrustZone in its revision. In the long run this would be the preferred solution.

If trusted memory enclaves are supported, then the next step would be to fully implement CPI [182]. This would represent the formal model behind memory locking that virtualizating the MMU would enforce. Having this in place, Iago attacks [60] as discussed in Chapter 11 would be canceled. What is more, if new attacks appear, the model could be exchanged. As a part of split-enforcement, memory locking is modular. This is, the enforcement mechanism is orthogonal to the model that generates the usage decisions, CPI in this case.

Trusted Storage. While our trusted storage solution guarantees the confidentiality and integrity of sensitive data generated in the trusted area, next generation I/O devices are able to enable a full storage solution. The advent of I/O devices featuring a full software stack and supporting software defined storage solutions represent a perfect platform for implementing a full trusted storage solution. These devices are powered by ARM processors, which allows us to replicate our architecture in them. Here, only the trusted modules that refer to storage usage policies are necessary; there is no need to replicate the whole Trusted Cell architecture. In this way, I/O requests would directly be mediated by trusted area in the storage hardware. This architecture can enable availability and durability of sensitive data. Specific challenges include:

- **I/O Trusted Cell.** While our intuition is that only a subset of TSM, TIM, and TUM (storage usage polices) are necessary to enable a reference monitor in the I/O device, we cannot make statements before we have implemented a prototype. We believe that we will face the same issues as when implementing the Trusted Cell in parallel with a commodity OS. However, latency an throughput issues that are intrinsic with storage devices can become a big adoption issue. Future work will judge the viability of this solution.
- **Trusted Cells Communication.** In order to guarantee durability of sensitive data,

we need to first make sure that sensitive data reaches the storage device; note that we still reuse the untrusted I/O stack to reduce the TCB. We believe that the two trusted areas could communicate and coordinate using the APB bus and the AXI-to-APB bridge to secure the communication. This would allow to exchange encryption keys, but it also would allow to acknowledge I/O completion, probably together with the hash of the received page. Again, while this solution looks viable in paper, we have not had the chance to implement a working prototype and evaluate it. However, our experimental measurements in terms of I/O overhead (Chapter 11) make us being enthusiastic about this research path.

- **Trusted Area DMA.** While acknowledging I/O completions together with I/O mediation in the storage hardware can guarantee durability, availability can be affected when reading sensitive data. In order to solve this, we have thought about the possibility of implementing DMA between the two trusted areas using the same APB infrastructure as mentioned above. However, we are not sure about the repercussion that this would have in terms of the TCB. As we have argued several time in this thesis, confidentiality and integrity are our top priorities, and any increase of the TCB that would lower them is not an option. Exploring this and other alternatives to provide availability is indeed a big challenge, and a topic of research in itself.

12.2.3 New Trusted Services

Finally, we provide a list of trusted service that we believe would improve the Trusted Cell. Some of these trusted service proposals reflect challenges that the security community inside and outside the academia are facing today. We have ideas regarding the requirements for these trusted services, but no clear vision on specific implementation challenges. Thus, we limit our presentation to a problem description, and occasional approach possibilities.

Trusted Service Installer. Since its apparition, both the industry and the academia have seen in TrustZone a platform to implement (equivalents to) trusted services in mobile platforms: From DRM to mobile banking or user identification. Indeed, recent work still focuses in these use cases [191, 229]. However, all these approaches (including our Trusted Cell) assume that the components enabling these trusted services are already present in the trusted area. In fact, commercial products implementing trusted services with TrustZone rely on manufacturers incorporating these services in their fabric configuration. A fundamental missing part is what we have referred to as a *Trusted Service Installer (TSI)*. We envision TSI as a trusted packet installer that allows to install *Trusted Services* in the Trusted Cell. Such an installer should be able to: (i) install the necessary *Trusted Modules* to support the trusted service, (ii) install the *Run-Time Security Primitives* in *Trusted System Primitive Module (TSPM)*, and (iii) install the user space API to support these run-time security primitives from user space. If the used TEE API is proprietary, and the *Rich Execution Environment (REE)* supports it, TSI should also install the (equivalent to) the Loadable Kernel Module (LKM) to implement the API. If TrustZone is to bring general purpose trusted services to ARM-powered devices, an equivalent to the TSI is necessary.

Trusted Service Store. Related to the previous, such a trusted service installer would trigger the need for a trusted service store. This opens a whole range of possibilities in terms of defining the trustworthiness of trusted services and the modules implementing them, as well as the trustworthiness of untrusted applications. If such a trusted service store would integrate with normal application stores such as App Store or Google Play, leaky and malicious applications as well as applications exhibiting unspecified behavior could be reported in the same way that *App reviews* work today. If the trusted cell detects an untrusted application not complying with its *Application Contract* it would directly report it to the application store, using reputation as a countermeasure against malicious and careless developers. We believe that the possibilities of such application store (including the trusted service store) are many, and could be equally used in mobile devices and cloud services.

Trusted Are Intrusion Detection. Detecting intrusions both in the untrusted and the trusted areas is also an area where the Trusted Cell could be useful. As we briefly describe in Section 8.2.1.1 secure containers could be used as bait to detect DoS or corruption attacks targeting the REE Trusted Cell. Machine learning techniques that are fed with access patterns, usage control policy generation, and techniques such as the one described with secure containers are a promising way to push trusted storage from resilience to *antifragility* [268].

Trusted Cell Verification. As soon as we count on a number of Trusted Cells being distributed either locally in the same device (e.g., host trusted cell + I/O device trusted cell) or remotely (e.g., devices over the Internet) verifying Trusted Cells is an issue. Traditionally, this problem has been solved by assigning unique keys at fabrication time by OEMs. However, depending on third parties that might be under the coercion of organizations of governments [142] is not a good root of trust. In general, authenticating (the equivalent) to the trusted area without depending on a third party is a challenge for the security community.

Split-Enforcement in other Secure Hardware. Finally, the apparition of advance secure hardware such as Intel SGX security extensions make us wonder how split-enforcement could be materialized in them. While our understanding is that SGX does not extend to the system bus, thus not supporting securing peripherals, investigating how to extend it to fully support split-enforcement is an interesting topic for future research. Indeed, one of the main issues that we have encountered when implementing split-enforcement in ARM TrustZone is related to the creation of trusted memory enclaves, which are by default supported in Intel SGX.

Glossary

Application Contract Contract in which an application explicitly states the assets it is going to use. The focus of the contract is that applications specify which sensitive information (e.g., contact list, archived mails), software resources (i.e., libraries), and peripherals (e.g., microphone, secondary storage) they are going to use. We refer to these as *Sensitive Asset*. In platforms such as Android, this is already done within the application manifest[1]. In [97], Dragoni et al. also propose the use of contracts that are shipped with applications. For us this contract is not enforcing, but as we will see, it can help the reference monitor to classify applications when they violate the device usage policy. Since the applications we refer to here are all untrusted, we sometimes refer to them as untrusted applications to emphasize this fact. We reserve the right to also denote them as (untrusted) services, specially when we refer to applications running as cloud services.

Certainty Boot A boot protection mechanisms that allows users to choose the OS they want to run in their devices while (i) giving them the certainty that the OS of their choice is effectively the one being booted, and (ii) allowing applications to verify the OS at run-time. Certainty boot is based on Cory DOctorow's proposal in [95].

Commodity OS Constitutes the operating system in the untrusted area. We do not make distinction between monolithic kernels, microkernels, or hybrids [261, 90]. The commodity OS is by definition untrusted. Typically, it is large, and complex. Examples of commodity OSs are Linux, BSD, or Mac OS.

Default Application Contract *Application Contract* with a default set of usage policies.

Device In this thesis we refer to device as an autonomous computer machine that is composed by different hardware components and provides an execution environment, where it is giveng a set of inputs and returns a set of outputs. Our assumption in this thesis is that a device can be physically divided between a *Trusted Area* and an *Untrusted Area* in order to isolate security-critical operations from the main OS and applications. Examples of devices include smart phones, set-top boxes, and laptops. In Chapter 7 we take more classical definition to refer to components that *attach* to a computer machine (e.g., secondary storage hardware, network cards) since it is the definition used in the systems community when referring to the drivers governing these components.

[1] `http://developer.android.com/guide/topics/manifest/manifest-intro.html`

Device Usage Policy List of usage policies that define the behavior of a device. These policies define how the device should behave when interacting with other systems (e.g., which information to share, protocols, privacy settings). These policies affect the device as a whole.

First Stage Boot Loader (FSBL) First logical component that executes in a device. It is normally used to initialize hardware components. It is normally proprietary.

Linux Kernel Mailing List (LKML) Set of mailing lists where kernel developers submit patches and discuss design and implementation details of the different kernel submodules. A list of the current mailing lists and information on how to subscribe to them can be found here: `http://vger.kernel.org/vger-lists.html`. Latest messages in: `https://lkml.org`.

LSM Trusted Cell Subpart of the *REE Trusted Cell* that handles connections with the Linux kernel through the Linux Security Module (LSM) framework.

REE Trusted Cell Part of the *Trusted Cell* that belongs in the *Rich Execution Environment (REE)*.

Reference Monitor Component that mediates access to *Sensitive Assets* based on a given usage policy. It is also a *Trusted Service* in the *Trusted Cell* that is in charge of enforcing usage decisions generated by *Trusted Usage Module (TUM)*.

Rich Execution Environment (REE) Execution environment in the *Untrusted Area* of a *Device*.

Run-Time Security Set of protection mechanism that guarantee the integrity of a system at run-time. The mechanisms to achieve this are diverse. An example is *Split-Enforcement*.

Run-Time Security Primitives System primitives that explicitly refer to security-critical operations in a running system. A system primitive in this context is composed by (i) an operation in the trusted area (e.g., *Trusted Service* or *Secure Task*, a *Secure System Call* supporting it in the commodity OS, and the user space library that exposes it to *Untrusted Applications*.

Second Stage Boot Loader (SSBL) Logical component executed after the *First Stage Boot Loader (FSBL)*. It finishes hardware initialization before executing what is normally referred to as the OS bootloader (e.g., uboot). It is normally proprietary.

Secure Area It is a portion of the trusted area that defines an execution environment. In other words, it is the software part of the trusted area. The secure area is synonymous with a TEE. In some secure hardware platforms such a smart card, the trusted area and the secure area are the same, but this is not always the case (e.g., ARM TrustZone).

Secure Container A encrypted container formated and used *Trusted Security Module (TSM)* to provide trusted storage for sensitive data generated in the *Trusted Area*. Secure containers are typically stored in untrusted secondary storage in form of files.

Secure Device Tree Device tree describing the hardware composing the trusted area.

Secure Element (SE) Special variant of a smart card, which is usually shipped as an embedded integrated circuit in mobile devices together with Near field Communication (NFC).

Secure OS Constitutes the operating system in the *Secure Area*..

Secure Round Trip It describes the process of unloading and saving the untrusted software stack, executing TrustZone's secure monitor (SMC call), loading the trusted stack, and coming back to the untrusted area (i.e., unloading and saving the trusted stack, executing the SMC call, and loading the untrusted stack). We use this as a metric to evaluated the performance overhead of executing a trusted service or secure task in Chapter 11.

Secure System Call System call that explicitly refers to internal security-critical operations. It is the way that untrusted applications can trigger *Trusted Services*. They represent the user space support for *Run-Time Security Primitives*.

Secure Task It is a well-defined piece of code that executes in the secure area. A secure task has a very specific purpose both in terms of computation and interaction with other components.

Sensitive Asset An asset in a device that is considered sensitive because it either generates, manages, or contains sensitive information. Sensitive information (e.g., encryption keys, mails, photographies) is a sensitive asset in itself. Examples of other sensitive assets include pin- and password-entry devices such as keyboards and touch screens, cameras, microphones, and biometric sensors. Sensitive assets are the trophy for an attacker aiming to compromise a system..

Sensitive State In the state machine representing the usage of sensitive assets (Figure 6.1), untrusted state represents the state where the untrusted application has access to sensitive assets. The untrusted application reaches this state if and only if the reference monitor has allowed it. This decision is based on a usage request, accessibility to other sensitive assets, and the device's context.

Software Defined Storage (SDS) Approach to data storage in which the software controls storage-related tasks is decoupled from the storage device (i.e.,hardware). SDS enables the host to enforce storage policies and implement logic that has traditionally been implemented in the Flash Translation Layer (FTL) such as data placement or garbage collection. From a security perspective, these storage policies resemble TUM usage policies, which allows for more control over I/O requests, thus enabling novel trusted storage solutions..

Split-Enforcement A two-phase enforcement mechanism that allows to enforce usage policies without increasing the TCB in the *Secure Area* with replicated functionality (e.g., I/O stack). In the first phase, *Sensitive Assets* are locked are boot-time. In the second phase, these sensitive assets are unlocked on demand at run-time. Since the locking and unlocking mechanisms reside in the secure area, *Untrusted Applications* are forced to

go through the *Reference Monitor*, thus being subject to the usage decision generated by it.

TEE Trusted Cell Part of the *Trusted Cell* that belongs in the *Trusted Execution Environment (TEE)*.

Trusted Area An area in a device defined by a hardware security perimeter containing an execution environment and a number of hardware resources to support it (e.g., CPU, memory, peripherals.). The trusted area is not intrinsically *trusted*; no software executing in it, and no hardware attached to it offers more guarantees than an equivalent outside of the security perimeter. However, since the trusted area is separated by hardware, its isolation is guaranteed. The level of isolation is given by the specific technology supporting it (Chapter 2). In this way, software and hardware outside the trusted area cannot directly access the security perimeter; they need to use the interfaces defined by it. It is then a software design issue to define the right interfaces in order to guarantee the integrity of the trusted area. From an architectural perspective, if a piece of software or hardware is considered trusted, the trusted area guarantees its immutability against a set of logical and physical attacks. These can be defined as a function of (i) the hardware defining the security perimeter, and (ii) the software managing it.

Trusted Cell Distributed framework that leverages the capabilities of a given *Trusted Execution Environment (TEE)* to provide *Trusted Services*. The Trusted Cell is divided between the *REE Trusted Cell* and the *TEE Trusted Cell*, which correspond to the untrusted and trusted part of the Trusted Cell respectively.

Trusted Computing Base (TCB) TCB is normally defined as everything in a computing system that provides a secure environment. This includes hardware, firmware, and software.

Trusted Execution Environment (TEE) Execution environment in the *Trusted Area* of a *Device*. The TEE is not intrinsically trusted; no software executing in the TEE offers more guarantees than an equivalent outside of the TEE. Code executing in the TEE is guaranteed not to be tampered with as a function of the level of tamper-resistance exhibited by the hardware defining the trusted area that supports the TEE.

Trusted Integrity Module (TIM) *Trusted Module* in charge of maintaining integrity measurements in the *TEE Trusted Cell*.

Trusted Memory Enclave Is a piece of memory accessible from the *Commodity OS* which from a limited period of time is considered trusted, and can be attested by the *Secure Area*. In oder words, it is a trusted enclave in the untrusted address space. The secure area guarantees the integrity of trusted memory enclaves, and enforces that any memory operation involving it is mediated by the reference monitor. We borrow the name from Intel SGX's enclaves (Section 2.3.2).

Trusted Module Component in the *TEE Trusted Cell* that serves to one or several *Trusted Services*.

Trusted Security Module (TSM) *Trusted Module* in charge of cryptographic operations in the *TEE Trusted Cell*.

Trusted Service Services implemented in the *Trusted Area* that involve the use of *Sensitive Assets*. Trusted Services are provided by means of *Trusted Modules*. An example of a trusted service is trusted storage, which is largely supported by the *Trusted Security Module (TSM)* trusted module.

Trusted Service Installer (TSI) It is a packet installer that allows to install *Trusted Services* in the Trusted Cell. Such an installer should be able to: (i) install the necessary *Trusted Modules* to support the trusted service, (ii) install the *Run-Time Security Primitives* in *Trusted System Primitive Module (TSPM)*, and (iii) install the user space API to support these run-time security primitives from user space. If the used TEE API is proprietary, and the *Rich Execution Environment (REE)* supports it, TSI should also install the (equivalent to) the Loadable Kernel Module (LKM) to implement the API..

Trusted Storage A *Trusted Service* in the *Trusted Cell* that guarantees the confidentiality and integrity of sensitive data. It still needs to guarantee durability and availability of sensitive data to be a complete storage solution.

Trusted System Primitive Module (TSPM) *Trusted Module* in charge of defining and dispatching *Run-Time Security Primitives* in the *TEE Trusted Cell*. TSPM acts in general as a Trusted Cell manager in the *Trusted Area*.

Trusted Usage Module (TUM) *Trusted Module* in charge of the usage control model in the *TEE Trusted Cell*.

TrustZone Protection Controller (TZPC) Virtual peripheral in the TrustZone security extensions that is attached to the AXI-to-APB bridge and allows to configure secure peripherals on the fly. More information: `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0448h/CHDHFEHG.html`.

Two-Phase Boot Verification Mechanism that enables *Certainty Boot*.

tz_device User space interface that allows to communicate with the generic TrustZone driver in the Linux kernel. Other interfaces such as Global Platform Client API can be built on top of it to provide higher abstractions.

Untrusted Application User application in the *Untrusted Area*. While untrusted applications represent any application, in most cases we refer to them when describing innovative applications exhibiting large complexity. The assumption is that an untrusted application is either buggy, indeterministic, or malicious, and executes on top of a *Commodity OS* that is either buggy, indeterministic, or malicious. Thus, untrusted applications are always untrusted.

Untrusted Area Everything outside the security perimeter defined by the *Trusted Area*. The untrusted area is the typical execution environment where hardware and software resources are shared and multiplexed by the kernel to serve applications. While isolation can be provided for the whole environment, no hardware-supported perimeter

isolates a portion of the environment. Note that hardware virtualization does not provide the type of isolation we are referring to in the trusted area.

Untrusted Service Synonym with *Untrusted Application*.

Untrusted State In the state machine representing the usage of sensitive assets (Figure 6.1), untrusted state represents the state where the untrusted application outsources a specific task to the secure area (i.e., executes a secure task). The secure task takes the control flow, executes, and returns it to the untrusted application when it finishes. This corresponds to traditional secure co-processor's use cases where sensitive assets are only accessible from within the TEE.

Untrusted State In the state machine representing the usage of sensitive assets (Figure 6.1), untrusted state represents the state where the untrusted application has no access to sensitive assets. The untrusted application reaches this state either because it has not made use of any sensitive assets, or because it has released them.

Usage Control Model that allows to mediate *how* a subject uses an object. See Section 3.1.

Usage Engine A state machine that generates usage decisions based on contextual information, i.e., state, assets, and actions. States are depicted in Figure 6.1; the state machine representing the usage engine is depicted in Figure 6.2.

Bibliography

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[2] A. Acquisti and R. Gross. Imagined communities: Awareness, information sharing, and privacy on the facebook. In *Privacy enhancing technologies*, pages 36–58. Springer, 2006.

[3] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. *ACM Sigplan Notices*, 41(11):2–13, 2006.

[4] S. Aissi et al. Runtime environment security models. 2003.

[5] T. Allard, N. Anciaux, L. Bouganim, Y. Guo, L. Le Folgoc, B. Nguyen, P. Pucheral, I. Ray, I. Ray, and S. Yin. Secure personal data servers: a vision paper. *Proceedings of the VLDB Endowment*, 3(1-2):25–35, 2010.

[6] T. Alves and D. Felton. Trustzone: Integrated hardware and software security. *ARM white paper*, 3(4), 2004.

[7] AMD. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, 3.23 edition, May 2013.

[8] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy, HASP*, volume 13, 2013.

[9] N. Anciaux, M. Benzine, L. Bouganim, P. Pucheral, and D. Shasha. Ghostdb: querying visible and hidden data without leaks. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 677–688. ACM, 2007.

[10] N. Anciaux, P. Bonnet, L. Bouganim, B. Nguyen, I. Sandu Popa, and P. Pucheral. Trusted cells: A sea change for personal data services. *CIDR*, 2013.

[11] N. Anciaux, L. Bouganim, P. Pucheral, Y. Guo, L. Le Folgoc, and S. Yin. Milo-db: a personal, secure and portable database machine. *Distributed and Parallel Databases*, 32(1):37–63, 2014.

[12] J. P. Anderson. Computer security technology planning study. volume 2. Technical report, DTIC Document, 1972.

[13] R. Anderson, M. Bond, J. Clulow, and S. Skorobogatov. Cryptographic processors-a survey. *Proceedings of the IEEE*, 94(2):357–369, 2006.

[14] M. Andreessen. Why software is eating the world'. *Wall Street Journal*, 20, 2011.

[15] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh. Cells: a virtual mobile smartphone architecture. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 173–187. ACM, 2011.

[16] W. Arbaugh. Trusted computing, 2005.

[17] W. Arbaugh, D. Farber, and J. Smith. A secure and reliable bootstrap architecture. In *Symposium on Security and Privacy*, pages 65–71, May 1997.

[18] W. Arbaugh, A. Keromytis, D. Farber, and J. Smith. Automated recovery in a secure bootstrap process. *Technical Reports (CIS)*, 1997.

[19] W. A. Arbaugh, D. J. Farber, A. D. Keromytis, and J. M. Smith. Secure and reliable bootstrap architecture, Feb. 6 2001. US Patent 6,185,678.

[20] ARM. Amba® axi protocol v1.0, March 2004.

[21] ARM. Cortex cortex-a9 technical reference manual. In *Revision: r2p0*. 2009.

[22] ARM. *ARM® Cortex®-A15 MPCoreTM Processor. Technical Reference Manual. Revision:r4p0*, June 2013.

[23] ARM. *ARM® CortexTM-M4 Processor*, June 2013.

[24] ARM. Smc calling convention system software on arm ® platforms, 2013.

[25] ARM. *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition (A3-142 - Processor privilege levels, execution privilege, and access privilege)*, May 2014.

[26] ARM. *ARM Architecture Refrence Manual, ARMv8, for ARMv80A architecture profile (D1.1 Exception Levels), Beta*, 2014.

[27] ARM. Arm® cortex®-a53 mpcore processor. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0500d/DDI0500D_cortex_a53_r0p2_trm.pdf`, 2014.

[28] ARM. Arm® cortex®-a57 mpcore processor technical reference manual. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0488c/DDI0488C_cortex_a57_mpcore_r1p0_trm.pdf`, 2014.

[29] ARM Security Technology. Building a secure system using trustzone technology. Technical report, ARM, 2009.

[30] T. D. Association et al. Truecrypt-free open-source disk encryption software for windows 7/vista/xp, mac os x, and linux, 2009.

[31] S. Bajaj and R. Sion. Trusteddb: A trusted hardware-based database with privacy and data confidentiality. *Knowledge and Data Engineering, IEEE Transactions on*, 26(3):752–765, 2014.

[32] H. Banuri, M. Alam, S. Khan, J. Manzoor, B. Ali, Y. Khan, M. Yaseen, M. N. Tahir, T. Ali, Q. Alam, et al. An android runtime security policy enforcement framework. *Personal and Ubiquitous Computing*, 16(6):631–641, 2012.

[33] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[34] E. Barker and A. Roginsky. Transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths. *NIST Special Publication*, 800:131A, 2011.

[35] A. Bauer, J.-C. Küster, and G. Vegliach. Runtime verification meets android security. In *NASA Formal Methods*, pages 174–180. Springer, 2012.

[36] M. Bauer. Paranoid penguin: an introduction to novell apparmor. *Linux Journal*, 2006(148):13, 2006.

[37] M. Bauer. Security features in suse 10. 1. *Linux Journal*, (144):42, 2006.

[38] M. Bauer. Paranoid penguin: Apparmor in ubuntu 9. *Linux Journal*, 2009(185):9, 2009.

[39] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pages 267–283. USENIX Association, 2014.

[40] A. Baumann, M. Peinado, G. Hunt, K. Zmudzinski, C. V. Rozas, and M. Hoekstra. Don't trust the os! secure execution of unmodified applications on an untrusted host. `http://sigops.org/sosp/sosp13/wipslist.html`, November 2013.

[41] F. Belanger, J. S. Hiller, and W. J. Smith. Trustworthiness in electronic commerce: the role of privacy, security, and site attributes. *The Journal of Strategic Information Systems*, 11(3):245–270, 2002.

[42] I. Bente, B. Hellmann, T. Rossow, J. Vieweg, and J. von Helden. On remote attestation for google chrome os. In *Network-Based Information Systems (NBiS), 2012 15th International Conference on*, pages 376–383. IEEE, 2012.

[43] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The keccak sha-3 submission. *Submission to NIST (Round 3)*, 6(7):16, 2011.

[44] J. Beshears, J. J. Choi, D. Laibson, and B. C. Madrian. The importance of default options for retirement saving outcomes: Evidence from the united states. In *Social security policy in a changing environment*, pages 167–195. University of Chicago Press, 2009.

[45] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. Depsky: dependable and secure storage in a cloud-of-clouds. *ACM Transactions on Storage (TOS)*, 9(4):12, 2013.

[46] A. Biryukov, O. Dunkelman, N. Keller, D. Khovratovich, and A. Shamir. Key recovery attacks of practical complexity on aes variants with up to 10 rounds. *IACR eprint server*, 374, 2009.

[47] A. Biryukov and D. Khovratovich. Related-key cryptanalysis of the full aes-192 and aes-256. In *Advances in Cryptology–ASIACRYPT 2009*, pages 1–18. Springer, 2009.

[48] M. Bjørling, J. Madsen, P. Bonnet, A. Zuck, Z. Bandic, and Q. Wang. Lightnvm: Lightning fast evaluation platform for non-volatile memories.

[49] M. Bjørling, J. Madsen, J. González, and P. Bonnet. Linux kernel abstractions for open-channel solid state drives. In *NVMW Workshop*, 2015.

[50] M. Bjørling, M. Wei, J. Madsen, J. González, S. Swanson, and P. Bonnet. Appnvm: Software-defined, application-driven ssd. In *NVMW Workshop*, 2015.

[51] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40. ACM, 2011.

[52] B. Blunden. *The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System*. Jones & Bartlett Publishers, 2013.

[53] P. Bonnet, J. González, and J. A. Granados. A distributed architecture for sharing ecological data sets with access and usage control guarantees. 2014.

[54] J. Bonwick et al. The slab allocator: An object-caching kernel memory allocator. In *USENIX summer*, volume 16. Boston, MA, USA, 1994.

[55] L. Bouganim and P. Pucheral. Chip-secured data access: Confidential data on untrusted servers. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 131–142. VLDB Endowment, 2002.

[56] K. Brannock, P. Dewan, F. McKeen, and U. Savagaonkar. Providing a safe execution environment. *Intel Technology Journal*, 13(2), 2009.

[57] Bromium. Bromium vsentry. defeat the unknown attack. defeat the unknown attack. defeat the unknown attack. *Whitepaper*, 2014.

[58] J. Butler, B. Arbaugh, and N. Petroni. R2: The exponential growth of rootkit techniques. In *BlackHat USA*, 2006.

[59] S. Chacon and J. C. Hamano. *Pro git*, volume 288. Springer, 2009.

[60] S. Checkoway and H. Shacham. *Iago attacks: why the system call API is a bad untrusted RPC interface*, volume 41. ACM, 2013.

[61] L. Chen. From tpm 1.2 to tpm 2.0. `http://www.iaik.tugraz.at/content/about_iaik/events/ETISS_INTRUST_2013/ETISS/slides/ETISS-INTRUST-2013-LiqunChen.pdf`, 2013.

[62] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 2–13. ACM, 2008.

[63] Citrix. Xenclient. extending the benefits of desktop virtualization to mobile laptop users. *Whitepaper*, 2010.

[64] G. Coker. Xen security modules (xsm). *Xen Summit*, pages 1–33, 2006.

[65] G. P. Consortium. Tee initial configuration test suite 1.1.0.1. `http://www.globalplatform.org/specificationsdevice.asp`.

[66] G. P. Consortium. Tee client api specification x1.0. `http://www.globalplatform.org/specificationsdevice.asp`, July 2010.

[67] G. P. Consortium. Tee internal api specification v1.0. `http://www.globalplatform.org/specificationsdevice.asp`, December 2011.

[68] G. P. Consortium. Tee system architecture v1.0. `http://www.globalplatform.org/specificationsdevice.asp`, December 2011.

[69] G. P. Consortium. Tee protection profile v1.0. `http://www.globalplatform.org/specificationsdevice.asp`, August 2013.

[70] G. P. Consortium. Tee secure element api specification v1.0. http://www.globalplatform.org/specificationsdevice.asp, August 2013.

[71] G. P. Consortium. Tee internal core api specification v1.1. `http://www.globalplatform.org/specificationsdevice.asp`, July 2014.

[72] G. P. Consortium. Tee ta debug specification v1.0. `http://www.globalplatform.org/specificationsdevice.asp`, February 2014.

[73] K. Cook. Yama lsm. 2010. *URL: http://lwn. net/Articles/393012/(v isited on 06/04/2012)*.

[74] J. Corbet. Creating linux virtual filesystems. `http://lwn.net/Articles/57369/`.

[75] J. Corbet. The integrity measurement architecture. `http://lwn.net/Articles/137306/`, May 2005.

[76] J. Corbet. Detecting kernel memory leaks. `Detectingkernelmemoryleaks`, June 2006.

[77] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux device drivers*. " O'Reilly Media, Inc.", 2005.

[78] M. Corporation. Secure startup – full volume encryption: Technical overview.

[79] J. Criswell, N. Dautenhahn, and V. Adve. Virtual ghost: protecting applications from hostile operating systems. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 81–96. ACM, 2014.

[80] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and R. W. Stewart. *The Zynq Book*. Strathclyde Academic Media, 2014.

[81] C. Dall and J. Nieh. Kvm/arm: The design and implementation of the linux arm hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 333–348, New York, NY, USA, 2014. ACM.

[82] L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security Symposium*, 2014.

[83] D. L. Davis. Secure boot, Aug. 10 1999. US Patent 5,937,063.

[84] R. Dayan, J. Freeman, W. Keown, and R. Springfield. Method and system for providing a trusted boot source in a partition, 2001.

[85] J. M. del Álamo, A. M. Fernández, R. Trapero, J. C. Yelmo, and M. A. Monjas. A privacy-considerate framework for identity management in mobile services. *Mobile Networks and Applications*, 16(4):446–459, 2011.

[86] M. F. Dennedy, J. Fox, and T. Finneran. *The Privacy Engineer's Manifesto*. Apress, 2014.

[87] M. F. Dennedy, J. Fox, and T. Finneran. *The Privacy Engineer's Manifesto*. Number p. 242 - 243. Apress, 2014.

[88] L. Desmet, W. Joosen, F. Massacci, P. Philippaerts, F. Piessens, I. Siahaan, and D. Vanoverberghe. Security-by-contract on the. net platform. *Information Security Technical Report*, 13(1):25–32, 2008.

[89] A. D.G, D. G.M, D. G.P, and S. J.V. Transaction security system. *IBM Systems Journal*, 30(2):206–229, 1991.

[90] C. DiBona, S. Ockman, and M. Stone. Appendix a: The tanenbaum-torvalds debate'. *Open Sources: Voices from the Open Source Revolution, Eunice, J 1998a, Beyond the Cathedral, Beyond the Bazaar, http://www. illuminata. com/public/content/cathedral/intro. ht m*, 1999.

[91] K. Dietrich and J. Winter. Secure boot revisited. In *Young Computer Scientists, ICYCS 2008.*, pages 2360–2365, Nov 2008.

[92] K. Dietrich and J. Winter. Implementation aspects of mobile and embedded trusted computing. In *Trusted Computing*, pages 29–44. Springer, 2009.

[93] I. Dinur, O. Dunkelman, and A. Shamir. New attacks on keccak-224 and keccak-256. In *Fast Software Encryption*, pages 442–461. Springer, 2012.

[94] I. Dinur, O. Dunkelman, and A. Shamir. Collision attacks on up to 5 rounds of sha-3 using generalized internal differentials. *FSE. LNCS, Springer (to appear, 2013)*, 2013.

[95] C. Doctorow. Lockdown, the coming war on general-purpose computing.

[96] M. Domeika. *Software development for embedded multi-core systems: a practical guide using embedded Intel architecture*. Newnes, 2011.

[97] N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan. Security-by-contract: Toward a semantics for digital signatures on mobile code. In *Public Key Infrastructure*, pages 297–312. Springer, 2007.

[98] J. Edge. Kernel key management. `http://lwn.net/Articles/210502/`, November 2006.

[99] J. Edge. The ceph filesystem. `http://lwn.net/Articles/258516/`, November 2007.

[100] J. Edge. Lsm stacking (again). `http://lwn.net/Articles/393008/`, June 2010.

[101] J. Edge. Trusted and encrypted keys. `http://lwn.net/Articles/408439/`, October 2010.

[102] J. Edge. Supporting multiple lsms. `http://lwn.net/Articles/426921/`, February 2011.

[103] J. Edge. Another lsm stacking approach. `http://lwn.net/Articles/518345/`, October 2012.

[104] J. Edge. Ima appraisal extension. `http://lwn.net/Articles/488906/`, March 2012.

[105] J.-E. Ekberg, K. Kostiainen, and N. Asokan. Trusted execution environments on mobile devices. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1497–1498. ACM, 2013.

[106] D. S. A. Elminaam, H. M. Abdual-Kader, and M. M. Hadhoud. Evaluating the performance of symmetric encryption algorithms. *IJ Network Security*, 10(3):216–222, 2010.

[107] I. . EMC. The digital universe of opportunities: Rich data and the increasing value of the internet of things. http://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm, April 2014.

[108] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information flow tracking system for real-time privacy monitoring on smartphones. *Communications of the ACM*, 57(3):99–106, 2014.

[109] Enisa. Roadmap for nis education programmes in europe. `https://www.enisa.europa.eu/activities/stakeholder-relations/nis-brokerage-1/roadmap-for-nis-education-programmes-in-europe`, October 2014.

[110] M. Escandell, L. McLane, and E. Reyes. Memory allocation with identification of requesting loadable kernel module, Apr. 17 2014. US Patent App. 14/023,290.

[111] V. ESXi. Bare metal hypervisor.

[112] Y. Falcone. You should better enforce than verify. In *Runtime Verification*, pages 89–105. Springer, 2010.

[113] Y. Falcone, S. Currea, and M. Jaber. Runtime verification and enforcement for android applications with rv-droid. In *Runtime Verification*, pages 88–95. Springer, 2013.

[114] D. F. Ferraiolo and D. R. Kuhn. Role-based access controls. *arXiv preprint arXiv:0903.2171*, 2009.

[115] N. Feske. Introducing genode. In *Talk at the Free and Open Source Software Developers' European Meeting. Slide available at http://genode. org*, 2012.

[116] W. E. Forum. Global risks 2014. *Ninth Edition*, 2014.

[117] J. Franklin, A. Seshadri, N. Qu, S. Chaki, and A. Datta. Attacking, repairing, and verifying secvisor: A retrospective on the security of a hypervisor. Technical report, Technical Report CMU-CyLab-08-008, Carnegie Mellon University, 2008.

[118] T. Frenzel, A. Lackorzynski, A. Warg, and H. Härtig. Arm trustzone as a virtualization technique in embedded systems. In *Proceedings of Twelfth Real-Time Linux Workshop, Nairobi, Kenya*, 2010.

[119] K. Fu, M. F. Kaashoek, and D. Mazieres. Fast and secure distributed read-only file system. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*, pages 13–13. USENIX Association, 2000.

[120] W. T. B. Futral. Trusted platform module family "2.0", 2013.

[121] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 193–206. ACM, 2003.

[122] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson. The digital distributed system security architecture. In *Proceedings of the 12th National Computer Security Conference*, pages 305–319, 1989.

[123] Genode. An exploration of arm trustzone technology. `http://genode.org/documentation/articles/trustzone`.

[124] S. Glass. Verified boot on chrome os and how to do it yourself. `http://static.googleusercontent.com/media/research.google.com/en//pubs/archive/42038.pdf`, October 2013.

[125] S. Glass. Verified u-boot. http://lwn.net/Articles/571031/, October 2013.

[126] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. Sirius: Securing remote untrusted storage. In *NDSS*, volume 3, pages 131–145, 2003.

[127] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *IEEE S&P*, 2014.

[128] O. Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*, volume 2. Cambridge university press, 2009.

[129] J. González. Generic support for arm trustzone. http://www.linuxplumbersconf.org/2014/ocw/system/presentations/1977/original/tz_plumbers.pdf, October 2014.

[130] J. González. Trusted platform module (tpm) on zynq, October 2014.

[131] J. González and P. Bonnet. Towards an open framework leveraging a trusted execution environment. In *Cyberspace Safety and Security*. Springer, 2013.

[132] J. González and P. Bonnet. Tee-based trusted storage. ISSN 1600–6100 ISBN 978-87-7949-310-0, IT-Universitetet i København, February 2014.

[133] J. González and P. Bonnet. Versatile endpoint storage security with trusted integrity modules. ISSN 1600–6100 ISBN 978-87-7949311-7, IT University of Copenhagen, February 2014.

[134] J. González, M. Hölzl, P. Riedl, P. Bonnet, and R. Mayrhofer. A practical hardware-assisted approach to customize trusted boot for mobile devices. In *Information Security*, pages 542–554. Springer, 2014.

[135] J. Goodacre. Technology preview: The armv8 architecture. white paper. Technical report, ARM, 2011.

[136] V. Gough. Encfs encrypted filesystem. *Located at: http://arg0. net/wiki/encfs*, 1(3):22, 2003.

[137] M. Graphics. Openamp framework user reference. Technical report, Mentor Graphics, 2014.

[138] G. Greenwald. *No Place to Hide: Edward Snowden, the NSA, and the US Surveillance State*. Metropolitan Books, 2014.

[139] M. A. Halcrow. ecryptfs: An enterprise-class encrypted filesystem for linux. In *Proceedings of the 2005 Linux Symposium*, volume 1, pages 201–218, 2005.

[140] T. Handa. Lsm/tomoyo: Lsm hooks for chmod/chown/chroot/mount/open/execve. http://lwn.net/Articles/351018/, September 2009.

[141] D. R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software: Practice and Experience*, 20(1):5–12, 1990.

[142] L. Harding. *The Snowden Files: The Inside Story of the World's Most Wanted Man*. Guardian Faber Publishing, 2014.

[143] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, page 11. ACM, 2013.

[144] S. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 1998.

[145] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of computer security*, 6(3):151–180, 1998.

[146] G. Hoglund and G. McGraw. *Exploiting software: how to break code*. Pearson Education India, 2004.

[147] M. Hölzl, R. Mayrhofer, and M. Roland. Requirements for an open ecosystem for embedded tamper resistant hardware on mobile devices. In *Proc. MoMM 2013: International Conference on Advances in Mobile Computing Multimedia*, pages 249–252, New York, USA, 2013. ACM.

[148] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 639–652. ACM, 2011.

[149] R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *USENIX Security Symposium*, pages 383–398, 2009.

[150] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space aslr. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 191–205. IEEE, 2013.

[151] M. I. Husain, L. Mandvekar, C. Qiao, and R. Sridhar. How to bypass verified boot security in chromium os. *arXiv preprint arXiv:1202.5282*, 2012.

[152] J.-Y. Hwang, S.-B. Suh, S.-K. Heo, C.-J. Park, J.-M. Ryu, S.-Y. Park, and C.-R. Kim. Xen on arm: System virtualization using xen hypervisor for arm-based secure mobile phones. In *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, pages 257–261. IEEE, 2008.

[153] S. Z. S. Idrus, S. A. Aljunid, S. M. Asi, S. Sudin, and R. B. Ahmad. Performance analysis of encryption algorithms text length size on web browsers,". *IJCSNS International Journal of Computer Science and Network Security*, 8(1):20–25, 2008.

[154] IEEE. Draft standard for information technology - portable operating system interface (posix)— part 1: System application program interface (api)— amendment: Protection, audit and control interfaces [c language], 1997.

[155] IEEE. Portable operating system interface (posix)-part 1: System application program interface (api): Protection, audit and control interfaces: C language, October 1997.

[156] P. Institute. 2014 cost of cyber crime study: United states. `http://www.ponemon.org/news-2/44`, October 2014.

[157] Intel. Intel® architecture instruction set extensions programming reference.

[158] Intel. Software guard extensions programming reference. https://software.intel.com/sites/default/files/329298-001.pdf.

[159] Intel. Intel® software guard extensions (intel® sgx) programming reference. https://software.intel.com/sites/default/files/329298-001.pdf, September 2013.

[160] Intel. Introduction to intel memory protection extensions. https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions, July 2013.

[161] Intel. Intel® 64 and ia-32 architectures. software developer's manual. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf, September 2014.

[162] Intel. Intel® software guard extensions (intel® sgx) programming reference rev.2. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, October 2014.

[163] Intel. Intel® trusted execution technology (intel® txt). http://www.intel.com/content/dam/www/public/us/en/documents/guides/intel-txt-software-development-guide.pdf, May 2014.

[164] Inter. *Inter 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1*, 2009.

[165] A. Iqbal. Security threats in integrated circuits security threats in integrated circuits. https://sdm.mit.edu/news/news_articles/iqbal-threats-solutions-ic-security-threats/iqbal-threats-solutions-ic-security-threats.html, October 2013.

[166] T. Jaeger. Reference monitor concept. Technical report, Systems and Internet Infrastructure Security Lab Pennsylvania State University.

[167] D. Jones. Checkpatch, a patch checking script. http://lwn.net/Articles/232240/, April 2007.

[168] M. Joye and J.-J. Quisquater. Hessian elliptic curves and side-channel attacks. In *Cryptographic Hardware and Embedded Systems—CHES 2001*, pages 402–410. Springer, 2001.

[169] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Fast*, volume 3, pages 29–42, 2003.

[170] S. H. Kan. *Metrics and models in software quality engineering*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[171] B. Kauer. Oslo: Improving the security of trusted computing. In *USENIX Security*, 2007.

[172] N. L. Kelem and R. J. Feiertag. A separation model for virtual machine monitors. In *Research in Security and Privacy, 1991. Proceedings., 1991 IEEE Computer Society Symposium on*, pages 78–86. IEEE, 1991.

[173] L. Kernel. Capabilities faq. https://www.kernel.org/pub/linux/libs/security/linux-privs/kernel-2.2/capfaq-0.2.txt.

[174] M. Kerrish. Namespaces in operation part 1 - 7. http://lwn.net/Articles/531114/, January 2013.

[175] G. H. Kim and E. H. Spafford. The design and implementation of tripwire: A file system integrity checker. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 18–29. ACM, 1994.

[176] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. sel4: formal verification of an operating-system kernel. *Communications of the ACM*, 53(6):107–115, 2010.

[177] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Advances in Cryptology—CRYPTO'99*, pages 388–397. Springer, 1999.

[178] P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology—CRYPTO'96*, pages 104–113. Springer, 1996.

[179] O. Kömmerling and M. G. Kuhn. Design principles for tamper-resistant smartcard processors. In *USENIX workshop on Smartcard Technology*, volume 12, pages 9–20, 1999.

[180] M. Kuhn. Smartcard tamper resistance. In *Encyclopedia of Cryptography and Security*, pages 1225–1227. Springer, 2011.

[181] K. Kursawe, D. Schellekens, and B. Preneel. Analyzing trusted platform communication. In *ECRYPT Workshop, CRASH-CRyptographic Advances in Secure Hardware*. Citeseer, 2005.

[182] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity.

[183] P. W. Lee. Combination nonvolatile integrated memory system using a universal technology most suitable for high-density, high-flexibility and high-security sim-card, smart-card and e-passport applications, Feb. 13 2007. US Patent 7,177,190.

[184] F. Leens. An introduction to i 2 c and spi protocols. *Instrumentation & Measurement Magazine, IEEE*, 12(1):8–13, 2009.

[185] K. Lewis, J. Kaufman, and N. Christakis. The taste for privacy: An analysis of college student privacy settings in an online social network. *Journal of Computer-Mediated Communication*, 14(1):79–100, 2008.

[186] J. Li, M. N. Krohn, D. Mazieres, and D. Shasha. Secure untrusted data repository (sundr). In *OSDI*, volume 4, pages 9–9, 2004.

[187] W. Li, M. Ma, J. Han, Y. Xia, B. Zang, C.-K. Chu, and T. Li. Building trusted path on untrusted device drivers for mobile devices. 2014.

[188] X. Li, H. Hu, G. Bai, Y. Jia, Z. Liang, and P. Saxena. Droidvault: A trusted data vault for android devices.

[189] S. Liebergeld and M. Lange. Android security, pitfalls and lessons learned. *Information Sciences and Systems*, 2013.

[190] J. Ligatti and S. Reddy. A theory of runtime enforcement, with results. In *Computer Security–ESORICS 2010*, pages 87–100. Springer, 2010.

[191] D. Liu and L. P. Cox. Veriui: attested login for mobile devices. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*, page 7. ACM, 2014.

[192] H. Löhr, A.-R. Sadeghi, C. Stüble, M. Weber, and M. Winandy. Modeling trusted computing support in a protection profile for high assurance security kernels. In *Trusted Computing*, pages 45–62. Springer, 2009.

[193] R. Love. *Linux kernel development*. Pearson Education, 2010.

[194] A. Madhavapeddy and D. J. Scott. Unikernels: the rise of the virtual library operating system. *Communications of the ACM*, 57(1):61–69, 2014.

[195] G. Madlmayr, J. Langer, C. Kantner, and J. Scharinger. *NFC Devices: Security and Privacy*, pages 642–647. 2008.

[196] S. E. Madnick and J. J. Donovan. Application and analysis of the virtual machine approach to information system security and isolation. In *Proceedings of the workshop on virtual computer systems*, pages 210–224. ACM, 1973.

[197] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*, pages 10–10. USENIX Association, 2000.

[198] A. B. McCabe, J. R. Tuttle, and C. W. Wood Jr. Tamper resistant smart card and method of protecting data in a smart card, Nov. 23 1999. US Patent 5,988,510.

[199] J. M. McCune, T. Jaeger, S. Berger, R. Caceres, and R. Sailer. Shamon: A system for distributed mandatory access control. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*, pages 23–32. IEEE, 2006.

[200] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. Trustvisor: Efficient tcb reduction and attestation. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 143–158. IEEE, 2010.

[201] K. N. McGill. Trusted mobile devices: Requirements for a mobile trusted platform module. *Johns Hopkins APL technical digest*, 32(2):544, 2013.

[202] G. McGraw. Software security. *Security & Privacy, IEEE*, 2(2):80–83, 2004.

[203] G. McGraw and E. W. Felten. *Securing Java: getting down to business with mobile code*. John Wiley & Sons, Inc., 1999.

[204] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. *HASP*, 13:10, 2013.

[205] R. C. Merkle. Protocols for public key cryptosystems. In *2012 IEEE Symposium on Security and Privacy*, pages 122–122. IEEE Computer Society, 1980.

[206] Mobile Phone Work Group. TCG mobile trusted module sepecification version 1 rev 7.02. Technical report, Apr. 2010.

[207] P. Mochel. The sysfs filesystem. In *Linux Symposium*, page 313, 2005.

[208] A. Molina-Markham, P. Shenoy, K. Fu, E. Cecchet, and D. Irwin. Private memoirs of a smart meter. In *Proceedings of the 2nd ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Building*, BuildSys '10, pages 61–66, New York, NY, USA, 2010. ACM.

[209] J. Morris. Kernel korner: the linux kernel cryptographic api. *Linux Journal*, 2003(108):10, 2003.

[210] S. Mukkamala, G. Janoski, and A. Sung. Intrusion detection using neural networks and support vector machines. *Neuronal Networks, IJCNN'02*, 2002.

[211] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. sel4: from general purpose to a proof of information flow enforcement. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 415–429. IEEE, 2013.

[212] A. Nadeem and M. Y. Javed. A performance comparison of data encryption algorithms. In *Information and communication technologies, 2005. ICICT 2005. First international conference on*, pages 84–89. IEEE, 2005.

[213] Nergal. The advanced return-into-lib (c) exploit. `http://phrack.org/issues/58/4.html`, November 2007.

[214] H. Nicol. Sieves of eratosthenes. *Nature*, 166:565–566, 1950.

[215] H. Nissenbaum. *Privacy in Context - Technology, Policy, and the Integrity of Social Life*. Stanford University Press, 2010.

[216] I. Novell. Apparmor application security for linux, 2008.

[217] M. Nystrom, M. Nicoles, and V. Zimmer. Uefi networking and pre-os security. *Intel Technology Journal*, (15):80–1, 2011.

[218] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in android. *Security and Communication Networks*, 5(6):658–673, 2012.

[219] J. C. (original documentation contributor). Intel(r) txt overview. `https://www.kernel.org/doc/Documentation/intel_txt.txt`, June 2009.

[220] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang. Sdf: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 471–484. ACM, 2014.

[221] J. Park and R. Sandhu. The uconabc usage control model. *ACM Trans. Inf. Syst. Secur.*, 7(1):128–174, Feb. 2004.

[222] B. Parno. Trust extension as a mechanism for secure code execution on commodity computers. 2010.

[223] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 233–247. IEEE, 2008.

[224] S. Pearson and B. Balacheff. *Trusted computing platforms: TCPA technology in context*. Prentice Hall Professional, 2003.

[225] M. Pennings. Getting top performance from nxp's getting top performance from nxp's lpc processors. `http://www.fampennings.nl/maarten/boeken/topperf.pdf`, November 2009.

[226] A. G. Pennington, J. L. Griffin, J. S. Bucy, J. D. Strunk, and G. R. Ganger. Storage-based intrusion detection. *ACM Trans. Inf. Syst. Secur.*, 13(4):30:1–30:27, Dec. 2010.

[227] T. Petazzoni. Your new arm soc linux support check-list! `http://www.elinux.org/images/a/ad/Arm-soc-checklist.pdf`, 2012.

[228] S. Pinto, D. Oliveira, J. Pereira, N. Cardoso, M. Ekpanyapong, J. Cabral, and A. Tavares. Towards a lightweight embedded virtualization architecture exploiting arm trustzone. In *19th IEEE International Conference on Emerging Technologies and Factory Automation*, 2014.

[229] M. Pirker and D. Slamanig. A framework for privacy-preserving mobile payment on security enhanced arm trustzone platforms. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on*, pages 1155–1160. IEEE, 2012.

[230] N. R. Potlapally, S. Ravi, A. Raghunathan, and N. K. Jha. A study of the energy consumption characteristics of cryptographic algorithms and security protocols. *Mobile Computing, IEEE Transactions on*, 5(2):128–143, 2006.

[231] P. Pucheral, L. Bouganim, P. Valduriez, and C. Bobineau. Picodbms: Scaling down database techniques for the smartcard. *The VLDB Journal*, 10(2-3):120–132, 2001.

[232] J. Quade. Kernel rootkit tricks. `http://www.linux-magazine.com/Online/Features/Kernel-Rootkit-Tricks`.

[233] M. Quaritsch and T. Winkler. Linux security modules enhancements: Module stacking framework and tcp state transition hooks for state-driven nids. *Secure Information and Communication*, 7, 2004.

[234] W. Rankl and W. Effing. *Smart Card Handbook*. June 2010.

[235] M. J. Ranum. Security: The root of the problem. *Queue*, 2(4):44, 2004.

[236] P. Riedl, P. Koller, R. Mayrhofer, A. Möller, M. Koelle, and M. Kranz. Visualizations and switching mechanisms for security zones. In *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*, page 278. ACM, 2013.

[237] J. Rouse. Mobile devices - the most hostile environment for security? *Network Security*, 2012(3):11–13, 3 2012.

[238] B. Rowe. Smart card with onboard authentication facility, 2002.

[239] J. M. Rushby. Proof of separability a verification technique for a class of security kernels. In *International Symposium on Programming*, pages 352–367. Springer, 1982.

[240] J. Rutkowska. On formally verified microkernels (and on attacking them), 2010.

[241] J. Rutkowska and R. Wojtczuk. Qubes os architecture. *Invisible Things Lab Tech Rep*, 2010.

[242] R. Sailer, T. Jaeger, E. Valdez, R. Caceres, R. Perez, S. Berger, J. L. Griffin, and L. van Doorn. Building a mac-based security architecture for the xen open-source hypervisor. In *Computer security applications conference, 21st Annual*, pages 10–pp. IEEE, 2005.

[243] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn. Design and implementation of a tcg-based integrity measurement architecture. In *USENIX Security Symposium*, volume 13, pages 223–238, 2004.

[244] T. Saito. Smart card for passport, electronic passport, and method, system, and apparatus for authenticating person holding smart card or electronic passport, Apr. 26 2004. US Patent App. 10/832,781.

[245] Samsung. Samsung ssd 840 evo data sheet, rev. 1.1. `http://www.samsung.com/global/business/semiconductor/minisite/SSD/downloads/document/Samsung_SSD_840_EVO_Data_Sheet_rev_1_1.pdf`, August 2013.

[246] R. Sandhu. Engineering authority and trust in cyberspace. In *The OM-AM and RBAC Way, The Proceedings of ACM Workshop on Role Based Access Control 2000*.

[247] R. Sandhu and X. Zhang. Peer-to-peer access control architecture using trusted computing technology. In *Proceedings of the tenth ACM symposium on Access control models and technologies*, pages 147–158. ACM, 2005.

[248] R. S. Sandhu and P. Samarati. Access control: principle and practice. *Communications Magazine, IEEE*, 32(9):40–48, 1994.

[249] D. Sangorrin, S. Honda, and H. Takada. Dual operating system architecture for real-time embedded systems. In *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), Brussels, Belgium*, pages 6–15. Citeseer, 2010.

[250] C. Schaufler. Smack in embedded computing. In *Proc. Ottawa Linux Symposium*, 2008.

[251] M. D. Schroeder. Engineering a security kernel for multics. In *ACM SIGOPS Operating Systems Review*, volume 9, pages 25–32. ACM, 1975.

[252] J. Schwartz. Secure boot device selection method and system, 2001.

[253] A. Segall. using the tpm: machine authentication and attestation. `http://opensecuritytraining.info/IntroToTrustedComputing_files/Day2-1-auth-and-att.pdf`.

[254] P. Semiconductors. The i2c-bus specification. *Philips Semiconductors*, 9397(750):00954, 2000.

[255] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. *ACM SIGOPS Operating Systems Review*, 41(6):335–350, 2007.

[256] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM, 2007.

[257] G. Shah, W. Drewry, R. Spangler, R. Tabone, S. Gwalani, and L. Semenzato. Firmware verified boot, Jan. 8 2015. US Patent 20,150,012,738.

[258] K. M. Shelfer and J. D. Procaccino. Smart card evolution. *Communications of the ACM*, 45(7):83–88, 2002.

[259] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, et al. Bitvisor: a thin hypervisor for enforcing i/o device security. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 121–130. ACM, 2009.

[260] I. Shklovski, S. D. Mainwaring, H. H. Skúladóttir, and H. Borgthorsson. Leakiness and creepiness in app space: perceptions of privacy and mobile app use. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*, pages 2347–2356. ACM, 2014.

[261] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating system concepts*, volume 8. 2013.

[262] S. Smalley and R. Craig. Security enhanced (se) android: Bringing flexible mac to android. In *NDSS*, 2013.

[263] S. Smalley, C. Vance, and W. Salamon. Implementing selinux as a linux security module. *NAI Labs Report*, 1:43, 2001.

[264] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 574–588. IEEE, 2013.

[265] W. Stallings. Cryptography and network security, principles and practices, 2003. *Practice Hall*.

[266] J. Stoll, C. S. Tashman, W. K. Edwards, and K. Spafford. Sesame: informing user security decisions with system visualization. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1045–1054. ACM, 2008.

[267] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 279–292. USENIX Association, 2006.

[268] N. N. Taleb. *Antifragile: things that gain from disorder*. Random House LLC, 2012.

[269] C. Tarnovsky. Deconstructing a 'secure'processor. *Black Hat DC*, 2010, 2010.

[270] P. TCG. client specific implementation specification for conventional bios, 2005.

[271] P. Team. Pax address space layout randomization (aslr), 2003.

[272] R. H. Thaler and C. R. Sunstein. *Nudge: Improving decisions about health, wealth, and happiness*. Yale University Press, 2008.

[273] A. Toninelli, R. Montanari, O. Lassila, and D. Khushraj. What's on users' minds? toward a usable smart phone security model. *Pervasive Computing, IEEE*, 8(2):32–39, 2009.

[274] Trusted Computing Group. TPM main specification version 1.2 rev. 116. Technical report, Mar. 2011.

[275] Trusted Computing Group. Tpm mobile with trusted execution environment for comprehensive mobile device security, June 2012.

[276] Trusted Computing Group. TPM 2.0 main specification family 2.0, level 00, revision 01.16. `http://www.trustedcomputinggroup.org/resources/tpm_library_specification`, October 2014.

[277] Unified EFI. UEFI specification version 2.2. Technical report, Nov. 2010.

[278] A. T. J. H. University. Tpm 2.0 re-envisioning of the tpm.

[279] P. Van der Linden. *Expert C programming: deep C secrets*. Prentice Hall Professional, 1994.

[280] J. Viega and G. McGraw. *Building secure software: how to avoid security problems the right way*. Pearson Education, 2001.

[281] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift. Aerie: flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems*, page 14. ACM, 2014.

[282] Z. Wang and X. Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 380–395. IEEE, 2010.

[283] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on*, pages 133–145. IEEE, 1999.

[284] M. Wei, M. Bjørling, P. Bonnet, and S. Swanson. I/o speculation for the microsecond era. In *Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference*, pages 475–482. USENIX Association, 2014.

[285] C. Weinhold and H. Härtig. Vpfs: Building a virtual private file system with a small trusted computing base. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 81–93. ACM, 2008.

[286] D. A. Wheeler. Linux kernel 2.6: It's worth more, 2004 (revised in 2011).

[287] D. A. Wheeler. Shellshock. `http://www.dwheeler.com/essays/shellshock.html`, 12 2014.

[288] D. Wilkins. Evolving hardware-based security: Firmware transition to tpm 2.0. `http://www.uefi.org/sites/default/files/resources/Phoenix_Summerfest_2013.pdf`, July 2013.

[289] R. Wilkins and B. Richardson. Uefisecure boot in modern computer security solutions. 2013.

[290] J. Winter. Trusted computing building blocks for embedded linux-based arm trustzone platforms. In *Proceedings of the 3rd ACM workshop on Scalable trusted computing*, pages 21–30. ACM, 2008.

[291] J. Winter. Experimenting with arm trustzone–or: How i met friendly piece of trusted hardware. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on*, pages 1161–1166. IEEE, 2012.

[292] J. Winter and K. Dietrich. A hijacker's guide to the lpc bus. In *Public Key Infrastructures, Services and Applications*, pages 176–193. Springer, 2012.

[293] J. Winter, P. Wiegele, M. Pirker, and R. Tögl. A flexible software development and emulation framework for arm trustzone. In *Trusted Systems*, pages 1–15. Springer, 2012.

[294] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman. Linux security module framework. In *Ottawa Linux Symposium*, volume 8032, 2002.

[295] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *Foundations of Intrusion Tolerant Systems*, pages 213–213. IEEE Computer Society, 2003.

[296] C. P. Wright, M. C. Martino, and E. Zadok. Ncryptfs: A secure and convenient cryptographic file system. In *USENIX Annual Technical Conference, General Track*, pages 197–210, 2003.

[297] G. Wurster and P. C. van Oorschot. A control point for reducing root abuse of file-system privileges. In *CCS*, New York, NY, USA, 2010. ACM.

[298] Xilinx. Programming arm trustzone architecture on the xilinx zynq-7000 all programmable soc. http://www.xilinx.com/support/documentation/user_guides/ug1019-zynq-trustzone.pdf.

[299] J. Zdziarski. Identifying back doors, attack points, and surveillance mechanisms in ios devices. *Digital Investigation*, 11(1):3–19, 2014.

[300] S. Zhao, Q. Zhang, G. Hu, Y. Qin, and D. Feng. Providing root of trust for arm trustzone using on-chip sram. In *Proceedings of the 4th International Workshop on Trustworthy Embedded Devices*, pages 25–36. ACM, 2014.

[301] V. Zimmer, M. Rothman, and S. Marisetty. *Beyond BIOS: Developing with the Unified Extensible Firmware Interface*. Intel Press, 2010.