Darkroom

The Use of ARM TrustZone for Secure Data Processing on the Cloud

Tiago Luís de Oliveira Brito

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Nuno Miguel Carvalho Santos

Examination Committee

Chairperson:	Prof. José Manuel da Costa Alves Marques
Supervisor:	Prof. Nuno Miguel Carvalho Santos
Member of the Committee:	Prof. Henrique João Lopes Domingos

October 2016

Acknowledgements

I would like to thank my supervising professor, Nuno Santos, for all the support and help during the last year, and for the opportunities he introduced me, which have improved my scientific knowledge and sparked an interest in proceeding with my academic activities for the following years.

I would like to thank my girlfriend Cátia for caring deeply and for always supporting my decisions. I would also like to thank my family and friends for supporting me during this five year adventure which now comes to an end, only to start a new one ever so challenging and satisfying.

Lisboa, October 2016
Tiago Luís de Oliveira Brito

For my future self,

Resumo

Hoje em dia verifica-se uma tendência entre serviços *web* de transferir capacidades de processamento e armazenamento para serviços na nuvem. Isto acontece porque dispositivos com grande capacidade de armazenamento e processamento são caros, enquanto que os serviços em nuvem proporcionam uma alternativa mais barata, um serviço contínuo e consistente. Uma grande limitação dos serviços em nuvem é o facto de serem altamente acessíveis através da *Internet* e, consequentemente, expostos a ataques. Ao explorar vulnerabilidades subtis presentes em aplicações, sistemas operativos ou *hypervisors* que formam a infraestrutura do provedor de serviço em nuvem, um atacante pode comprometer o sistema e comprometer os dados hospedados nessa plataforma. Se os dados sensíveis dos utilizadores forem comprometidos, o atacante pode fazer uso destes dados para fins de espionagem, chantagem, roubo de identidade e assédio. Uma solução para este problema seria processar estes dados sem os expor a componentes não confiáveis, como o sistema operativo. Esta dissertação descreve o desenho e implementação do *Darkroom*, um sistema para processamento seguro de dados na nuvem, que faz uso da tecnologia ARM TrustZone para evitar exposição de dados a componentes não confiáveis. Como demonstração desta abordagem, a versão actual do sistema foca-se especificamente no processamento de imagens, o qual pode prestar suporte a serviços web populares como o Instagram ou o Facebook. Através de uma avaliação experimental do sistema, observámos que esta solução apresenta apenas uma pequena sobrecarga do desempenho quando comparado com um sistema de processamento que não faz uso desta tecnologia.

Abstract

Nowadays, offloading storage and processing capacity to cloud servers is a growing trend among web services. This happens because high storage capacity and powerful processors are expensive, whilst cloud services provide a cheaper, ongoing, and reliable solution. The problem with cloud-based solutions is that servers are highly accessible through the Internet and therefore considerably exposed to attacks. By exploring subtle vulnerabilities present in user applications, management applications, operating systems and hypervisors, which comprise the cloud provider's infrastructure, an attacker may compromise the system, thus compromising the privacy sensitive user data hosted in it. This data can then be used by attackers for malicious purposes such as espionage, blackmail, identity theft and harassment. A solution to this problem is processing data without exposing it to untrusted components, such as an OS, which might be compromised by an attacker. This dissertation describes the design and implementation of Darkroom, a secure data processing service for the cloud leveraging ARM TrustZone technology. Our system enables users to securely process data in a secure environment that prevents the data from being exposed to untrusted software. As a use case for this secure processing approach, the current version of the system focuses on image processing, which can be used to support popular web services such as Instagram and Facebook. Through experimental evaluation of our system we observe that our solution adds a small overhead to data processing when compared to computer platforms that require the entire operating system to be trusted.

Palavras Chave
Keywords

Palavras Chave

Segurança

Privacidade

Computação Confiável

Computação em Nuvem

Processamento de Imagens

ARM TrustZone

Keywords

Security

Privacy

Trusted Computing

Cloud Computing

Image Processing

ARM TrustZone

Contents

List of Figures

List of Tables

Acronyms

API Application Programming Interface

ATAG ARM TAG

CPSR Current Program Status Register

CSU Central Security Unit

DMA Direct Memory Access

FIQ Fast Interrupt Request

FPU Floating Point Unit

GPS Global Positioning System

IPU Image Processing Unit

IRQ Interrupt Request

ISA Instruction Set Architecture

IVT Interrupt Vector Table

KLOC Thousands of Lines Of Code

LOC Lines Of Code

MAC Message Authentication Code

QSB Quick Start Board

SAHARA Symmetric/Asymmetric Hashing and Random Accelerator

SCR Secure Configuration Register

SGX Security Guard Extensions

SMC Secure Monitor Call

SSL Secure Sockets Layer

SVN Security Version Number

TCB Trusted Computing Base

TEE Trusted Execution Environment

TES Trusted Execution System

TLS Transport Layer Security

TPM Trusted Platform Module

TZVMM	TrustZone Virtual Machine Monitor
UART	Universal Asynchronous Receiver/Transmitter
VMM	Virtual Machine Monitor
VPU	Video Processing Unit

1 Introduction

Over the last few years, the cloud has become immensely popular due to the proliferation of numerous online services for storage, streaming, and processing of content. This happens because cloud services offer a more flexible environment for processing and storing personal data, with a higher processing and storage capacity than most devices used by everyday users. This is specially true today, where mobile devices are starting to dominate the digital media market over the Personal Computer (PC) [12]. However, such services frequently handle user data which has security and privacy requirements that are not always properly considered by the providers of these services. In some cases, this negligent behavior has even lead to serious scandals such as celebrity photo [6], and user document [35] leaks. To make matters worse, instead of trying to enforce security protocols capable of preventing this type of accidents, these providers end up stitching user contract terms so they can be absolved of these incidents, giving a bad reputation to cloud providers in general [11]. This dissertation explores the adoption of the ARM TrustZone technology, which has been traditionally applied for secure mobile processing, to build a secure-by-design cloud architecture.

1.1 Motivation

To secure user generated content in the cloud, such as personal documents, images, or videos, a commonly used approach has been to encrypt the content at the client side before it is sent to the cloud. While this approach is effective whenever the cloud is used for persistent storage, it can no longer be applied to scenarios where content needs to be processed by the cloud service. Notable examples include cloud services such as Facebook or Instagram which apply transformations to the images uploaded, for example to re-scale, rotate, or re-encode images. To perform such operations, the image data must be in its unencrypted format, point at which it may become vulnerable, e.g., to an attacker that managed to exploit some critical bug in the application code or in the operating system.

A promising alternative to encryption is to leverage Trusted Execution Environments (TEE) in order to safely perform image transformations at the cloud server without the need to rely on the rich operating system running on the server. Thus, if the OS is compromised, the TEE ensures that an attacker cannot access the memory regions allocated to the TEE where security-sensitive images are located. This approach has been recently adopted in many mobile device studies, whether to provide safe storage [28], enforce authentication mechanisms [31, 50], or provide OS introspection and monitoring capabilities [15, 49, 58]. One of the reasons this approach has been so popular in the mobile landscape has to do with ARM TrustZone [4], a technology that allows the implementation of TEE systems and is present in the majority of mobile device processors.

Alternatively, there have been attempts to implement trusted execution environments on the cloud. At first, such attempts consisted on combining an hypervisor, with special isolation properties, with a Trusted Platform Module (TPM) chip installed on cloud nodes [39, 41]. This approach allows the nodes to maintain the confidentiality of processing done inside guest virtual machines. Following the same idea of isolated processing, with the goal of reducing the trusted computing base and reinforcing the threat model, Intel has developed a technology called Security Guard Extensions (Intel SGX) [1]. Intel SGX has been designed to implement trusted computing on the cloud. Projects such as Haven [5] and VC3 [42] have used this technology to run unmodified legacy code and verified code, respectively, inside the trusted environment provided by Intel's SGX. But Intel SGX is still not highly available in commodity hardware and is still in development, unlike the more mature and highly available ARM TrustZone technology.

1.2 Goals

The goal of this dissertation is to study how ARM TrustZone can be applied in the context of secure cloud computing. The idea is to explore this technology to offer an environment for secure data processing similar to the approach of other technologies, such as Intel SGX. The same way ARM processors have been studied to increase the energetic efficiency of datacenters [43], this dissertation studies ARM TrustZone to support a secure environment for the cloud. As a use case for secure data processing on the cloud, we focus on providing a secure environment for processing sensitive user images. By leveraging a trusted execution environment, the system will also provide processing functions which do not rely or trust code of both the application and the operating system running on the cloud server. Decrypting and rendering the content will be performed by the software living in the isolated execution environment. This isolation property is enforced by the ARM TrustZone hardware, a trusted hardware technology widely available on commodity mobile devices. While providing a trusted data processing service, this system must fulfill the following requirements:

Small Trusted Computing Base
To ensure the security of this system and of the trusted services running on top of it, this implementation must have a small TCB, comprising of a few KLOC. This can be done by using TrustZone to isolate the secure service responsible for the trusted processing from the rich operating system and by designing a carefully crafted trusted kernel responsible for encrypting, decrypting and processing the sensitive data.

End-to-end Security
Applications using the secure data processing service must be assured of the authenticity, confidentiality and integrity of their data. For this reason, the system must allow client applications to send encrypted data to the service under the guarantee that this data will be stored securely such that only the secure environment can access the sensitive data. Additionally, the system must guarantee that the data is processed by a secure service instance without being exposed to untrusted components, such as the operating system and user applications.

Reduced Overhead
Our system must represent a small overhead when compared to data processing services which do not feature a similar secure processing environment. This way, the adoption of our system by cloud providers is not hindered by a high performance impact.

Demonstrate System Applicability to Secure Image Processing
In order to fulfill the goal of providing the use case of a secure image processing service, the system must support a wide range of common image transformations supported by popular image processing services such as social networks, and online image editors. Additionally, it must support the addition of new image transformation functions by trusted cloud administrators.

1.3 Contributions

In this dissertation we introduce Darkroom, a secure data processing service for the cloud which leverages the ARM TrustZone technology to provide an isolated environment for processing data securely without exposing it to untrusted components, such as a rich operating system and user applications. In order to demonstrate the applicability of Darkroom for secure data processing, this dissertation focuses on the use case of secure image processing on the cloud. The use of ARM TrustZone allows Darkroom to keep a reduced trusted computing base and to isolate the processing logic from untrusted software components. Additionally, by employing cryptographic protocols inside the secure environment, our system achieves end-to-end security which allows the data to be deployed and processed securely. The contributions of this report are as follows:

- The design of Darkroom, one of the first systems leveraging ARM TrustZone for the cloud;
- Implementation of a system prototype on real TrustZone-enabled hardware;
- Implementation of a mobile application, which acts as a client for sending images, requesting transformations and receiving images to and from the prototype;
- Experimental evaluation of the prototype showing a reduced overhead (average 16% for the tested images), with a small TCB (approximately 25.5 KLOC), and easy to use by client applications.

During this research project, Darkroom was presented at a Microsoft Research event called PhD Summer School 2016. This event took place at Cambridge, where Microsoft researchers are actively working on trusted computing for the cloud by leveraging Intel Security Guard Extensions (SGX). In addition a paper describing Darkroom was accepted at the Workshop on Mobility and Cloud Security & Privacy (WMCSP 2016) where it was presented in Budapest, Hungary, during the 35th Symposium on Reliable Distributed Systems (SRDS 2016).

1.4 Research History

During the development of Darkroom we had to overcome several design and implementation challenges. These challenges were responsible for the natural shifts which occur within research projects such as this one. This means that these challenges made us commit to choices which guide the project towards slightly different goals than those we start off with. As an example of this, the initial goal for this project was to develop a small secure runtime from scratch, in order to achieve the aforementioned goal of maintaining a reduced trusted computing base. But building a small runtime which, at the same time, must be flexible enough to support the additional features and requirements of the system is complex and time consuming. This implementation challenge forced us to make a decision and search for an open source small runtime we could adapt to fit our requirements.

Another example was the implementation of cryptographic features to support our confidentiality, integrity and authenticity requirements. The real TrustZone-enabled hardware on which our prototype was built has a proprietary cryptographic coprocessor which implements the algorithms we need in hardware, thus providing a way to maintain a small trusted computing base and have better performance for cryptographic operations. But access to the documentation for this hardware component was denied to us, meaning we had to implement the necessary features in software. More details regarding the several implementation attempts and challenges are described in Section 5.7.

1.5 Structure of the Document

The rest of this document is organized as follows. Chapter 2 describes the related work regarding image processing on the cloud and the use of trusted hardware technology such as ARM TrustZone and Intel SGX. Chapter 3 describes Darkroom's architecture. Chapter 4 describes the technical details of our system, the implementation challenges and algorithms used by our Darkroom prototype to securely process images on the cloud. Chapter 5 presents the results of the experimental evaluation study. Finally, Chapter 6 concludes this document by summarizing our findings and future work.

2 Related Work

This chapter describes the related work and is organized in three main parts. First, in order to better motivate our proposal, we briefly discuss the existing methods to process image data on the cloud and the underlying security issues associated to them. In the second part of this section, we overview the state of the art of general-purpose cloud security mechanisms used to implement sensitive data handling server applications, such as those described in the first part. We also explain why such mechanisms fall short in supporting secure data computation when the underlying operating system becomes compromised. Finally, the third part describes a specific class of security systems which, similarly to *Darkroom*, leverage Trusted Execution Environments (TEE) to implement secure computing applications. In this third part we will also focus on hardware-based TEE such as ARM TrustZone and Intel SGX, and explain the advantages and disadvantages of their application on the cloud.

2.1 Image Processing on the Cloud

This section describes techniques employed by cloud services to process image data on the cloud. Additionally, this section also describes security issues which arise from computing sensitive image data on cloud servers and what precautions a cloud provider should take when handling user sensitive image data. Alongside this discussion, we will also present examples of popular commercial cloud-based systems which process user image data and could enjoy the added security offered by Darkroom.

Popular web services such as social networks and content management systems (Facebook [1], Instagram [2], Lychee [3]) process photos uploaded by their users for several purposes. The most common image processing done by these services are image transformations such as resizing an image, applying color filters, thumbnail generation for the profile picture, etc. These are fairly basic image transformations but still require image data to be available in its raw (*plaintext*) form for processing. This means that, even using typical end-to-end encryption techniques to protect data during transport, this data must be decrypted for the services to process it. By exposing sensitive user data to an untrusted system, data might be compromised when an attack is performed by a malicious agent exploring vulnerabilities in the operating system, hypervisor or service application.

The ideal way to process sensitive user images would be to process these images in its encrypted state. This would allow images to be transformed but still maintain the confidentiality properties offered by cryptographic schemes, thus rendering this content useless for a successful attacker. But although

[1] `https://www.facebook.com`
[2] `https://www.instagram.com/`
[3] `https://lychee.electerious.com/`

fully and partial homomorphic encryption schemes have been introduced and explored by the research community, allowing computation to be performed over encrypted data, these schemes suffer from huge performance overheads and limits the processing to be done over the target data [16, 17].

In order to understand how sensitive image data can be used nefariously we can look at how the research community has been exploring the idea of using large image datasets to train machine learning models for face recognition, to recognize landmarks, extrapolate the location of a photo, etc. DeepFace [54] is a face recognition system developed by Facebook to close the gap between the most popular benchmark in face recognition and human level accuracy. Auto-tagging [48] is another face recognition system which improves on previous face recognition systems by using social context from the social network Facebook to decide between potential matches in a photo. In order to train the neural networks which compose these face recognition systems, large datasets of images have to be processed and annotated for the system. *Tour the world* [60] is another data-mining system by Google which uses an image dataset of over 20 million GPS-tagged photos to organize, model and recognize landmarks all over the world.

Although the systems described above are not image transformation systems, but rather data-mining engines for extracting information from image data, they do process huge datasets of image data. And from these image datasets, these systems can perceive information which may be sensitive for image owners. It is clear that by using a face recognition system, an attacker can identify individuals in photographs, use a landmark recognition engine to identify the location where a photo was taken and eventually build a time-line of places visited by the targeted individual. Additionally, most of these data-mining systems require high computational power and storage capabilities to support the models and datasets needed for processing. This is why many researchers make use of powerful cloud computing services such as Amazon's AWS [4] and Microsoft's Azure [5]. This means that in a system which requires large image datasets these images are also hosted on the cloud computing service. These remarks work as reinforcement for the need to protect sensitive image content of users in cloud-hosted services. If a cloud-hosted service becomes compromised and high amounts of sensitive image data is disclosed, it is possible to process this data and gather private information about the original owners of the data.

2.2 Overview of Cloud Security Mechanisms

When the topic of cloud security is discussed it is common to think about how to secure the cloud provider's infrastructure from a malicious user trying to take control of resources in the datacenter. And for the cloud provider this is generally the most important security concern to be mitigated. But improvements on cloud security have reached a point where this is not a major concern. This is specially true after the extensive use of virtualization techniques to effectively isolate the code controlled by the user from the code of the cloud provider's software infrastructure. This new software layer - virtualization - is usually represented by the hypervisor, which itself must be properly configured, managed and secured. But it is clear that the potential to compromise this virtualization layer, or hypervisor, even though

[4]https://aws.amazon.com
[5]https://azure.microsoft.com/

largely theoretical, still exists [57]. And badly configured hypervisors might introduce critical vulnerabilities which may lead to attacks on the system and eventually to the cloud provider's infrastructure. But since the focus of this dissertation is on the protection of user sensitive data from an untrusted cloud provider software infrastructure, we will not discuss this typical cloud security approach any further.

Regarding untrusted cloud-hosted services, work has been developed on several fronts. One of these fronts has been securing stored data in untrusted cloud-hosted storage, and the other has been the use of full or partial homomorphic encryption techniques to achieve confidentiality of data and code computed on untrusted cloud services. For untrusted cloud-hosted storage services it is generally sufficient for systems such as Venus [44] or DEPSKY [7] to protect data confidentiality by relying exclusively on cryptographic techniques performed at the client side. This approach tends to be attractive for users because it precludes the need to trust in specific components controlled by the cloud provider. The downside of this approach, however, is that encrypted data cannot be processed by applications, e.g., for performing image transformations, which reduces the applicability of this technique on the cloud.

To overcome this limitation, researchers have studied ways to allow for encrypted data processing by leveraging advanced cryptographic techniques. Although fully homomorphic encryption schemes allow arbitrary computation to be processed on encrypted data, these schemes suffer from high overhead [16, 17]. On the other hand, partially homomorphic encryption has been successfully employed in several domains such as database queries [3] and MapReduce programs [55]. CryptDB [36] is a representative example of such a system which employs partially homomorphic encryption to enable SQL query processing over encrypted relational databases. Nevertheless, systems such as CryptDB tend to limit the functions that can be executed, introduce considerable performance overheads, and depend on weaker cryptographic schemes when compared with traditional ones.

Other cloud-based systems enjoy the idea of a hypervisor to isolate untrusted code from security critical code, which it aims to protect. But, unlike typical hypervisors used by cloud providers, these systems use trusted hypervisors to protect the code running inside the hypervisor from the operating system, and not the opposite. MiniBox [29] employs a trusted hypervisor to protect small parts of application logic from the host OS and also achieves protection of the host OS from the code running inside the hypervisor. Another project which relies on a trusted hypervisor is PrivateCore vCage [37]. This system executes guest virtual machines entirely in-cache and encrypts the data before transferring it to main memory. The advantage of this system is its resistance to memory probes and physical attacks. There are other systems [53, 59] which rely on a hypervisor to defend applications from a malicious operating system but were not designed for the cloud.

2.3 Secure Processing based on Intel SGX

An alternative approach for secure cloud processing is to leverage Intel's upcoming Security Guard Extensions (SGX). Intel SGX is a set of hardware extensions to the Intel architecture which enables processes to maintain secure address space regions. These regions are called *enclaves* and provide a Trusted Execution Environment (TEE) for running security-sensitive application code in isolation from the OS. Although this approach requires users to trust the SGX hardware deployed by the cloud provider, enclaves can run arbitrary functions at native processor speed, hence faster than homomorphic en-

Permission	Type	Instruction	Description	Version
Privileged	MEM	EADD	Add a page	r1
Privileged	MEM	EBLOCK	Block an EPC page	r1
Privileged	EXE	ECREATE	Create an enclave	r1
Privileged	DBG	EDBGRD	Read data by debugger	r1
Privileged	DBG	EDBGWR	Write data by debugger	r1
Privileged	MEM	EEXTEND	Extend EPC page measurement	r1
Privileged	EXE	EINIT	Initialize an enclave	r1
Privileged	MEM	ELDB	Load an EPC page as blocked	r1
Privileged	MEM	ELDU	Load an EPC page as unblocked	r1
Privileged	SEC	EPA	Add a version array	r1
Privileged	MEM	EREMOVE	Remove a page from EPC	r1
Privileged	MEM	ETRACK	Activate EBLOCK checks	r1
Privileged	MEM	EWB	Write back/invalidate an EPC page	r1
Privileged	MEM	EAUG	Allocate a page to an existing enclave	r2
Privileged	SEC	EMODPR	Restrict page permissions	r2
Privileged	EXE	EMODT	Change the type of an EPC page	r2
User-level	EXE	EENTER	Enter an enclave	r1
User-level	EXE	EEXIT	Exit an enclave	r1
User-level	SEC	EGETKEY	Create a cryptographic key	r1
User-level	SEC	EREPORT	Create a cryptographic report	r1
User-level	EXE	ERESUME	Re-enter an enclave	r1
User-level	MEM	EACCEPT	Accept changes to a page	r2
User-level	SEC	EMODPE	Enhance access rights	r2
User-level	MEM	EACCEPTCOPY	Copy a page to a new location	r2

Table 2.1: Intel SGX instructions (adapted from OpenSGX). MEM: Memory management related; EXE: Enclave execution related; SEC: Security or permissions related.

cryption schemes, and provide strong security properties. In particular, enclaves' internal state cannot be accessed neither by privileged system code nor through memory probe attacks. Projects such as Haven [5] and VC3 [42] have started to explore ways to run unmodified legacy code and verified code, respectively, inside enclaves. However, some fundamental limitations need to be overcome in order to make this technology fully practical and robust [5].

Intel SGX allows more than one *enclave* for each processor. This is achievable through the use of a new x86 instruction set architecture (ISA) extension. This new ISA enables the creation, expansion and removal of enclaves which contain the critical code and data of a sensitive application. Additionally, the ISA extension supports remote attestation and sealing that allows remote verification and securely saving enclave data in non-volatile memory, thus achieving the goal of confidentiality and integrity of critical data inside the enclave. Table 2.1 describes the new instructions added to the x86 instruction set architecture which enable Intel SGX's features. The instructions thereby presented are described in detail in Revision 1 [20] and Revision 2 [21]. Table 2.2 describes the support structures which complement the new ISA.

The remainder of this section describes the software lifecycle of a SGX-aware application, the features supported by Intel SGX and recent SGX based projects developed by the research community. We will focus on three Intel SGX based projects because they represent a wide range of projects achievable using Intel SGX technology. The first project discussed will be Haven, the second project is called VC3 and the third is called OpenSGX [22].

Structure		Description
EPCM	Enclave Page Cache Map	Meta-data of an EPC page
SECS	Enclave Control Structure	Meta-data of an enclave
TCS	Thread Control Structure	Meta-data of a single thread
SSA	State Save Area	Used to save processor state
PageInfo	Page Information	Used for EPC-management
SECINFO	Security Information	Meta-data of an enclave page
PCMD	Paging Crypto MetaData	Used to track a page-out page
SIGSTRUCT	Enclave Signature Structure	Enclave certificate
EINITTOKEN	EINIT Token Structure	Used to validate the enclave
REPORT	Report Structure	Return structure of EREPORT
TARGETINFO	Report Target Info	Parameter of EREPORT
KEYREQUEST	Key Request	Parameter of EGETKEY
VA	Version Array	Version for evicted EPC pages

Table 2.2: Intel SGX Data Structure (adapted from OpenSGX).

2.3.1 Threat Model and Assumptions

The goal of Intel SGX is to protect the confidentiality and integrity of user data in an untrusted cloud provider. In addition to assuming an untrusted cloud provider, Intel SGX also considers external adversaries which may have access to data such as network packets. Considering this, Intel SGX assumes a powerful adversary who may control all software components in a cloud provider's infrastructure, including the operating system, hypervisor and hardware, except the SGX-enabled processor. This means the adversary may have direct access to the cloud infrastructure, which accounts for the jobs running on the cloud, the users and nodes in a datacenter. A typical example of this type of adversary is a cloud provider's administrator which, by having access to the machines inside the datacenter, can login to a machine and explore kernel vulnerabilities, read user data in memory, in the network and on the disk.

The Intel SGX specification describes sufficient details to understand how the underlying mechanisms of Intel SGX work, but under-specifies other necessary software components, such as operating system support, debugging and software development toolchains. As pointed out in the article introducing OpenSGX, many SGX instructions (Table 2.1) require kernel privilege, but system call interface and operating system support is not specified in detail in Intel SGX's specification. The system call interface is critical for SGX applications because they must rely on necessary support from an operating system that is not trusted. This is why projects described later in this subsection define and implement an interface to provide support for SGX applications, thus filling this gap.

This is a critical gap for secure applications since a badly defined interface can expose several attack vectors such as Iago attacks [10]. These are attacks where a malicious kernel may lead a trusted process astray by falsifying its inputs through the system call interface. With the use of a well defined and implemented interface, Iago attacks can be thwarted because the new API can perform validations, for example by integrating EAUG/EACCEPT instructions into the dynamic memory allocation function, which a malicious operating system cannot bypass. On the other hand, a malicious operating system can still launch denial-of-service attacks on SGX [34], which are much harder to mitigate. For this reason, denial-of-service attacks are generally not considered when developing SGX applications.

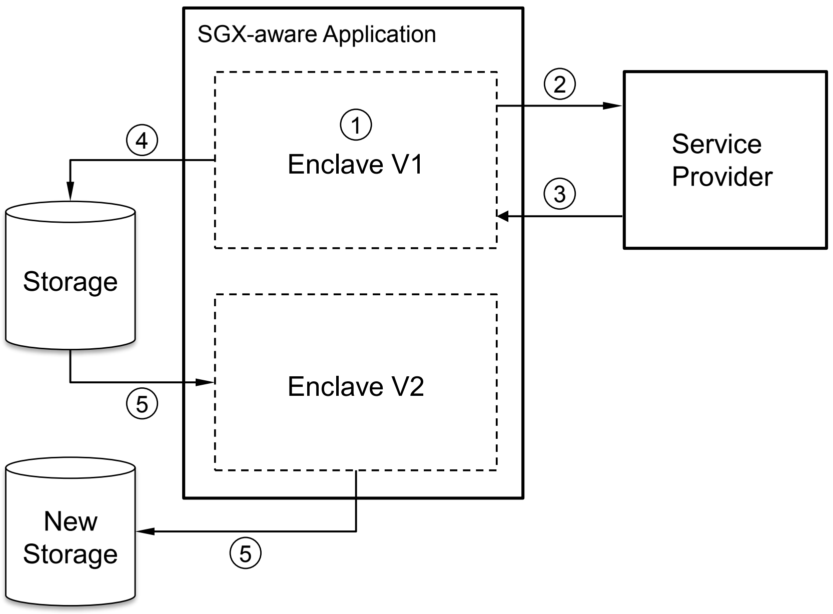

Figure 2.1: Software lifecycle of a SGX-aware application.

2.3.2 Software Lifecycle

The main advantage of Intel SGX regarding ARM TrustZone is that Intel SGX includes hardware based attestation and sealing, whereas for ARM TrustZone projects these features must be supported by custom software. This increases the trusted computing base and has a higher impact on performance. Since attestation and sealing represent fundamental features of Intel SGX, we follow with the description of the software lifecycle of SGX applications where both remote attestation and sealing are included [1]. In order to visualize the software lifecycle of a SGX application, Figure 2.1 illustrates the lifecycle described below. Note that code inside the enclave must never be accessible to untrusted software. For that reason, after enclave creation, the untrusted application must download the protected code into the enclave directly. The protected code transferred to the enclave will be called service provider's software.

1. Enclave Creation - The untrusted application launches the enclave environment to protect the service provider's software. While the enclave is built, a secure log is recorded reflecting the contents of the enclave and how it was loaded. This secure log is the enclave's *Measurement* and is described further bellow.

2. Attestation - The enclave contacts the service provider to have its sensitive data provisioned to the enclave. The platform produces a secure assertion that identifies the hardware environment and the enclave.

3. Provisioning - The service provider assesses the trustworthiness of the enclave. It uses the attestation to establish secure communication and send the protected data to the enclave. Using the secure channel, the service provider sends the data to the enclave.

4. Sealing/Unsealing - The enclave uses a persistent hardware-based encryption key to securely encrypt and store its sensitive data in a way that ensures the data can be retrieved only when the

trusted environment is restored.

5. Software Upgrade - To migrate data from an older software version to the newer version, the software can request seal keys from older versions to unseal the data and request the new version's seal. This makes the sealed data unavailable to previous versions of the software.

2.3.3 Measurement

Measurement is the first SGX-specific operation performed during the lifecycle of SGX-aware applications. Its purpose is to establish identities for the following operations of attestation and sealing. Each enclave has two identities implemented as measurement registers, *MRENCLAVE* and *MRSIGNER*. These values are recorded upon enclave creation and the registers become available to write before enclave execution. Only protected code inside the enclave can write to these registers.

MRENCLAVE can be described as the *enclave identity* and is a SHA-256 digest of a log recording all activity done while the enclave is built. This log consists of the contents of the memory pages inside the enclave (code, data, stack and heap), the relative position of the pages in the enclave and all security flags associated to these pages.

MRSIGNER can be described as the *sealing identity* and is the identity responsible for data protection. It consists of a *sealing authority*, a product ID and a version number. The sealing authority is an entity responsible for signing the enclave before distribution. This sealing authority is used upon the validation steps performed during the bootstrap process of the enclave. These validation steps are described bellow:

1. The enclave builder flashes the hardware with a RSA signed enclave certificate, called *SIGSTRUCT*, which contains the public key of the Sealing Authority and the expected value of the enclave identity, we will call it *SIGMRENCLAVE*.
2. While the enclave is built, a log with all that activity is recorded and later a SHA-256 digest of this log is recorded to the *MRENCLAVE* register. This is the process described earlier.
3. The hardware checks the signature on the certificate (*SIGMRENCLAVE*) against the measured *MRENCLAVE* using the public key of the Sealing Authority.
4. If this check passes, a hash of the public key of the Sealing Authority is stored in the *MRSIGNER* register, thus concluding the measurement process.

It is important to note that enclaves signed with the same Sealing Authority have the same *MRSIGNER* value and can use this sealing identity to share and migrate sealed data between them. This can be useful to migrate data between enclaves of the same Sealing Authority but different versions.

2.3.4 Attestation

Attestation is the process through which it is possible to demonstrate that a piece of software has been properly instantiated on a certain platform. Attestation is important for trusted computing because it allows a party to acknowledge the trustworthiness of a software/platform and thus achieving confidence on that software/platform. For Intel SGX, attestation enables one party to gain confidence that the correct software is being executed inside an enclave on a trusted platform. Attestation can be used in two ways with Intel SGX. On one hand, local attestation is the mechanism through which Intel SGX creates an authenticated assertion between two enclaves running on the same platform. On the other hand, a mechanism called remote attestation can be used to extend the feature of local attestation to third parties running outside the platform.

For intra-platform attestation between different, yet cooperative, enclaves, Intel SGX supports the *EREPORT* instruction. This instruction enables the creation of the *REPORT* structure. Included in the *REPORT* structure are the two identities of the enclave, enclave attributes, information for the target enclave defined by the developer, the trustworthiness of the hardware TCB and a message authentication code (MAC). This MAC is computed using the AES128-CMAC algorithm with a special key called *Report Key*, known only to the target enclave and *EREPORT* instruction. The target enclave can access the report key via the *EGETKEY* instruction and re-compute the MAC to verify that the *REPORT* structure was indeed produced by the attesting enclave.

For intra-platform attestation it is sufficient to use a symmetric key system, because enclaves on the same platform share the SGX enabled processor, which mitigates the key sharing problems inherent to symmetric key cryptography. But for inter-platform attestation these key sharing problems persist and the solution must shift from using symmetric key cryptography to asymmetric cryptography. To support remote attestation, Intel SGX presents a special enclave called the Quoting Enclave. The Quoting Enclave uses the intra-platform attestation method, described above, to verify the *REPORT* structure of enclaves on the same platform. This special enclave then replaces the MAC on these *REPORT* structures with a signature created using a private asymmetric key specific for each device. These new, modified *REPORT* structures form a new structure called *QUOTE*.

This remote attestation process using the Quoting Enclave is frequently used for provisioning a symmetric communication key between two enclaves at enrolment time. This newly provisioned key can then be used for subsequent communications between both enclaves. A paper by Intel which describes the SGX technology describes a remote attestation process example for this purpose [1]. In this example, a challenger issues a challenge to the application in order to demonstrate that it is executing inside an enclave. The enclave generates a manifest including a response to the challenge and an ephemerally generated public key to be used by the challenger for communicating secrets back to the enclave. Using the *EREPORT* instruction, the enclave binds the manifest to the enclave as described before. This *REPORT* is then forwarded to the Quoting Enclave which signs the *REPORT*, thus creating the *QUOTE* structure. The *QUOTE* is then sent to the challenger which can use the public key certificate of the Quoting Enclave to validate the *QUOTE* and check the manifest for the response to the original challenge.

A similar process can be implemented via software for ARM TrustZone based projects but, as mentioned before, Intel SGX has the advantage of supporting this feature out of the box and implemented in hardware. This not only allows for a smaller software TCB but also eases the development of trusted applications using the Intel SGX platform. And because attestation is a feature interesting for all trusted applications it should be implemented by the platform developers and not by application developers.

2.3.5 Sealing

Intel SGX instructions and memory protection measures protect data inside an enclave. But applications often require the need to store data in non-volatile memory for use in subsequent executions. For this reason Intel SGX supports a sealing mechanism to securely store enclave data outside the enclave whilst still maintaining the confidentiality and integrity properties of critical data. Intel SGX supports two sealing policies for enclaves. The two sealing policies are sealing to the Enclave Identity and sealing to the Sealing Identity.

Sealing to the Enclave Identity is to seal considering the value of *MRENCLAVE*. This means that the *EGETKEY* instruction will yield a different key for each enclave. This sealing policy provides full isolation between enclaves, even those which differ only by version number. The disadvantage of using this policy is that it provides full isolation between enclaves, thus preventing offline data migration between enclaves with different versions.

Sealing to the Sealing Identity is to seal considering the value of *MRSIGNER*. This means that the *EGETKEY* instruction will yield a different key for each Sealing Authority that signed the enclave's certificate. In contrast to the sealing policy described before, sealing to the sealing identity allows offline migration of sealed data between enclaves with different versions, since these enclaves share the same *MRSIGNER* value. But this feature introduces a new problem for the security properties they wish to maintain for data inside enclaves. If enclaves with different versions can share sealed data, then older enclaves can access data of newer enclaves. This is not desirable if the reason why an enclave was updated for a new version was to fix security bugs. For this reason, Intel SGX added a Security Version Number (SVN) as part of the sealing identity. The SVN can be specified when the *EGETKEY* instruction produces the Seal Key in order to determine the minimum SVN value with access to that enclave's data.

2.3.6 Intel SGX Projects

For the remainder of this section, we will focus on three Intel SGX projects as we feel they represent a wide range of possible SGX applications. For each project we will describe the main features and novelty as well as the challenges solved by the authors. Additionally we will discuss how some of these projects might be useful for our system in any way.

The first project is called Haven [5] and has the goal of achieving shielded (isolated) execution of unmodified legacy applications, such as SQL Server and Apache, on commodity OS and hardware by leveraging Intel SGX to defend against privileged code and physical attacks done from/by a malicious host. The second project is called VC3 [42] and aims at adapting Hadoop to run a MapReduce computation scheme inside a protected environment without having to trust on Hadoop, the operating system and the hypervisor. Lastly, the third project is called OpenSGX [22] and is a fully functional, instruction-compatible emulator of Intel SGX as well as a development platform for Intel SGX-aware applications.

Haven was one of the first projects to explore Intel SGX for shielded execution of unmodified legacy applications. Shielded execution is a concept introduced by the authors of Haven to define an execution to contrast with previous protection mechanisms employed by cloud providers. Usually, cloud providers employ mechanisms such as process isolation, sandboxing and managed code in order to confine untrusted programs in a specific region and protect the rest of the system from malicious actions that might be taken by it. Isolated execution is the inverse of this process. In isolated execution, the goal is to protect specific code from the rest of the system, even if this system is privileged, such as an operating system or hypervisor. Shielded execution builds on top of this isolation mechanism by using Intel SGX x86 instruction set architecture extension to support the isolated execution features, memory protection, attestation and sealing. Additionally, because isolation is not sufficient, a custom operating system API, LibOS, was designed and implemented to prevent threats such as Iago attacks [10], where a malicious operating system attempts to subvert an isolated application by exploiting its assumption of correct OS behaviour, as explained previously. An additional challenge solved by the authors was running unmodified binaries. This is because Intel SGX was developed to isolate and protect small portions of program logic and not full applications, which load code and data at runtime, use dynamic memory allocation, use thread-local storage, raise and handle exceptions, etc. Haven addresses this challenge by relying on enhancements to the SGX specification suggested by the authors, emulating unsupported instructions, validating and handling exceptions from within an enclave and by adapting LibOS behaviour accordingly.

VC3 is the first system to execute MapReduce jobs while achieving confidentiality and integrity of the code and data used for the jobs. Additionally, VC3 guarantees the correctness and completeness of the results of the MapReduce scheme computation. By carefully partitioning VC3 the authors keep Hadoop, the operating system and the hypervisor out of the TCB. This is achieved by having only the strictly critical code running inside the enclave without a strict dependence on other components of the system such as Hadoop or the OS. The authors also designed and implemented two new security protocols for MapReduce jobs using this system. The first protocol considers cloud attestation and key exchange and the second protocol is for gathering evidence of the correct execution of the job. The result is a verifiable and confidential cloud computing (VC3) tool for running a MapReduce computing scheme with negligible overhead for its security guarantees.

OpenSGX is a platform built for addressing the issue of designing and implementing SGX-aware applications. It includes an open-source emulator of Intel SGX, an enclave program loader/packager, an OpenSGX user library for addressing Iago attacks and malicious OS intervention, a debugging tool and a performance monitor. The goal of the authors is to spark the development of applications using Intel SGX features for commercial and research communities. The authors believe that Intel SGX has potential for becoming a new paradigm in trusted cloud computing but acknowledge the lack of a usable platform for the research community to work with. In fact, the authors believe that Intel SGX is still limited to a select group of people who have been actively exploring the technology [5, 42, 46]. As use cases to evaluate OpenSGX, the authors developed two applications. The first is an adaptation of the server-side Tor software to work with Intel SGX, but shielding the programming logic of directory servers. The second application is a secure IO path mechanism for secure communication between the CPU/memory and devices.

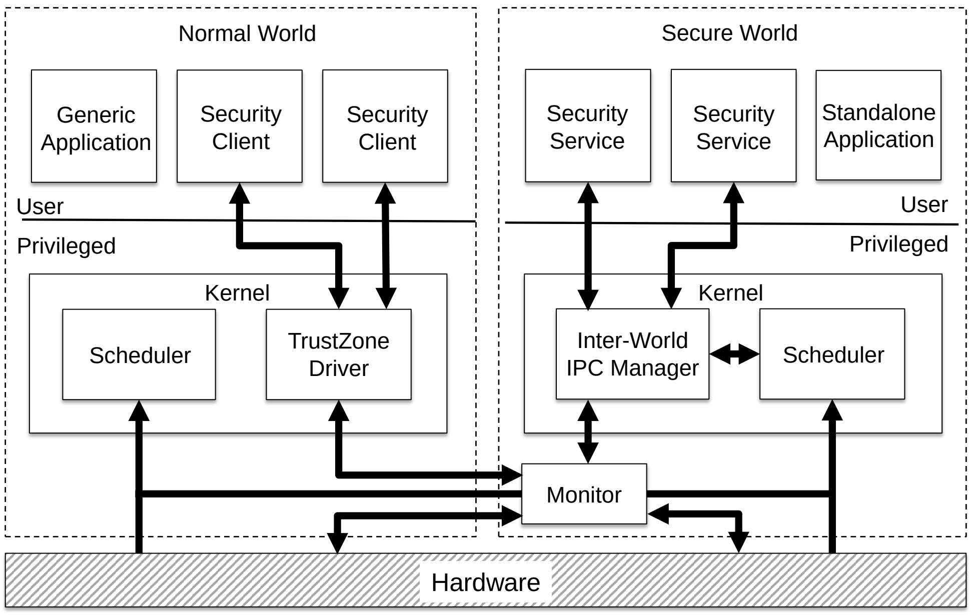

Figure 2.2: TrustZone software architecture (adapted from ARM TrustZone whitepaper).

2.4 Secure Processing based on ARM TrustZone

In contrast to Intel SGX, which was developed for secure computing on the cloud, ARM has a hardware-based trusted computing technology developed to protect data in mobile devices. This technology is called ARM TrustZone. ARM TrustZone is a security extension present in ARM processors that provides a hardware-level isolation between two execution domains: *normal world* and *secure world*. Compared to the normal world, the secure world has higher privileges, as it can access the memory, CPU registers and peripherals of the normal world, but not the other way around. This different component isolation is enforced by a secure monitor present in the secure world. More specifically, TrustZone checks and controls the state of the CPU through a NS bit, and partitions the whole memory address space into secure and non-secure regions. In order to perform a context switch between the different worlds, TrustZone offers the `SMC` instruction, which generates a software interrupt that is then handled by the secure monitor. TrustZone follows a similar approach with interrupts, where IRQs and FIQs can be configured to be either secure or non-secure interrupts. This means that secure interrupts can only be intercepted by the secure world. This interrupt approach in combination with memory isolation is fundamental in guaranteeing peripheral access isolation between worlds. Additionally, TrustZone offers a secure boot mechanism that ensures the integrity and authenticity of the OS running on the secure world, at system boot time.

TrustZone hardware and software architecture are described in a whitepaper [4] by ARM. A typical architecture using TrustZone is shown in Figure 2.2. This design allows for concurrent execution of multiple secure world apps and services that are completely independent from the normal world environment, thus, even with a compromised normal world, the secure world executes as expected. Moreover, the kernel design can enforce the logical isolation of secure tasks from each other, preventing one secure task from tampering with the memory space of another. These advantages sparked the interest of the research community in developing several systems which leverage this technology.

2.4.1 TrustZone Isolation

To isolate the rich OS from the secure world ARM TrustZone hardware checks and controls the state of the CPU through a NS bit, and partitions the memory address space into secure and non-secure regions. This execution strategy removes the need for a dedicated security processor core and allows high performance security software to run alongside the normal world. Another security feature available in TrustZone-enabled processors is the ability to secure peripherals, such as interrupt controllers, timers and user I/O devices. This means that secure applications with needs beyond a simple secure data processing environment can use this feature to protect critical hardware components of the platform.

To perform a context switch between the different worlds, TrustZone offers the `SMC` instruction, which generates a software interrupt that is then handled by the secure monitor mode software. The secure monitor mode software is the component responsible for managing the switches between the secure and non-secure processor domains. This secure monitor mode software is implementation defined, meaning that the developer is in full control of what happens when an interrupt is trapped by the monitor. The ARM TrustZone specification [4] suggests the actions to be taken in consideration by the monitor mode software. Namely, the specification describes that this monitor mode generally saves the state of the current world and restores the state of the world being switched to. The monitor then performs a return-from-exception instruction to restart processing in the previous world from where it was interrupted before.

Beyond the dedicated SMC instruction, there are other hardware mechanisms by which the processor can enter monitor mode from the normal world. These mechanisms are tightly controlled and can be configured to fit several requirements of a specific design. Basically, Interrupt Requests (IRQs), Fast Interrupt Requests (FIQs), external Data Abort and external Prefetch Abort can be configured as either secure or non-secure interrupts. This means that secure interrupts can only be intercepted by the secure world and not the normal world. Both interrupt management and memory isolation mechanisms are fundamental in guaranteeing peripheral access isolation between worlds.

TrustZone in itself allows systems built to be executed in the secure world to effectively discard the need for trusting a rich operating system and applications running on it. These components generally compose the trusted computing base of applications running on a computing platform because applications rely on operations performed by these components in order to work properly. Big and complex software platforms are inherently more vulnerable to attacks because they have a higher probability of coding vulnerabilities, have interactions between several components which may have different security levels and concerns, and lastly because complex software generally has a larger attack surface. When applications rely on these big and complex software platforms they are inheriting the vulnerabilities and security concerns of those systems.

By providing isolation between the rich operating system and the security critical system, TrustZone is reducing the trusted computing base of the secure world application to the secure hardware and to what is strictly implemented by the developer. This removes the security concerns and vulnerabilities of a rich OS from the secure system. The disadvantage of this approach is that the secure system must implement the software stack needed to work properly, whereas before it could enjoy features implemented by other software components, such as the rich OS. The design choice of TrustZone was to give control and flexibility to developers, so they can adapt the systems to their particular needs.

Table 2.3: TrustZone-based system categorization.

Trusted Kernels	Trusted Services
Genode	TrustOTP
T6	TrustDump
TLK	AdAttester
TrustICE	TrustUI
SierraTEE	Android Key Store
OP-TEE	DroidVault
Andix OS	Brasser et al.
Samsung KNOX	
Nokia ObC	
TLR	

2.4.2 TrustZone Applied to Mobile

Given that the majority of mobile devices are equipped with ARM processors, TrustZone has been studied mostly to overcome security issues on mobile platforms. Many authors propose solutions based on TrustZone-enabled TEE for hosting mobile security services, which allow for: detecting and preventing mobile app ad frauds [26], implementing OS introspection mechanisms [15], enabling secure data storage [28], providing secure authentication mechanisms [31], implementing one-time-password solutions [50], or providing forensic tools for trusted memory acquisition [49]. Other systems provide general-purpose frameworks for splitting mobile app code and run it in the TEE [40] or enabling trusted I/O between the user and TrustZone-based services [27].

To improve security, device manufacturers have been designing hardware architectures enhanced with trusted hardware. Among the available security architectures there is ARM's TrustZone technology, a trusted hardware which allows the development of a diverse set of security systems and services, such as Samsung KNOX [38] and DroidVault [28]. TrustZone is becoming popular for mobile devices as it supports code to be executed isolated from a full-featured operating system such as Android. This enables a reduction of the trusted computing base on which critical applications depend.

One of the most important uses of TrustZone is building trusted execution environments, which are compact systems running in the secure world to provide an isolated environment for critical applications. Since its formal standardization by the OMTP in 2007, several TEE software stack architectures have been implemented. This standard comprises a set of security requirements on functionality a TEE should support. The GlobalPlatform [18] organization went a step further by defining standard APIs: the internal APIs that a trusted application can rely on and the communication interfaces that rich OS software can use to interact with its trusted applications.

Table 2.3 categorizes the existing systems in two main dimensions. When surveying the literature, TrustZone-based systems can be divided into two separate groups depending on whether they support general or specific application code hosting: Trusted Kernels, and Trusted Services. Trusted Kernels, which comprise the TEEs, allow the execution of generic code in the secure world environment. And Trusted Services implement special-purpose applications in the secure domain and can run directly on bare metal (e.g., a secure key store, an authentication service, etc.).

The remainder of this section describes in more detail the existing TrustZone-enabled systems according to both categories: *(i)* Trusted Kernels and *(ii)* Trusted Services.

2.4.2.1 Trusted Kernels

Trusted Kernels have the goal of executing generic code in its isolated environment, and most of theses kernels have similar architectures to that shown in Figure 2.2. This architecture is generally composed of a small trusted kernel running in the secure world of TrustZone processors, a normal world user space client API and a kernel TEE device driver, used to communicate between worlds.

OP-TEE [30], TLK [56], TLR [40] and AndixOS [13] are TEE implementations which share this general architecture. On-board Credentials (ObC) [25] is another TEE system, originally developed for Nokia mobile devices using the TI M-Shield technology and later ported to ARM TrustZone. ObC supports the development of secure credential and authentication mechanisms. Although these systems use TrustZone hardware-based isolation to ensure that applications running inside the secure world are not modified by a compromised rich OS, they were implemented with the goal of reducing the TCB in order to ensure a less vulnerable system. For this reason there are some limitations regarding the features they can support.

A reduced TCB means that most features of standard mobile operating systems are not supported. For instance, in both TLR and OP-TEE, as well as AndixOS, the secure world kernel lacks drivers for peripherals such as the touchscreen or code to control the framebuffer. Thus, these systems are not capable of supporting trusted UI. For this reason, these systems do not allow developers to easily build trusted applications for sensitive data display. Instead, these sytems support an RPC-like mechanism for in-between-world communication, secure persistent storage and basic cryptographic systems allowing for the development of simple trusted services. On one hand, because these systems were developed for mobile devices, not supporting trusted UI is a factor which limits the use of these systems. On the other hand, since our system is a cloud-based data processing service, and trusted UI is not a necessary feature, these systems comply with some of our goals and could be used as a building block.

Other systems were developed with the focus of overcoming the limiting factors of not supporting trusted UI. As mentioned before, GlobalPlatform defined standard APIs for the communication between the rich OS running in the normal world and the secure OS. However, this organization also defined device specifications that TEEs must comply in order to be certified. Included in these device specifications there is a trusted UI clause, meaning that every TEE which complies with GlobalPlatform's device specifications must support trusted UI.

SierraTEE [45], T6 [52] and Open-TEE [33] comply with the GlobalPlatform standard, and for this reason allow the development of trusted applications with secure user interfaces. Open-TEE's trusted UI feature is being developed by the community as it was not originally supported. The Genode OS Framework [23] is a tool kit for building highly secure special-purpose operating systems to be executed in TrustZone-enabled processors. Genode implements a framebuffer and display drivers to be used by the secure kernel, thus trusted applications running on top of Genode-based TEEs can offer trusted user interfaces. But, because Genode is an OS framework, it allows developers to build operating systems according to their needs. Thus, Genode can be used to build trusted kernels which have no need of supporting trusted UI by removing the software modules which implement it.

Samsung KNOX [38] is a defense-grade mobile security platform which provides strong guarantees for the protection of enterprise data. Security is achieved through several layers of data protections which include secure boot, TrustZone-based integrity measurement architecture (TIMA) and Security Enhancements for Android (SEAndroid [47]). Samsung KNOX offers a product called KNOX Workspace, which is a container designed to separate, isolate, encrypt, and protect work data from attackers. This enterprise-ready solution provides management tools and utilities to meet security needs of large and small enterprises. Workspace provides this separate secure environment within the mobile device, complete with its own home screen, launcher, applications, and widgets.

Unlike the solutions previously described, such as Genode OS and Samsung KNOX, which provide isolated computing environments in the secure world, TrustICE [51] aims at creating Isolated Computing Environments (ICEs) in the normal world domain. For this reason, TrustICE's architecture is slightly different from those described before.

Figure 2.3 compares TrustICE's architecture with that of a traditional TrustZone TEE, where trusted applications run inside the secure world. TrustICE works by implementing a trusted domain controller (TDC) in the secure world, which is responsible for suspending the execution of the rich OS as well as other ICE's when another ICE is running. Thus, TrustICE supports CPU isolation for running ICE's. For memory isolation a watermarking mechanism is implemented so the rich OS cannot access code running in the normal world memory. In order to isolate I/O devices the secure world blocks all unnecessary external interrupts from arriving at the TDC, thus protecting the TDC from being interrupted by malicious devices, the exception being a minimal set of required interrupts to allow trusted UI.

In summary, the research community has made an effort in allowing generic code to be deployed on the secure domain of ARM TrustZone-enabled processors. Some of these systems aim at reducing the TCB considerably, whilst others support additional features such as secure I/O. For the development of our system, the main goal is having a reduced TCB rather than supporting additional features. For this reason, a good candidate as a building block for our system is the Genode OS Framework, as it allows to build a TrustZone-aware micro-kernel stripped of all unnecessary functionality, thus accomplishing the goal of having a reduced TCB.

2.4.2.2 Trusted Services

As opposed to Trusted Kernels, which enable the execution of general-purpose application code on the secure world, Trusted Services are designed to implement specific applications in the secure world natively. Some trusted services, such as DroidVault [28] and Restricted Spaces [8], use custom trusted kernels to fully control the underlying hardware and execution environment.

A system by Brasser et al. [8], which will be referred to as Restricted Spaces for the remainder of this section, allows for third-parties (hosts) to regulate how users (guests) use their devices (e.g., manage device resources), while in a specific space. This system comprises authentication and communication mechanisms between the guest's secure world and the host's. It also supports remote memory operations, which allow for configuration changes such as uninstalling peripheral drivers. This can be done by either pointing their interfaces to NULL or to dummy drivers that just return error codes. With this, Restricted Spaces is capable of securely refine permissions using a context-aware approach.

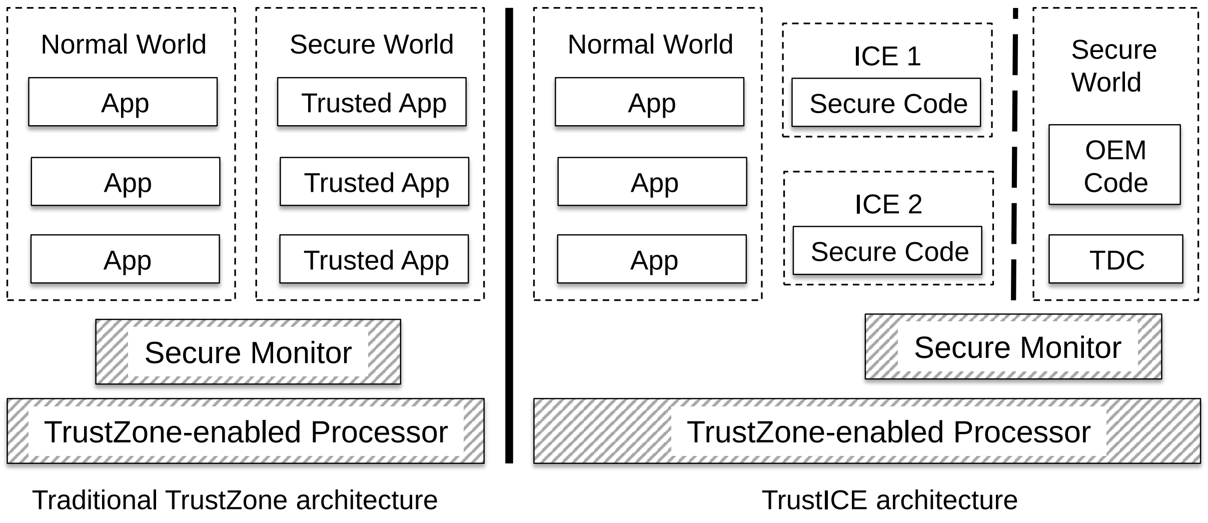

Figure 2.3: Architecture comparison between traditional TrustZone's software stack and TrustICE

DroidVault [28] introduces the notion of data vault, which is an isolated data protection manager running in the trusted domain for secure file management in Android. To achieve this, DroidVault adopts the memory manager and interrupt handler from Open Virtualization's SierraTEE [45] and is implemented with a data protection manager, encryption library and a port of a lightweight SSL/TLS library called PolarSSL [32]. Much like Restricted Spaces, DroidVault supports world switching through software interrupts, secure boot and even inter-world communication. With this Trusted Service a user can download a sensitive file from an authority and securely store it on the device. The sensitive file is encrypted and signed by the data protection manager before it is stored in the untrusted Android OS, in order to save space in the limited storage capacity available at the secure world.

Android Key Store [2] is another security service in Android. This service allows for cryptographic keys to be stored in a container (keystore), so its extraction from the device becomes difficult and so they can be used for cryptographic operations. The encryption and decryption of the container is handled by the keystore service, which in turn links with a hardware abstraction layer module called "keymaster". The Android Open Source Project (AOSP) provides a software implementation of this module called "softkeymaster", but device vendors can offer support for hardware based protected storage by using TrustZone.

TrustOTP [50] is a One-Time-Password (OTP) system, secured by hardware, where the OTP is generated based on Time and a Counter secured by TrustZone's memory management. Most trusted applications described before require inter-world communication to trigger the world-switching mechanism. This system leverages hardware interrupts to trigger the world-switch. This mitigates denial-of-service attacks by a malicious rich OS which may control the inter-world communication mechanism and intercept the calls (software interrupts) required to trigger the world-switch.

Providing a different service, TrustDump [49] is a secure memory acquisition tool that offloads the memory through micro-USB. Similarly to TrustOTP, this system relies on hardware interrupts to trigger world-switches. This solution may be implemented by systems which require no inter-world communication, but for systems which need to offer seamless integration with the normal world this approach may have to be leveraged with other development strategies. Both of these systems support trusted UI by implementing secure display and input drivers, as well as display controllers to manage the secure framebuffers.

Instead of implementing the required drivers to support additional features, such as trusted UI, some systems designed mechanisms to allow the reuse of untrusted drivers, implemented in the rich OS, by the secure world domain. TrustUI [27] excludes the device drivers for input, display and network from the secure world, and instead reuses the existing drivers from the normal world, thus achieving a much smaller TCB than previously described systems. Because we are only interested in secure display, the following explanation discards the network delegation mode and input mechanism. To achieve trusted UI, device drivers are split into two parts: a backend running in the normal world domain and a frontend running in the secure world. Both parts have corresponding proxy modules running in both worlds, which communicate via shared memory. Whenever secure display is necessary, the frontend asks for a framebuffer from the backend driver and sets that memory region as secure only, thus isolating the framebuffer from rich OS manipulation.

This mechanism can still be victim of framebuffer overlay attacks, where a malicious backend driver gives a false framebuffer to the secure world. For this reason, the system randomizes the background and foreground colours used in the display and uses two LEDs, controlled by the secure world, to show these same colours. A user can visually check if the colours shown in the secure LEDs match those of the display. If they match then the user is assured that the display shown is being controlled by the secure world.

These systems support secure display, as such, they do not disclose sensitive data when it is displayed to the user. However, TrustOTP and TrustDump do not offer a fully integrated environment with the Android running in the normal domain. TrustUI, on the other hand, fully integrates its environment with that of the Android operating system. Furthermore, TrustUI describes a novel mechanism for the reuse of untrusted driver in the normal world without compromising security, which significantly reduces the secure system's TCB. The main disadvantage of TrustUI is that it is not immune to denial-of-service attacks by a malicious operating system running in the normal world, which may compromise the execution of the secure system. Although denial-of-service attacks are problematic for the correct execution of the system, these attacks cannot disclose the sensitive data being protected since this data can only be read by the secure world.

2.4.3 TrustZone Applied to the Cloud

Since ARM processors are mainly used extensively for the mobile market, TrustZone has been explored for protecting data inside mobile devices. However, attempts have been made on exploring this trusted computing technology for the cloud. Brenner et al. [9] took some first steps at using ARM TrustZone on the cloud by building a TEE-protected privacy proxy for Zookeeper [19]. Zookeeper is a fault-tolerant coordination service for distributed systems which allows the implementation of coordination tasks such as leader election and locks via file system like client API. This interface allows client to manage the so called `znodes` that are payloading files and folders simultaneously.

The goal of Brenner et al. was to protect the privacy of all data stored inside Zookeeper. This is done by leveraging Zookeeper Privacy Proxies (ZPP) as a lightweight and transparent encryption layer running inside the trusted execution environment created by using ARM TrustZone. They place ZPPs in between the Zookeeper clients and the Zookeeper replicas in such a way that client and server implementations remain unmodified. Zookeeper clients connect to ZPPs like they would connect to Zookeeper server replicas and ZPPs connect to real Zookeeper server replicas as traditional clients would. For each client session, a ZPP receives a packet from the client protected using SSL encryption, it then extracts and gathers all the sensitive information which is then encrypted for secure storage and sent to the real Zookeeper replica.

Brenner et al. laid the first stone of applying ARM TrustZone to the cloud, but the focus of their project was different from what we hope to achieve in this dissertation. They use their system as a transparent encryption layer to an already existing service, which constitutes a different application scenario than the focus of our work. With this dissertation we hope to continue the foundation work started by Brenner et al. of applying the TrustZone technology to the cloud.

2.5 Discussion

The advantage of trusted services is the reduced Trusted Computing Based inherent to the sole implementation of strictly necessary components and features. Trusted kernels have to implement features to support generic code execution which in turn means it needs additional code, thus a higher TCB. For this reason, and because our system does not need to execute generic code, it closely resembles more a trusted service rather than a trusted kernel. On the other hand, a lot of features supported by our system must be built on top of common kernel components such as memory management and interrupt handling. Because of this, our system needs a custom micro-kernel in order to achieve the goals proposed in this dissertation.

And so far we have discussed two hardware-based isolation technologies which allow secure processing of data inside a container. Intel SGX was developed to be used in the context of cloud computing, while ARM TrustZone was originally developed for secure mobile computing. The major difference between these two technologies is that, unlike ARM TrustZone, Intel SGX allows more than one container (called enclave) for each processor, meaning that the hardware can protect several secure containers at the same time. This is inherently better for cloud computing since it is common for a cloud server to be processing data from different users and applications at the same time. In addition, Intel SGX provides hardware based attestation and sealing methods to allow exporting secrets to outside the enclave (for example to store them in persistent memory). These secure features are achievable through the use of a new x86 instruction set architecture (ISA) extension.

This means that for our system, which aims at protecting data processed on the cloud, Intel SGX would be the obvious choice. But Intel SGX is a fairly recent technology, still in active development, which is being explored only by a select group of researchers and is not yet available on commodity hardware. ARM TrustZone, on the other hand, is a more mature and widely available technology, which has been widely explored by the research community for secure mobile computing, and has generally a simpler architecture. Additionally, exploring ARM TrustZone, which was originally developed for mobile computing, for the cloud is a challenging research problem. For these reasons we chose ARM TrustZone as the hardware-based isolation technology for our secure data processing system.

Summary

This chapter described the related work regarding secure data processing on the cloud. We started by describing techniques and security issues regarding image processing on the cloud as well as some image processing projects which handle sensitive image data that can be used nefariously if disclosed. This first part increases the motivation behind exploring the use case of secure image processing on the cloud for our system. We then gave an overview on cloud security mechanism and why these mechanism are not sufficient for secure data processing. We first expose the flaws present in basic encryption techniques, where data processed at one end must always be processed in its raw (unencrypted) format, thus exposing sensitive user data to untrusted components. We then describe the limitations of advanced cryptographic techniques, such as fully homomorphic encryption schemes. These allow data to be processed without exposing it to untrusted components but have huge limitations on the data format to process, as well as high overheads. We then explore the advances made using virtualization techniques to protect code and data being executed inside software containers. But this technique often relies on large and complex trusted computing base, which can have vulnerabilities.

We then describe two hardware-based isolation technologies which can be applied for secure data processing on the cloud. ARM TrustZone and Intel SGX are security extensions to processors which allow the creation of one or more secure hardware-managed containers for processing sensitive data. We also discuss the fact that Intel SGX is a fairly recent technology only available to a select group of researchers. This justifies our choice of using ARM TrustZone as our hardware-based isolation technology. The next chapter will introduce the design details of our system.

3 Design

This chapter describes the design of our system, called Darkroom. In order to securely compute sensitive data, Darkroom will leverage the security primitives supported by ARM TrustZone. This technology enforces hardware-level isolation between domains with different execution privileges. But, in order to ensure that Darkroom maintains the security properties of confidentiality, integrity and authentication for sensitive data, additional requirements must be accounted for, namely: keeping a small TCB, end-to-end security, reduced overhead and applicability to secure image processing.

In this chapter, we start by describing the assumptions and threat model taking into account when designing Darkroom (Section 3.1). We then describe the architecture of our system by presenting a typical execution flow of Darkroom for secure image processing and the components which comprise the system (Section 3.2). We then describe the process of deploying Darkroom on ARM-based cloud servers as well as explain the secure system bootstrapping feature (Sections 3.3 and 3.4, respectively). The deployment process and secure bootstrap are used together to give clients guarantees that their data is being processed by a Darkroom instance. We then conclude this chapter by discussing how image transformation functions are supported by the system (Section 3.5) and how Darkroom uses these functions to perform image transformations on sensitive images (Section 3.6).

3.1 Assumptions and Threat Model

In order to justify the solutions present in our design, we must describe the assumptions and threat model considered for Darkroom. As mentioned before, Darkroom aims at protecting user data processed on a cloud platform from being exposed to untrusted components such as the operating system and user applications. Since we are assuming an untrusted software platform, we can also assume a powerful adversary which may acquire control of most of the provider's software infrastructure. To this end, we consider as untrusted all software running in the normal world of our TrustZone-aware platform.

As for the hardware, we assume that the processor is correct. This means that we assume all TrustZone security features supported by the processor are correctly implemented and cannot be compromised by an attacker. So, our threat model consists of a powerful adversary who controls the cloud provider's software stack, including the rich operating system and applications running in the normal world. This results in an adversary which might interrupt the execution of the user's program indefinitely (denial-of-service attacks are not considered by Darkroom) and falsify the results of calls made to the operating system. Additionally, the adversary also controls hardware devices besides the processor, memory and system bus. On the other hand, we do not consider side-channel attacks, since most solutions to these attacks require hardware modifications which are out of the scope of this dissertation.

Note that we are primarily concerned with potential security breaches resulting from software exploits to normal world programs. In particular, we want to prevent external attackers that manage to take over control of the application or the operating system residing in the normal world from violating the confidentiality of users' images (e.g., by leaking them out). We assume that both the cloud provider and cloud administrators are trusted. This is, we assume the cloud provider and cloud administrators are not colluding with an adversary and are not the adversary themselves. The trusted computing base (TCB) of our system comprises the hardware platform (TrustZone processor) and the Darkroom components hosted in the secure world.

3.2 Architecture

In order to describe the architecture of Darkroom we will start by following a typical execution flow of a remote client application leveraging Darkroom for secure image processing. Afterwards, we briefly describe the components present in the execution flow, as well as explain the reasoning behind the design of this architecture.

Figure 3.1 shows the components of our solution and represents the possible execution flow. The flow starts at a client application, which is represented by the mobile device. The client application sends an encrypted image to the Image Cloud Service component, which can store the image data locally. Both the Image Cloud Service and the operating system run in the normal world context of the TrustZone-enabled processor. After uploading the image data, the client application issues a transformation request for the image. Upon receiving a transformation request, the Image Cloud Service sends the encrypted image data to the secure world. It also sends the transformation request and triggers a world switch via a Secure Monitor Call (`SMC`). This `SMC` instruction gives control to the secure world which decrypts the image data and executes the requested transformation on it. After processing, the image is encrypted once again and sent to the normal world.

In this figure we can identify the components which comprise Darkroom in the secure world and the untrusted components present in the normal world. Starting by the normal world, the first component which interacts with the sensitive encrypted data is the Image Cloud Service. This component represents the server application of image processing services such as Facebook or Instagram. This is an untrusted application which leverages the Darkroom API in order to use the secure world processing capabilities provided by Darkroom. The Darkroom API is the component responsible for sending the security critical data to the secure world and is also responsible for triggering the world-switch via the `SMC` instruction. Once the world-switch is triggered, the secure Darkroom kernel receives the encrypted data and redirects it to the two remaining secure world components. The Crypto Engine is the secure component responsible for decrypting the image for processing, and encrypting it again before it is sent back to the normal world. The Image Processing Engine is the secure world component responsible for manipulating the unencrypted data.

As mentioned in Section 2.4.1, the use of ARM TrustZone for isolation does allow for a relative reduction of the trusted computing base of a security critical system such as Darkroom. But there is another design choice which has a direct impact on the system's TCB and on other design choices. ARM TrustZone specification [4] defines three main types of software architectures of TrustZone-aware

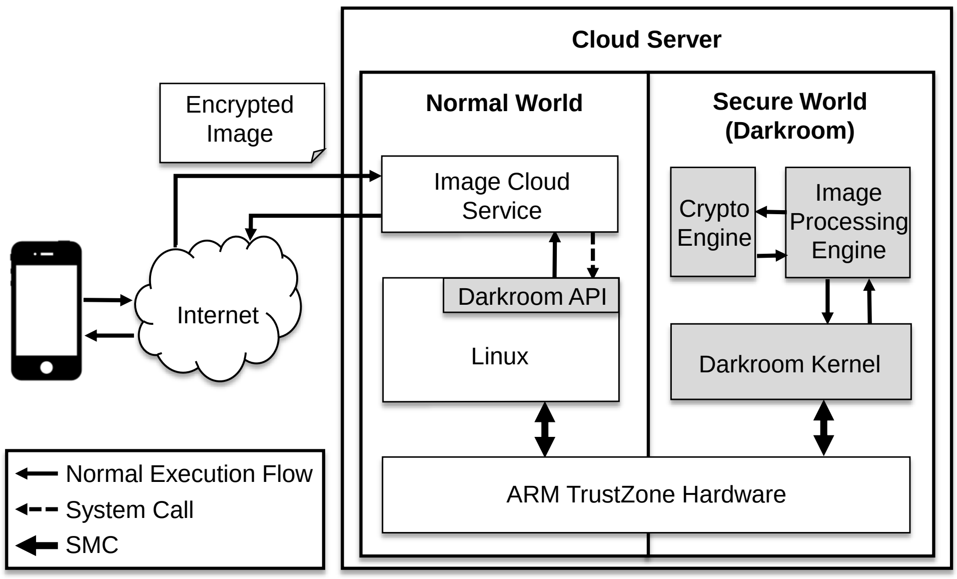

Figure 3.1: Darkroom Architecture.

systems. The first is the use of a dedicated secure world operating system, the second is a synchronous library of code running inside the secure world and the third is an intermediate option between a dedicated secure world OS and a secure world synchronous library of code. All of these architectures have advantages and disadvantages characteristics to what they support.

The dedicated secure world operating system approach is the most complex architecture for the software running in the secure world, but it is also the most powerful approach. This is because having a dedicated custom OS running in the secure world allows the execution of multiple concurrent secure world applications running on top of the secure OS. Another advantage of using this approach is that applications running on top of it can be completely independent from the normal world OS. Additionally, the processor's MMU can be used to separate secure world memory region into multiple user-space containers which allows several user-level secure tasks to be executed concurrently without having to trust each other. This means that a secure task cannot temper with the memory space of another task. For Darkroom this approach is too complex and may result in a larger TCB than that necessary for our system. Additionally, this approach lacks the inter-world communication desired for Darkroom. One advantage of building Darkroom on top of a secure world OS is that Darkroom could enjoy programming abstractions which are typically present in operating systems, such as memory management, dynamic memory, threading and scheduling.

The other extreme is a synchronous library of code running in the secure world. This approach is generally used by applications which do not require the complexity of an OS running in the secure world and thus depend on a much smaller and simpler TCB. The advantage of such a library is that it can be used together with the normal world OS via special purpose software calls. This means the secure world application and its normal world counterpart cannot work independently of one another, thus if the normal world OS is compromised the application running on top of it can effectively launch a denial-of-service attack. As explained in our threat model, Darkroom does not consider denial-of-service attacks. This is mostly because denial-of-service attacks do not compromise the confidentiality of Darkroom, it compromises only its availability. Since we are primarily concerned about confidentiality protection, a library-based approach could work for Darkroom.

Since Darkroom needs the inter-world communication and small TCB offered by the synchronous library approach, but also needs the OS abstractions and flexibility offered by a secure world OS, we designed Darkroom as a small-sized kernel that provides basic features only required for image processing, such as memory management, threading for supporting the several secure world components and scheduling. Additionally, an inter-world communication must be supported by this secure world kernel via the normal world Darkroom API.

3.3 Deployment on Cloud Servers

We envision Darkroom to be deployed on cloud servers maintained by the service provider. As opposed to Intel-based servers commonly adopted in today's cloud infrastructures, Darkroom's target servers must be based on ARM TrustZone technology. To deploy the system, the cloud provider must flash the firmware of its ARM servers so that the servers bootstrap into the secure world and run Darkroom's setup code before switching to the normal world and loading up a rich operating system or hypervisor. The details regarding Darkroom's boot process are described in Section 3.4.

In this process, the cloud provider must generate a *root key* for each server. A root key is a public key pair KR whose private part (KR^{-}) is bundled into the Darkroom image right before the image is flashed onto a server's firmware. The private part of the root key will be accessible only by Darkroom within the secure world and there will be a unique root key for each cloud server. The public part KR^{+} is certified by the cloud provider in order to assert the authenticity of this key. This key is fundamental to ascertain the authenticity of the Darkroom platform to a remote client and to securely share secrets with the Darkroom runtime without the need to trust the rich operating system.

Note also that, in addition to the root key, the Darkroom image contains public keys of cloud administrators authorized to perform some security-critical operations, e.g., setting up image transformation functions, as explained in Section 3.5. The KR^{-} key bundled inside the Darkroom image is a fundamental part of the chain of trust started by the SoC ROM bootloader and guarantees to a remote client that this platform is running a Darkroom instance, which in itself was authenticated by the platform's bootloader. Details about the secure system bootstraping process are explained next.

3.4 Secure System Bootstrapping

Figure 3.2 represents the boot sequence of Darkroom. In order to execute the critical security verifications before the normal world can temper with the system, TrustZone-enabled processors start executing in the secure world immediately after power on. After that, a ROM-based bootloader, responsible for initializing critical components such as memory controllers, is executed. After these critical components are initialized, Darkroom's bootloader, stored in non-volatile memory, is launched and Darkroom boots. It is on this step that we manage the critical components initialized by the ROM bootloader

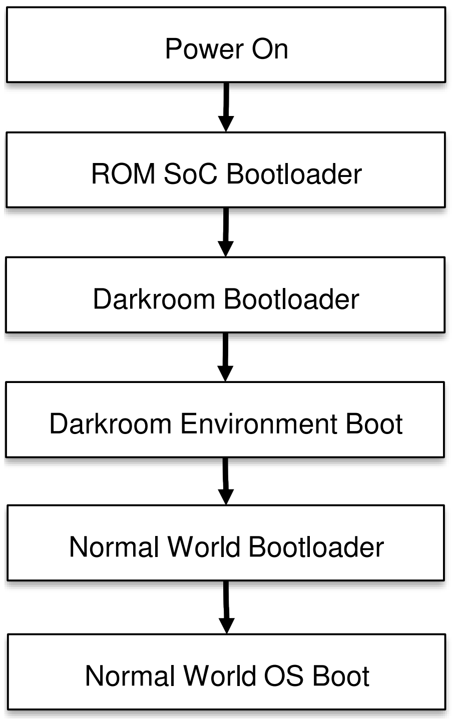

Figure 3.2: Darkroom Boot Sequence.

in order to protect these components from normal world tampering. The details regarding critical components and component permission management are specific to each TrustZone-enabled platform and will be explained fully in Chapter 4. Afterwards, Darkroom switches to the normal world and launches the normal world bootloader, which consequently boots the normal world rich OS.

In order to prevent attacks which may replace the secure world binary, cryptographic checks must be performed throughout the boot process. This guarantees the integrity and authenticity of software running in the secure world. Darkroom assumes the trusted platform vendor establishes a *vendor key pair* and deploys the public part KV^+ on the device, such that it cannot be replaced by an attacker. This public key is used for a cryptographic signature protocol where the vendor uses the private key KV^- to generate a signature of Darkroom's code which will be flashed along side Darkroom's binary when the system is deployed on ARM-based cloud servers. Using this protocol, the bootloader located in ROM SoC can start a chain of trust where every software component is authenticated before executing, namely Darkroom's bootloader. The chain of trust must start in SoC ROM memory and with the ROM-based bootloader because this component is the most difficult to modify by an attacker performing, for example, a reprogramming attack. After verifying the authenticity of both Darkroom's bootloader and binary image, Darkroom's bootloader is launched and boots Darkroom, thus concluding the chain of trust and the secure bootstrapping process.

On one hand the secure boot process guarantees that Darkroom's image was not tempered by an attacker and the cloud server deployment process guarantees to a remote client that the system is running a Darkroom instance. On the other hand, the secure boot and deployment processes do not guarantee to a remote client that Darkroom is effectively running in the secure world (container) of a TrustZone-based processor. This is because, unlike Intel SGX, ARM TrustZone does not support hardware instructions which can generate unique keys and ids for secure containers and does not support instructions for authenticating these containers to third parties. So, by guaranteeing that Darkroom's runtime was not modified by an attacker and that the data is guaranteed to be processed by a Darkroom instance, we achieve part of Darkroom's goal of end-to-end security. The remaining parts of this goal, which consider secure storage of the sensitive data and checking the result of Darkroom's processing will be explained in the following sections.

3.5 Setting Up Image Transformation Functions

Once a cloud server is up and running, the rich operating system takes up control of system resources and Darkroom is suspended until requests arise from the normal world to perform image transformation operations (e.g., image rescaling). To allow for additional flexibility, rather than hardcoding transformation functions (TF) into the firmware, these functions can be loaded dynamically by cloud administrators into the Darkroom runtime. This makes it easy to upgrade the system with new or more efficient functions without the need to reflash the servers' firmware.

To support dynamic loading of transformation functions, we must ensure that the code to be loaded is trustworthy. This is because this code will have access to the raw image data submitted by client applications and can potentially compromise that data, e.g., by leaking it from the server. In addition, we must provide mechanisms that allow transformation functions to be uniquely identified in order to ensure that the correct transformation is applied to a given image. Such mechanisms must be able to tolerate modifications in the set of available transformation functions, as these can be installed from the system.

To ensure that the transformation functions installed into the Darkroom runtime are trustworthy, we require that such functions are installed or uninstalled only by trusted cloud administrators. In particular, to install a TF, a cloud administrator must issue a *load* request which must be signed by the administrator key (KA). Darkroom validates the request against the public part of the authorized admin key $KA+$ and allows the operation to proceed if the test passes; otherwise the operation is aborted. To uninstall a TF from the system, the corresponding *unload* request must also be signed by a trusted cloud administrator. To allow for unique identification of a given TF, Darkroom leverages the hash of the TF binary. This binary is included into the (signed) load request provided to Darkroom. This request contains additional metadata that indicates the TF name and the type of input parameters, e.g., *rotation (int degree)*.

3.6 Performing Image Transformation Operations

Normally, performing image transformations involves a four-stage life cycle. First stage is to securely *setup* a shared key between the client and the secure service. This stage is only performed once per client and will allow the server to check the integrity of the messages exchanged after this initial setup. The second stage is to securely *upload* a security-sensitive image from the client to the cloud server. In this process, we must ensure that the image is encrypted in such a way that it can only be decrypted within the Darkroom runtime. Third stage corresponds to *transforming* the image by invoking a transformation function of Darkroom. Note that one or more transformations can be chained together (e.g., rotation followed by scaling, etc.). Fourth stage is to *download* the result of the transformed image, while allowing only the client to decrypt the resulting image. In addition, the client must be able to trace the authenticity of the transformation sequence, in order to ensure that the resulting image derives from a valid sequence of transformations properly applied to the original image submitted by the client to the cloud provider.

But before explaining these stages, we must describe a necessary pre-configuration step. In particular, Darkroom must be set up with a public key pair called *service key* (KS). The role of this key is to associate the image transformation operations to the context of a specific cloud service (whose logic runs in the rich OS and client). To this end, cloud administrators must generate a public key pair KS and securely load it to the Darkroom runtime of each cloud server. Security is achieved by mutually authenticating the cloud administrator's key with the server's root key. Once the service is loaded into each server, it is now possible to perform the following four stages, represented visually in Figure 3.3:

1. Initial setup: To allow the secure Darkroom service and client to check the integrity of future exchanged messages, the client needs to share a secret key with the secure service. The client generates a symmetric key KC which will be the client key associated to that client. This key is sent to Darkroom encrypted with the public service key of the server KS^+. Additionally, the client also sends a challenge to account for freshness in the exchange. The secure service receives this request and generates a client identifier cID which the client must use in future communications in order to identify the messages. The server then sends this client ID together with the answer to the client's challenge and encrypts this message with the newly exchanged shared symmetric key.

2. Image upload: To securely upload the image to the cloud server, the client simply generates a symmetric key KI and encrypts the image with that key. Then, KI is encrypted with the public key KS^+ to ensure that the image can be decrypted only by Darkroom-enhanced servers allocated to the service to which KS^+ is bound. For integrity verification, the client also computes the HMAC of the message using the client key KC, where the message m is the concatenation of the encrypted image and encrypted key KI. The resulting blob $\{I\}_{KI}\{KI\}_{KS^+}$*HMAC*(m, KC) is then uploaded to the cloud service. We name this blob: *envelope*. With this envelope stored on the normal world OS Darkroom achieves the goal of securely storing the sensitive data in the normal world operating system. This is because the envelope comprises the image data in such a way that only Darkroom can access the raw content. Additionally the client also sends an image identifier iID to identify this envelope inside the Image Cloud Service.

3. Image transformation: When the client requests a transformation on a specific image it needs to send a transformation request to the Image Cloud Service running on the server platform. In this message the client needs to specify the client ID (cID), so Darkroom can select the correct client key KC, the identifier of the image to be transformed (iID), the identifier of the new resulting image $niID$ and the transformation identifier tID (note that more than one transformation can be requested at the same time). The transformation identifier is encrypted with the Darkroom's public key KS^+ in order to maintain the transformation request confidential between the client and the secure service. Additionally, the client also sends an HMAC for integrity checking.

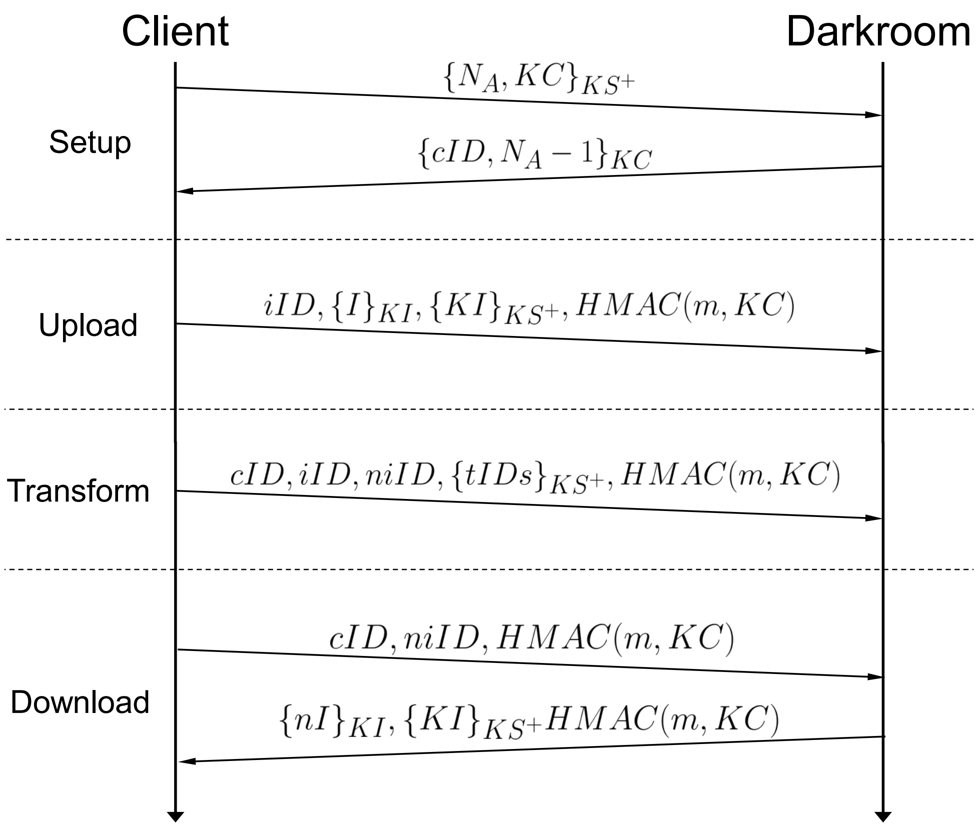
Client
Darkroom
Setup
$\{N_A, KC\}_{KS^+}$
$\{cID, N_A - 1\}_{KC}$
Upload
$iID, \{I\}_{KI}, \{KI\}_{KS^+}, HMAC(m, KC)$
Transform
$cID, iID, niID, \{tIDs\}_{KS^+}, HMAC(m, KC)$
Download
$cID, niID, HMAC(m, KC)$
$\{nI\}_{KI}, \{KI\}_{KS^+} HMAC(m, KC)$

Figure 3.3: Communication of Darkroom processing stages.

Whenever the cloud service needs to perform some specific transformation to the encrypted image, it makes a request to Darkroom by invoking a specific system call (through the Image Cloud Service). Through this system call, the service provides as input the image envelope, the encrypted IDs of the transformations to be invoked, and any additional parameters required by the transformation function. The system call issues a world switch to Darkroom, which performs three steps: 1) obtains the encryption key KI by decrypting it with the private part of the service key (KS^-), 2) validates the message integrity by recomputing the HMAC of the message `m` in that envelope, and 3) decrypts the image using KI. If all goes well, Darkroom executes the requested transformations and produces another envelope containing the resulting image encrypted with the symmetric key KI (which means that only the original message owner and Darkroom-enhanced servers will be able to decrypt the resulting image). To be able to trace the transformations that were applied to a given image, Darkroom builds a cryptographic-protected log which is included in the resulting envelope. The log contains a hash chain of the history of transformations applied to the image.

4. Image download: Finally, the last stage in the image transformation life cycle is to download the resulting image to the client and recover it. Since the client has access to the key KI, it can perform the same sequence of steps as Darkroom in order to validate the integrity of the image and decrypt it. In addition, since the envelope contains a history of transformations performed to the image, it is possible to verify that the received image has resulted from a sequence of transformations applied to the original image. The client sends a download request which contains the client identifier cID and the identifier of the image envelope to be downloaded, $niID$. The return is the envelope of the new image nI.

Summary

In this chapter we described the design of Darkroom, a secure data processing service for the cloud. We started by presenting the assumptions and threat model considered during the design of Darkroom. We then presented Darkroom's architecture by describing a possible execution flow for Darkroom and explained the different components which comprise Darkroom. We also explained the different architecture approaches possible for TrustZone-based projects and the reasoning behind choosing an intermediate architecture between a secure world operating system and a secure library of code.

The following sections were mainly focused on addressing the goal of end-to-end security by explaining a secure deployment protocol and the secure system bootstrap. We also address secure data storage of by developing a structure (blob) called *envelope* which contains the sensitive data to be processed and can only be opened by both the remote client and Darkroom. Finally, we address the goal of system applicability to secure image processing by supporting the deployment of new authenticated transformation functions to Darkroom at runtime. Next we present Chapter 4, which describes the implementation details behind the implemented Darkroom prototype.

4 Implementation

This chapter describes a prototype implementation of Darkroom, a TrustZone-based system for trusted data processing on the cloud. We implemented a prototype of Darkroom for the Freescale NXP i.MX53 Quick Start Board (QSB) and some of the implementation details described in this chapter are specific to this development board. Note that the current prototype does not implement all features described in Section 3, namely we have not implemented the secure system bootstrap, nor the dynamic loading of transformation functions.

Next, we describe the implementation challenges for our prototype (Section 4.1). Then we introduce the details of the Darkroom kernel (Section 4.2) and the normal world components (Section 4.3). We conclude this chapter by presenting the details on the Cryptographic Engine (Section 4.4) and the Image Processing Engine (Section 4.5).

4.1 Implementation Challenges

The implementation of Darkroom on a TrustZone-aware development board must consider several implementation challenges inherent to the development of the features and components described in our design. In particular, there is a fundamental concern in keeping a small Trusted Computing Base (TCB) for the software executing in the secure world. By achieving a small TCB for Darkroom we simplify the codebase and consequent execution flow of the system, which in turn decreases the chance of design flaws, code flaws and unnecessary services that can be used by attackers to compromise the system and disclose the sensitive image content we aim to protect. While maintaining a small TCB we also want to build a kernel flexible enough to feature memory management, threading and interrupt handling.

This is achievable by building Darkroom with a minimalistic approach, meaning it will only support the fundamental features necessary for secure data processing. Considering the workflow described in Section 3, where a secure image, packed inside an envelope, can go through the normal world OS, it becomes clear that Darkroom can easily avoid the implementation of unnecessary components such as network drivers and sockets. This can be supported by the untrusted rich OS running in the normal world. Additionally, support for commonly used development libraries such as `libc` allow for easier development of the user-space components which compose Darkroom, such as the Cryptographic Engine and the Image Processing Engine. It is also fundamental for the secure kernel to detect and trap `SMC` interrupts, which are used by the normal world environment to request a world switch. This means the secure kernel must be TrustZone-aware and correctly configure the monitor mode, secure memory regions and access to peripherals.

Additionally, Darkroom must interact with a full-featured operating system running in the normal world of the TrustZone-aware platform. However, most operating systems have no support for ARM TrustZone security extensions, for example, they do not offer an interface support for `SMC` interrupts triggered by their user applications. Offering an interface for `SMC` interrupts is important because it prepares the conditions for the world-switch which follows the `SMC` instruction. Besides the lack of an `SMC` interface support, rich operating systems are developed under the assumption of having full control over memory controllers, monitor mode and peripherals. Because Darkroom will be executing in the secure world and will restrict access to security critical components such as the memory controller, secure memory regions and peripherals, the rich OS running in the normal world OS must be modified to account for this configuration.

The following sections will describe the implementation strategies used to overcome the noted challenges. We start by introducing the Darkroom kernel which is the component responsible for the reduced trusted computing base, flexible runtime with support for memory management, threading and interrupt handling. This component is also responsible for configuring the monitor mode, secure memory regions and access to peripherals. We then describe the normal world operating system and modifications which allow the support of an `SMC` interface. We conclude this chapter by describing the remaining secure world components which consist of the Cryptographic Engine and Image Processing Engine.

4.2 Darkroom Kernel

The Darkroom kernel is a fundamental component of our system. It lives in the secure world and is responsible for memory management, thread execution, and context switch operations between worlds. To reduce the chance of code vulnerabilities and keep the kernel size small, we adopted the Genode [23] framework to build our secure world kernel. Genode is a framework for building special-purpose operating systems. It offers a collection of small building blocks, out of which we can compose a custom system to fit the needs of Darkroom's design. This framework includes building blocks such as kernels, device drivers, library support and protocol stacks.

4.2.1 Adapting a Genode Microkernel

Developing a microkernel from scratch to support the design requirements of Darkroom is a complex and time consuming task. In order achieve this a developer would have to become familiar with many of the low-level details of their particular platform. The developer would also have to become proficient at that platform's machine language, at building device drivers for all the devices needed, porting the necessary libraries (such as `libc`) and have an extensive knowledge of microkernel development.

For this reason, we have adapted a TrustZone-aware Genode microkernel named *base-hw* [24]. This is a custom microkernel tightly integrated with our development platform and with a code base of approximately 20 KLOC (thousand lines of code). Additionally, this *base-hw* microkernel also includes a client application responsible for managing a rich operating system running in the normal world and a secure world manager for leveraging the Multi-Master Multi-Memory Interface (M4IF). Because this client application acts similarly to a Virtual Machine Monitor (VMM), we call it TrustZone VMM (TZ VMM). Figure 4.1 shows the implemented components of this prototype. In a darker color we can see the

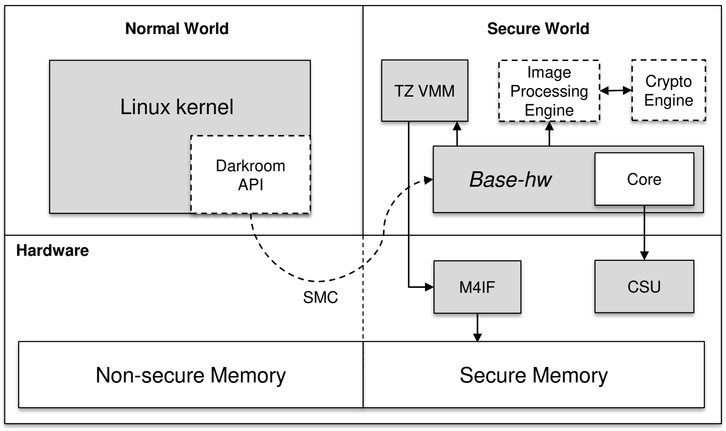

Figure 4.1: Implemented Darkroom components.

components adapted from the original *base-hw* as well as the adapted TZ VMM and normal world rich OS, which is a paravirtualized Linux kernel. In traced outline we have the components added to the system, such as the Darkroom API, the Image Processing Engine and the Cryptographic Engine, which support the Darkroom features.

The Cryptographic Engine (Section 4.4), Image Processing Engine (Section 4.5) and Darkroom API (Section 4.3.1) were added from scratch and are explained in detail below. The *base-hw* microkernel was adapted to support interaction between the newly added components of Darkroom. The TZ VMM and the paravirtualized Linux kernel were modified to support the inter-communication mechanism explain in Section 4.2.3.

Adapting the *base-hw* microkernel allowed us to focus on the problem we want to solve by offering a platform on which to build the Darkroom runtime. Taking this into account, this microkernel and the VMM client application do not solve all low-level problems associated with Darkroom, such as the inter-world communication and image processing, but it significantly decreased the difficulties of implementing a custom microkernel from scratch.

4.2.2 Supporting World-switch

For a secure world kernel to be TrustZone-aware it must be prepared to answer world-switch requests from the normal world OS. This means the secure kernel must configure the monitor mode to trap and handle the world-switch request. This configuration is done by leveraging the Secure Configuration Register (SCR), which allows the CPU to raise an exception in monitor mode instead of doing it in the corresponding exception mode of the normal and secure worlds. Because the monitor is a security critical component, as it works as a gatekeeper between the two worlds, the normal world entry to monitor mode is tightly controlled. The only way the normal world can request a world-switch is by issuing an interrupt, such as an external abort or an explicit call via the `SMC` instruction. On the other hand, the secure world can enter the monitor mode by directly writing to the Current Program Status Register (CPSR) and then access the SCR, through which the NS bit can be manipulated. Because Darkroom needs the world-switch to be explicitly triggered by a normal world cloud application, we adapted the

Exception	Vector Offset	Corresponding Mode
Reset	0x00	Supervisor Mode
Undef	0x04	Undef Mode / Monitor Mode
SWI	0x08	Supervisor Mode / Monitor Mode
Prefetch Abort	0x0C	Abort Mode / Monitor Mode
Data Abort	0x10	Abort Mode / Monitor Mode
Reserved	0x14	Reserved / Monitor Mode
IRQ	0x18	IRQ Mode / Monitor Mode
FIQ	0x1C	FIQ Mode / Monitor Mode

Table 4.1: Interrupt Vector Table of an ARM System

base-hw to configure the SCR to trap only the `SMC` instruction to the secure monitor.

After trapping the `SMC` triggered interrupt, the monitor's vector table is used to jump to a handling routine inside the secure world runtime. The monitor's vector table works similarly to an interrupt vector table (IVT), where specific addresses are used by the processor to assess which handling routine must be called for a determined interrupt. Table 4.1 shows the interrupts present on a typical IVT for an ARM processor platform. The same interrupts are supported in the monitor vector table. The flexible architecture of the monitor mode allows the routine responsible for handling the world switch request triggered by the `SMC` instruction to be implemented by the developer. This means that the developer is free to handle this situation in many different ways, but to support consistent world-switches there are a few steps which need to be implemented. The first is building a non-reentrant handling routine. This can be done by disabling the interrupts when the monitor mode is executed. This ensures the development of a more robust and secure handling routine which does not need to take into account other interrupts being triggered during its execution.

The second consideration to take into account is whether or not the switch from secure world to normal world must also be executed by the monitor software. The original *base-hw* microkernel already suported a basic context-switch mechanism in which the monitor software is not responsible for the switch from the secure world to the normal world. The secure world can directly manipulate the SCR and for this reason there is no need to execute the monitor code when a switch from secure world to normal world is desired. By choosing not to use the monitor code for this switch, the assembler world-switch routine is much simpler, since it does not have to consider both situations.

Finally, this routine saves the CPU state of the requesting world and restores the state of the world which will take control. Because the switch from secure to normal world is not considered by the monitor software, this routine only has to save the current CPU state as the non-secure world's state and restore the secure kernel context. To switch from the secure world to the non-secure domain the reverse operations are performed, only this time these operations are performed outside the monitor. This means the kernel context must be saved, stored in a secure memory region and then the saved normal world state must be restored and the world-switch executed by writing 1 to the NS bit in the SCR (bit 0). Figure 4.2 summarizes the context-switching process of Darkroom as explained above.

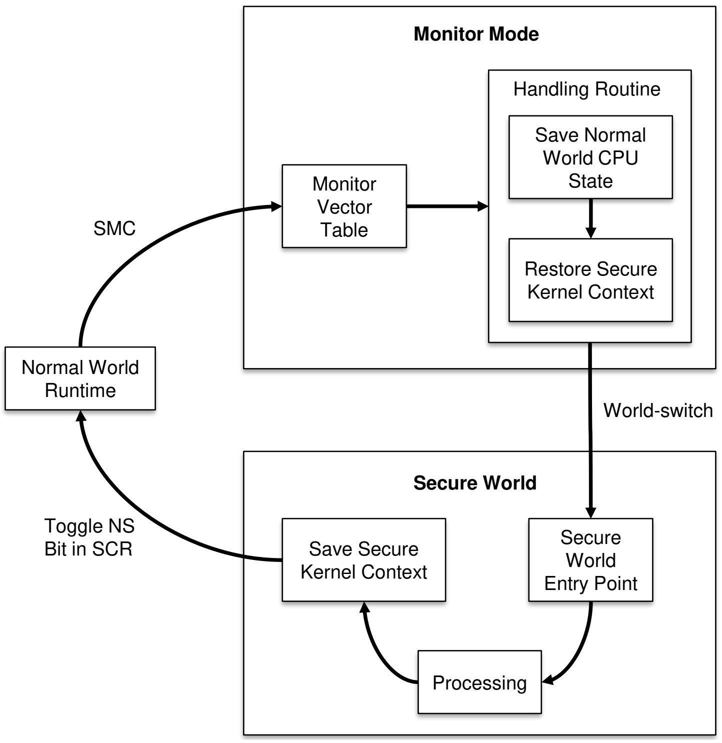

Figure 4.2: Context-switch process.

4.2.3 Inter-world Communication

As explained in the section above, when the normal world requests a context-switch, its CPU state is saved by the monitor software so it can be restored once the control is given back to the normal world. The component responsible for storing this normal world state is the TZ VMM, since it is the component which manages the normal world rich OS. And although the TZ VMM saves the processor state (registers and stack) when a `SMC` instruction is executed and restores this state when the control is given back to the normal world, it lacks a mechanism to identify the source of the `SMC`, a mechanism to send arguments from the normal world to the secure world and a mechanism to receive the image envelope from the normal world. We modified the original VMM to support these new features, which are essential to the inter-world communication mechanism.

The TZ VMM saves the CPU state of the normal world OS in such a way to make it available to the secure kernel to access. This means the VMM allows the secure world to access CPU registers, the CPSR, the stack pointer, the link register for every CPU mode, and a pointer to the start of the normal world's RAM. With this information we were able to develop a method for sending arguments from the normal world to the secure world. The normal world OS can request a world-switch and relay arguments to the secure world through the use of CPU registers (see Section 4.3.1 for more details on the Darkroom API). The normal world OS can push the current CPU registers to the normal world stack and proceed to write the desired arguments to these registers. The secure world can then access these registers via the saved CPU state of the normal world OS. After processing the image, the reverse operation is performed. The secure world can modify the saved CPU state of the normal world OS to send data to the untrusted domain. The normal world OS, after gaining control back from the secure kernel, can access its CPU registers to check for new data and can pop the previous register values from the stack to restore the CPU state to before requesting the world-switch.

Now that we can send arguments from the normal world to the secure world we can use it to (1) identify the source of the world-switch request and (2) receive large amounts of data from the normal world OS. Identifying the source of the world-switch request is important in order to achieve a flexible system. This means that, by developing a way to identify the source of the world-switch request we can support different services running in the secure world. We can use one identifier for the secure image processing service and other identifiers for additional services supported in the future. This makes the Darkroom kernel a flexible and generic platform capable of supporting several secure processing services. In the context of this prototype, we have created a simple identifier for the image processing service and the Darkroom API writes this identifier in the register zero (`r0`) before requesting the world-switch. The secure kernel, after gaining control over the execution, checks this register and calls the appropriate service. This service, which for this prototype is solely composed of the image processing component, is then responsible for handling the remainder of the execution. Particularly, this service becomes responsible for accessing the CPU state of the normal world OS in order to read the necessary arguments for its correct execution.

Another important functionality we want to add to TZ VMM is the ability to receive large amounts of data from the normal world OS. But the CPU registers can only store four byte words, which can represent integers, addresses, characters, etc. This means that, to support large amounts of data communication between both worlds, we have to employ the use of CPU registers to send metadata to the secure world, which the kernel can then use to recover the intended data. As described in Darkroom's design, the inter-world communication of images is done through the use of a shared memory strategy. To this end, the metadata we need to send is metadata which allows the secure world to access the shared memory region where the image data is stored. In order for the secure world to access a shared memory region, it needs the pointer to the start of that shared memory region and the size of the image being sent. This means the metadata sent to the secure world via CPU register one (`r1`) and two (`r2`) are the physical address corresponding to the start and size of the allocated memory region, respectively.

4.2.4 Boot Time Process and System Configuration

At boot time and before launching the normal world operating system, we have to configure the TrustZone components responsible for securing critical devices, the monitor, interrupts, and the secure memory regions for Darkroom. The first configuration executed by the secure world kernel is configuring the i.MX53 QSB custom TrustZone protection controller called Central Security Unit (CSU). This controller allows the assignment of device groups to the secure or normal world by using the corresponding configuration bits. In Darkroom, some devices are protected using the CSU. These include the GPIO, the TVE (TV encoder), I2C (provides serial interface for controlling peripheral devices), the IPU, GPU and Video Processing Unit (VPU), although these are not used by Darkroom nor the normal world OS. The Universal Asynchronous Receiver/Transmitter (UART) used for serial communication is considered unsecure in order for it to be used by both the secure and normal worlds for testing purposes, namely for sending commands via the serial console. The CSU responds to access violations by synchronously yielding control to the exception handler, meaning that the offending instruction can be determined and

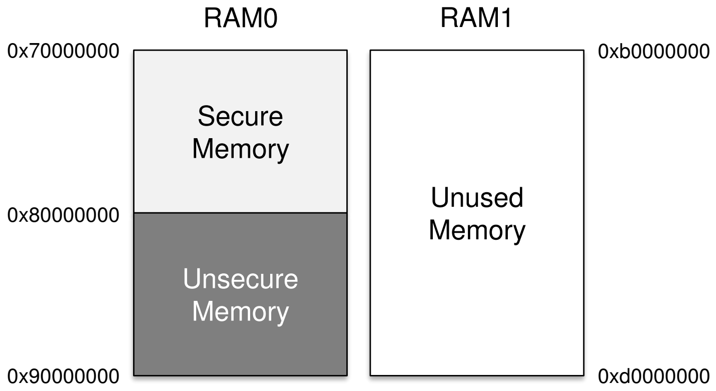

Figure 4.3: Memory layout.

handled appropriately, either via software emulation or by ignoring the request.

Another configuration performed at boot time is the configuration of the Secure Configuration Register (SCR) which is the register that contains the NS bit used to switch execution modes from the secure to the normal world and vice versa. The original *base-hw* microkernel implements functions which allow the secure kernel to explicitly modify this NS bit via the SCR and also functions which allow the secure world to explicitly set the address of the monitor table and interrupt table. Both these functions are used during the boot time to configure the monitor and interrupts. Lastly, the i.MX53 QSB supports a watermarking mechanism, to protect memory regions from unauthorized accesses, via the Multi-Master Multi-Memory Interface (M4IF), which is part of the DDR memory controller.

The M4IF enables the masking of DDR RAM resources such that particular ranges of memory can be protected and reserved for the secure world. The i.MX53 QSB has two configurable memory banks with 512 MBs each, summing up to a total memory of 1 GB. The M4IF can use its watermarking mechanism to protect up to 256 MBs of memory in each configurable region [14]. At boot time Darkroom leverages M4IF to protect 256 MBs of the first memory bank which become reserved for the use of the secure runtime. The remaining 256 MBs of memory from the first memory bank are used for the normal world OS. This means the system only uses the first memory bank of the i.MX53 QSB board. This was an implementation choice done by Genode developers when implementing the *base-hw* microkernel and the TZVMM. Since we have not noticed this reduced amount of memory to be a limitation for our Darkroom prototype, this remained unchanged. Figure 4.3 shows the memory organization of Darkroom for the i.MX53 QSB development board's available RAM. All memory allocations done inside Darkroom are protected via the watermarking mechanism, thus protecting the memory of the secure world runtime.

4.3 Normal World Components

As explained in the section above, Darkroom kernel leverages a TrustZone Virtual Machine Monitor to manage a normal world rich operating system. This operating system houses the application which will use the secure image processing service offered by Darkroom through the use of a Darkroom API. The normal world runs a paravirtualized Linux kernel modified to be executed along side Darkroom. We chose Linux as the normal world OS because these systems represent the majority of servers deployed around the world. Additionally, this particular paravirtualized Linux kernel, version 2.6.35, was chosen because it was modified to run along side Genode's *base-hw* microkernel.

However, this modified kernel does not support the features necessary for the use of Darkroom's image processing service, namely it lacks support for a world-switch request by user-level applications and an inter-world communication mechanism for transferring large amounts of data to the secure world. For this reason, we modified this paravirtualized version of the Linux kernel to reflect these changes and implemented a user-level application to work as intermediate between the remote client and Darkroom's image processing service. This section describes the implementation details of these two components.

4.3.1 The Use of Darkroom API

In order for untrusted applications to use Darkroom for secure image processing, the rich OS needs to support a mechanism for these applications to request a world-switch. One of the most explicit interfaces between user-level applications and the Linux kernel are system calls. System calls are the way in which a computer program can request a service from the OS, in this case the service is a world-switch. To this end, we modified the Linux kernel to support the new system call named `smc`. Among the arguments, this system call receives a character buffer holding the envelope binary data, an integer corresponding to the size of the envelope and an integer identifying the transformation requested. The data inside the envelope holding buffer must then be copied to the shared memory region for inter-world communication, and the size of this data as well as the identifier of the transformation are sent to the secure world via a CPU register.

For our inter-world communication mechanism, the normal world must allocate the necessary memory region to house the envelope to be sent to Darkroom. The normal world must allocate this shared memory region so that both domains can access it. This is because the memory allocated by the secure world is inaccessible to the normal world components. Additionally, by allowing the normal world to allocate the necessary memory regions we avoid unnecessary messaging between the normal and secure worlds for metadata exchange. This metadata exchange would be necessary because the secure world would have to know in advance how much memory needs to be allocated. To allocate the necessary memory region, the system call `smc` receives the size of the image envelope to be sent and, to populate this memory region, it receives the image envelope data as a character buffer. The system call implementation then uses these arguments to allocate a memory region for the corresponding envelope.

There are several implementation options to allocate a shared memory region in the normal world. We started by employing the use of Direct Memory Access (DMA) memory. The use of DMA memory has advantages when writing and reading large amounts of data to that memory region, because the CPU is not fully occupied for the entire duration of the read or write operations, since the DMA controller is the one responsible for these operations. This leaves the CPU free for processing other operations while the transfer is in progress, but allocating DMA memory is expensive when compared to other commonly used memory management operations such as `malloc`, `realloc`, `calloc` or `free`.

Our second attempt was still using the DMA, but instead of allocating the memory region every time an `smc` system call is executed we would only allocate this memory region during the first `smc` call. However, this approach raises several problems, such as how much memory to allocate in order to support all future image transformation requests to the secure world, and by allocating a large DMA region we may be using the system's memory inefficiently, since the memory region will, most of the time, be allocated and not in use. Additionally, our tests concluded that the performance advantage of allocating once instead of allocating every time the `smc` system call is executed is negligible for the envelope sizes tested. For this reason we discarded this strategy. Similarly, the use of `kmalloc` in the beginning instead of using it for every world-switch request shares the same problems as before, although the use of `kmalloc` proved to be more efficient for our particular use cases when compared to DMA. For this reason, we chose the use of the `kmalloc` operation (`malloc` for the kernel) to allocate the shared memory region between the normal world OS and the secure kernel every time a new transformation is requested.

After allocating the shared memory region, the contents of the envelope are copied to this memory region via the `memcpy` function. The following procedure is to translate the virtual address corresponding to that memory position, known to us through the return of `kmalloc`, to a physical address, so the secure kernel can access it. This translation is a necessity because the virtual address (or logical address) represents the set of ranges the operating system makes available to that application process and these addresses have no real correspondence to the real (physical) address. In reality, under a 32 bit OS such as the Linux kernel we are using, it is quite common for multiple processes to use the same virtual addresses. To proceed with this translation, the system call must execute an OS function called `virt_to_phys`. This function receives a virtual address represented by a pointer, in this case a pointer to the memory region allocated with `kmalloc`, and translates it to the physical address. This physical address can then be sent to the secure world, so the secure kernel can access the shared memory region.

Before triggering the `SMC` instruction, the physical address of the shared memory region is written to register one (`r1`), the size of the image envelope is written to register two (`r2`) and, finally, the transformation identifier is written to register three (`r3`). This allows the secure world to access the shared memory region via the physical address, read that memory region according to the size and process the image transformation identified. Darkroom must then copy the contents of the shared memory region to a secure memory region so it can be processed and avoid exposing the decrypted data to normal world memory regions. After processing, the secure kernel copies the resulting envelope back to the shared memory region and returns control to the normal world OS. The system call, now with execution control again, can copy the envelope from the shared memory region to the application buffer and free the shared memory. Once the system call returns, the application can read the results from the input/output buffer and write the result to a persistent file.

4.3.2 User-level Server Application

To communicate with a remote client who wishes to use Darkroom, we implemented a normal world application responsible for the network communication (upload and download of image envelopes) and storing the envelope containing the image to be processed in the file system of the normal world OS. The purpose of this application is to simulate a realistic server app, such as Facebook or Instagram, which can leverage the security features introduced by Darkroom in secure image processing.

Additionally, we have implemented an extension to this server application which allows us to process the image transformations requested to Darkroom inside this application. This extension is meant to be

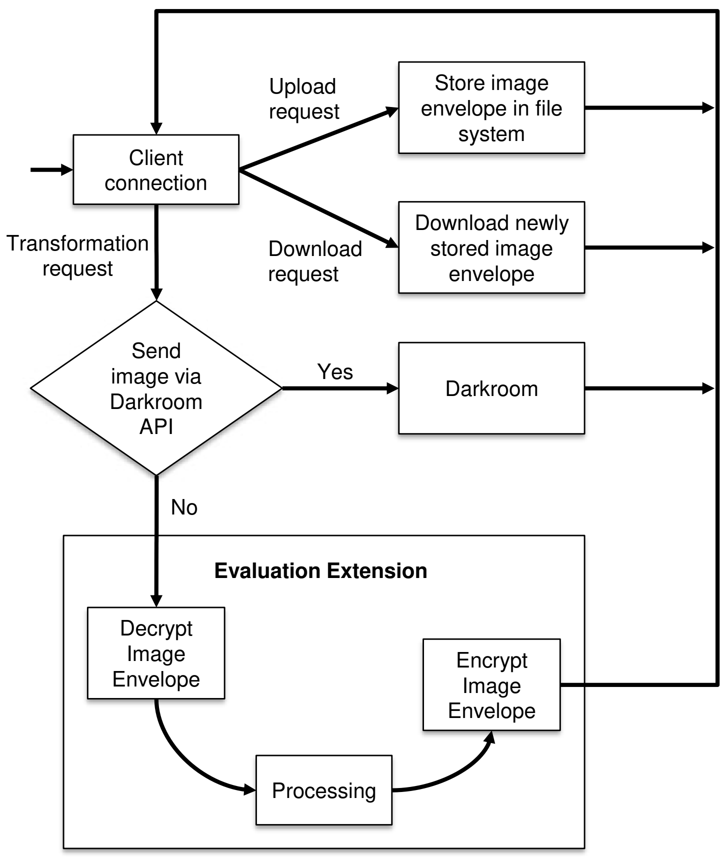

Figure 4.4: User-level server application workflow.

used for evaluation purposes discussed in Chapter 5. When we want to test the transformation on the normal world, the envelope received by the application is sent to the extension, which is provided with the same cryptographic keys as Darkroom in order to open the envelope, and the image is decrypted, processed and encrypted again into a new envelope. It is obvious that this extension defeats the goal of Darkroom, but is only implemented here for evaluation purposes and would not be implemented in a realistic, deploy-oriented environment.

This is a simple server application written using the C language and leveraging Unix sockets for networking. Figure 4.4 describes the execution flow of this user-level server application responsible for relaying secure image transformation requests from a remote client to the secure environment provided by Darkroom or to the extension developed for testing purposes. This application was compiled using an ARM cross compiler on an Ubuntu laptop and deployed to the normal world Linux kernel by including it in the ramdisk (`initrd`).

4.4 Cryptographic Engine

To support the cryptographic operations described in Darkroom's design, we implemented a software component to run on top of the Darkroom kernel. This component supports the management of both symmetric and asymmetric cryptographic keys and implements core cryptographic algorithms. The protocols supported are AES-128 for symmetric cryptography, RSA for asymmetric cryptography and HMAC with MD5 as an hashing function. During the implementation process of the prototype we faced some difficulties and challenges discussed in this section.

The development board used for our implementation of Darkroom has a hardware component called Symmetric/Asymmetric Hashing and Random Accelerator (SAHARA). This hardware component is a security coprocessor which implements symmetric encryption algorithms, such as AES, DES, 3DES, RC4 and C2, public key algorithms, such as RSA and ECC, hashing algorithms, such as MD5, SHA-1, SHA-224 and SHA-256, and a hardware true random number generator. Because we want to keep the software trusted computing base small and we trust the correctness of the hardware, using SAHARA would allow us to support our cryptographic features without having to rely on software implementations and without the challenges of implementing these algorithms on an embedded platform. The problem is that SAHARA is a proprietary hardware coprocessor, and we were not given access to the necessary documentation needed to leverage this cryptographic hardware technology. Instead we had to resort to software implementations of cryptographic algorithms.

Our first attempt was looking at Genode supported cryptographic libraries. Genode OS frameworks allows developers to add official Genode ports of popular libraries, and one example of these libraries is OpenSSL [1]. OpenSSL is an open-source, full-featured toolkit for the Transport Layer Security (TLS), Secure Sockets Layer (SSL) protocols and is also a general-purpose cryptography library. Another cryptographic port developed for Genode was a port for *mbed TLS* library [32]. The *mbed TLS* is a cryptographic library developed by ARM and adapted from the former *polarSSL* for embedded devices. But these ports were not developed to be executed on top of the `base-hw` microkernel and as such they do not work on Darkroom's kernel, which is adapted from the aforementioned microkernel. Another problem with using Genode ports is that we have to include all the features supported by the ported library, even features we do not use in Darkroom. This increases the TCB of Darkroom, and with a library such as OpenSSL, which has approximately 500 KLOC [2], this increase has a substantial impact on the size of Darkroom's TCB.

As an alternative, we tried to manually adapt the *mbed TLS* library. The disadvantage of this library is that, although it is coded using the ANSI C implementation and with a minimalistic coding footprint, it still relies on many Linux specific system calls for file management, input/output and for entropy gathering. This means that, in order to adapt these algorithms to work inside Darkroom, which does not support these features, we had to modify the algorithms to rely on different strategies. One example of this is modifying the way the algorithm reads a cryptographic key from a file to reading a key from a memory buffer. This change allowed us to use the AES-128 implementation of *mbed TLS*. But for other features, such as the entropy gathering functions, modifying the implemented support is difficult and because Darkroom does not support any trustworthy alternatives, we decided to search a simpler alternative implementation for both RSA and HMAC.

Finally, we implemented all necessary cryptographic functions by adapting the popular *mbed TLS* library for the AES-128 protocol and simpler implementations for the remaining RSA and HMAC algorithms. On the other hand, for a deployment environment, the system would be better-off using the

[1]`https://www.openssl.org/`

[2]`https://www.openhub.net/p/openssl/analyses/latest/languages_summary`

```
import marshal

def transformation(image):
    new_pixels = []
    for pixel in image.pixels:
        color = pixel;
        alpha = (color >> 24) & 0xff
        red   = (color >> 16) & 0xff
        green = (color >>  8) & 0xff
        blue  =  color & 0xff
        lum = (red * 0.299 + green * 0.587 + blue * 0.114)
        new_pixel = (alpha<<24) | (lum<<16) | (lum<<8) | lum
        new_pixels.append(new_pixel)
    image.pixels = new_pixels

code_string = marshal.dumps(transformation.func_code)
```

Figure 4.5: Sample of Python client code for serializing a transformation function.

```
import marshal, types

image = imageFromEnvelope()
code  = marshal.loads(code_string)
func  = types.FunctionType(code, globals(), "transformation")

func(image) # processing of the image
```

Figure 4.6: Sample of Python server code for reconstructing a transformation function.

SAHARA hardware coprocessor. The use of this coprocessor would not only allow for a reduced TCB, but also better performance. In addition to the algorithms, this component also has the responsibility to implement key management. In this prototype the keys are provisioned to the system when Darkroom is flashed to the development board. This means the keys are in the binary loaded to the hardware.

4.5 Image Transformation Functions

The Image Processing Engine runs on top of the Darkroom kernel and manages the transformation functions implemented in the system. Note that, in the current version of this prototype, the dynamic loading feature, which allows a cloud administrator to add trustworthy image transformations to Darkroom, is not implemented because supporting dynamic loading of binary functions in low-level languages is complex. The transformation of a C project into an executable binary program relies on two stages: compiling and linking.

The compiler processes the source code files and creates an `object` file, but this does not create an executable. This means the user cannot use this file to run the program it describes. What the compiler does is it creates the machine language instructions that correspond to the source code file that was compiled. But to run these files a developer needs to turn this object file into an executable. This is where the linker is introduced.

Name	Description
T1	Grey-scale transformation
T2	Color invert transformation
T3	Color swap transformation
T4	90 degree rotation transformation
T5	180 degree rotation transformation
T6	Mirror transformation

Table 4.2: Description of the transformation functions implemented.

```
for (i = 0; i < length(oldp); i++) {
    color = oldp[i];
    alpha = (color >> 24) & 0xff;
    red   = (color >> 16) & 0xff;
    green = (color >>  8) & 0xff;
    blue  =  color & 0xff;
    lum = (red * 0.299 + green * 0.587 + blue * 0.114);
    newp[i] = (alpha << 24) | (lum << 16) | (lum << 8) | lum;
}
```

Figure 4.7: Sample code of gray-scale transformation function.

The linker is responsible for creating a single executable file from multiple object files. Unlike the compiler, which looks at each file and translates the source code to machine language, the linker looks at every file and links the functions implemented on the object files to each other. It is in this step that we get undefined function errors while compiling a C language project. This means the linker is responsible for making sure that every function in an object file knows where the machine language for the functions they call is. And because this association is done at compile time, supporting dynamic loading of functions is much harder to do in compiled languages then in interpreted languages, where this process can be performed in runtime.

Because Darkroom is implemented using compiled languages, the support for dynamic loaded functions is complex. On the other hand, if Darkroom could support an interpreter for a language such as the Python programming language, then the support for dynamic loading of image transformation functions would be possible. As a way to understand how this process would work we implemented a Python server which can successfully receive a binary object representing a Python function and interpret it at runtime, thus effectively supporting dynamic loading of functions. This works by leveraging a client Python application which can serialize a function's bytecode which the server will reconstruct in order to call it. Figure 4.5 shows a simple implementation of the client logic responsible for serializing the function and Figure 4.6 shows the server logic responsible for reconstructing it and call the newly reconstructed function. This code uses the `marshal` Python module for serializing and loading the function and the `types` Python module for reconstructing the transformation function.

In our current prototype, we implemented a small set of simple transformation functions just to demonstrate the feasibility of our approach. In a real world setting, more sophisticated functions could be developed in order to serve the needs of external services such as image managing websites, social networks and personal record management services. The transformation functions currently implemented in our system are listed in Table 4.2. All transformations offered by this component were implemented from scratch without having to rely on any image library. Figure 4.7 provides the sample code of a transformation function to change the color palette of an image to gray scale.

The choice of implementing these image transformations without using an image library relies on the fact that, similarly to cryptographic libraries, image libraries such as libpng, libjpeg and libtiff depend on large trusted computing bases and implementing these in Darkroom would have a significant impact on the size of Darkroom's TCB. The alternative to implementing these libraries is to process the pixels of these images directly. This means the image envelope sent to the secure world must not contain the image in its encoded, compressed format but rather the uncompressed image pixel data. To do this, the client application, described above, is responsible for decoding the image and retrieving the pixel data to be sent to the Darkroom server.

Summary

We implemented a prototype of Darkroom for a TrustZone-aware development board with the fundamental concern of keeping a small Trusted Computing Base (TCB), which essentially consists of the components that live in the secure world: Darkroom Kernel, Cryptography Engine, and Image Processing Engine. We described the Darkroom kernel as fundamental component of our system. It lives in the secure world and is responsible for memory management, thread execution, and context switch operations between worlds. To reduce the chance of code vulnerabilities and keep the kernel size small, we adopted the Genode framework to build our secure world kernel. In addition, Genode implements a Virtual Machine Monitor (VMM) which can manage a paravirtualized full-featured operating system running in the normal world.

To communicate between worlds, we adopt a shared memory strategy where the normal world operating system is responsible for allocating the necessary memory region for sending an image envelope to the secure world. Implemented in the normal world OS is a new system call called `smc` which is responsible for preparing the system for a world switch and allocating the shared memory region. The Cryptographic Engine runs on top of the secure world kernel and must support the management of both symmetric and asymmetric cryptographic keys and implement core cryptographic algorithms, such as AES, RSA and HMAC. The Image Processing Engine runs on top of the Darkroom kernel and manages the transformation functions loaded into the system. In our current prototype, we implemented a small set of simple transformation functions just to demonstrate the feasibility of our approach. All transformations offered by this component were implemented from scratch without having to rely on any image library. In the next chapter we present the experimental evaluation made using this prototype.

5 Evaluation

In order to evaluate our prototype we have decided to measure and study different aspects of this implementation. In this chapter we start by describing the methodology used throughout our experiments (Section 5.1), then we analyse the performance of image transformation functions (Section 5.2). We then evaluate the overhead of the context-switch mechanism (Section 5.3), the image envelope structure (Section 5.4) and the trusted computing base size (Section 5.5). Finally, we discuss the implemented use case client application (Section 5.6), the first implementation attempts (Section 5.7) and conclude with some security considerations and limitations (Section 5.8).

5.1 Methodology

Our evaluation testbed consisted of an i.MX53 Quick Start Board, featuring a 1 GHz ARM Cortex-A8 Processor, and 1 GB of DDR3 RAM memory. The board executed our system from a mini SD card, which was flashed with the modified Genode and Linux versions. For each experiment, we report a mean of 50 runs and the corresponding standard deviation. We also stress that the results recorded in these experiments were collected after both the normal and secure world components were compiled with the O3 optimization flag.

Our testbed also includes an LG Nexus 4 device, featuring a Quad-core 1.5 GHz Krait processor, and 2 GB of DDR2 RAM memory. This device was used as a host device for the Android application implemented as a client for Darkroom. Additionally, we have also measured the experiments in this device using the same methodology described above, where for each experiment we report a mean of 50 runs and the corresponding standard deviation.

To measure the execution time of a specific operation executed in the development board, either on the normal or secure worlds, we implemented two functions which leverage the `gettimeofday` system call to start and end a timer. On Android we measure the execution time using the same approach but resorting to the use of the `java.lang.System.currentTimeMillis()` method.

5.2 Performance of Image Transformation Functions

The goal of this evaluation is to measure the performance of Darkroom when executing image transformation functions. In order to have a baseline for our image processing service, we measure the execution time of processing images in the normal world and compare these times with the execution time of processing the same image transformation functions using Darkroom. By comparing these

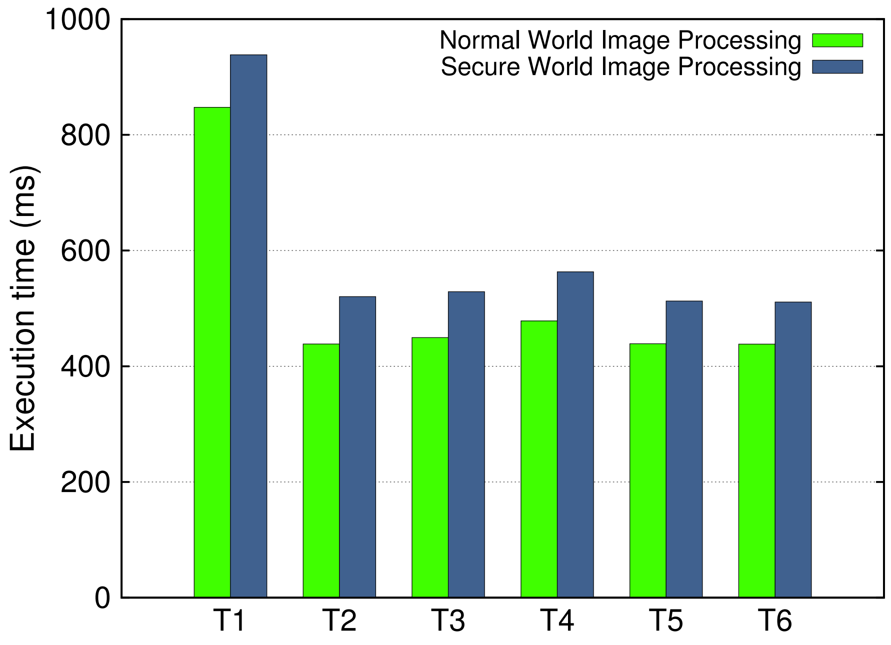

Figure 5.1: Execution time of all transformations.

execution times we can conclude if there is a penalty for using Darkroom features, such as the context-switch, and more specifically of components such as the *base-hw* microkernel.

The execution flow for the secure world case is as follows: we start to measure the execution time as soon as the normal world application receives an image transformation request from the remote client. The normal world application then reads the image envelope from the normal world file system and calls the `smc` system call to proceed with the world-switch request. The secure kernel receives the image envelope, executes the image transformation and sends the result to the normal world. The normal world reads the new image envelope and writes it to the normal world file system. At this point we stop measuring the execution time. The normal world execution-flow is very similar with the difference that the image envelope is never sent to the secure world. In turn, the normal world opens the envelope, executes the image transformation and creates the resulting envelope.

Table 4.2 presents the six different image transformations supported by our prototype, as well as the names we picked to identify them during our result analysis. To get a clear and visual perspective of the performance costs of these transformations, we chose to measure their execution time over a single image. The image we picked has a 1024x1024 pixel resolution, not only because it is a resolution supported by many of today's mobile devices, but also because it is a common resolution for network computing devices. Figure 5.1 shows the execution times of all these transformations for the aforementioned image.

By analysing this graph, which depicts the execution time in milliseconds in the Y-axis and the corresponding transformation in the X-axis, we can observe a constant penalty for the secure world execution flow (dark-shaded bar) when compared to the same transformation executed in the normal world (ligh-shaded bar). This overhead is associated to the context-switch, the implementation of *base-hw*'s memory manager, scheduler and compiler optimizations. As an example, the memory manager and scheduler implementations of the *base-hw* microkernel are much simpler than that of the Linux

Image	T1	T2	T3	T4	T5	T6	Mean	STDEV	Mean (%)
128x128	2	1.12	0.98	0.94	1.02	0.98	0.17	0.41	15.7%
256x256	7	4.94	4.68	4.92	4.42	4.56	5.01	0.96	16.1%
512x512	23.03	19.86	19.5	19.86	20.78	17.84	20.15	1.71	16.4%
1024x1024	90.82	82.04	79.32	84.92	73.92	72.76	80.63	6.83	16.4%

Table 5.1: Overhead (in ms) of every transformation for each image.

kernel. The scheduler running inside the secure world is, unlike that of the Linux kernel, a simple round-robin implementation, and this may have performance implications on the execution time of these transformations which go beyond the scope of this dissertation.

Besides being difficult to assess, from the analysis of this graph, what is the source of the overhead, we can observe that it is constant across all transformations, even when the transformation has an execution time above that of other transformations. As an example, the grey-scale transformation (T1) is by far the most expensive because, unlike the remaining transformations represented in this evaluation, it consists of a loop with complexity *O(N $\times$ M)*, where *N* is the width and *M* is the height of the image. The remaining transformations are all simple loops, with complexity *O(N)*.

To understand whether this penalty is associated to the size of the image and not the transformation's complexity, we measured the same execution flow for images with 128x128, 256x256 and 512x512 pixel resolutions. For consistency reasons, these small sized-images are the result of downgrading the original 1024x1024 picture resolution to those three lower resolutions. Table 5.1 shows the evaluation results, in milliseconds, of the overhead penalty for the remaining images. The overhead shown in this table is the difference between executing an image transformation in the normal world and executing the same transformation using Darkroom.

As we can observe, the penalty also remains constant across all transformations for the newly tested images. However, across different images we observe an increase of the absolute value of the overhead. This proves that the observed overhead is a consequence of the size of the image being processed by Darkroom. Although we observe an increase in the absolute value of the overhead when larger images are processed, we also observe that the mean percentage of the overhead remains the same for all images (approximately 16% overhead). Considering the added security features supported by Darkroom, this overhead is negligible, since for the larger 1024x1024 resolution image we observe an overhead of less than 100 milliseconds (unnoticeable by the user).

5.3 Context-switch Overhead

After analysing the performance overhead of using the secure world for image processing, we evaluate the overhead of switching from the normal world Linux kernel to the secure world runtime. This allows us to better understand how much of the overhead measured in the section above is caused by the context-switch. To evaluate the context-switch overhead, we measured the execution time of transformation T1 for different image resolutions. We focused on T1 since it is the most computationally demanding of all transformations implemented in the prototype.

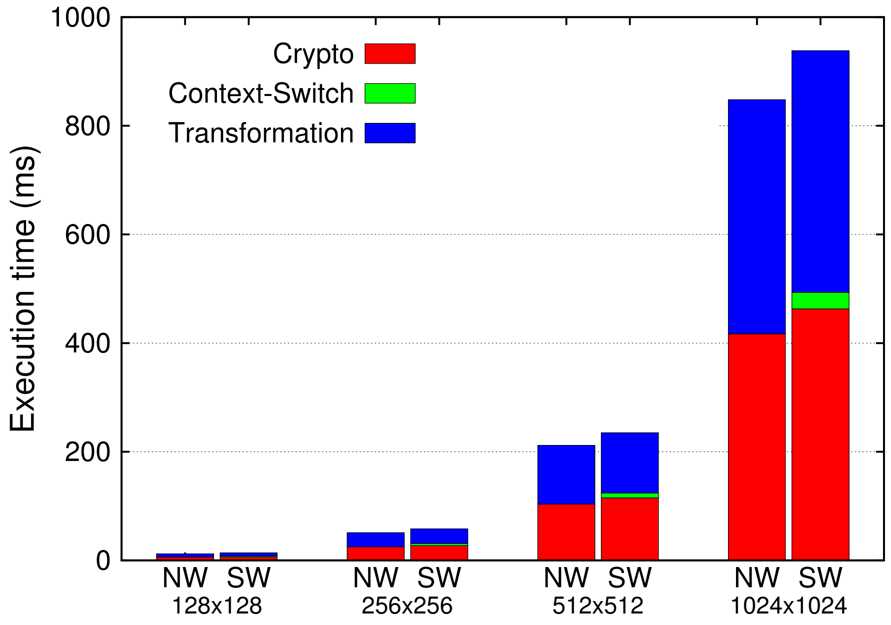

Figure 5.2: Execution time of the grey-scale transformation.

Figure 5.2 shows the impact of T1 on a 128x128, 256x256, 512x512, as well as on the original 1024x1024 picture. The execution time is divided into three parts: the transformation itself, the context-switch, and cryptographic operations. In this graph the Y-axis represents the execution time in milliseconds of the same execution-flow described before. On the X-axis we can see the different sized-images used for this evaluation. The total overhead shown by this figure is the same as that shown in the previous section, but this graph can represent the three main stages of the work-flow and their execution time. The bottom rectangle of each bar represents the execution time of the cryptographic operations performed in Darkroom. The middle rectangle represents the context-switch execution time and the top rectangle represents the execution time of the image processing stage, the image transformation itself.

Note that the context-switching mechanism includes the execution of our custom syscall from the normal world application, allocation of memory for the shared buffer, translation of the virtual memory address to a physical address, and the call of the `SMC` instruction, which triggers the world-switch. As such, the context-switch time described by this figure is highly influenced by the size of the image being sent to the secure world. This is because with a larger image the normal world must allocate a larger shared memory region and copy more data to and from this shared memory region. And this is clearly visible in the figure, where larger images display a more noticeable overhead than that of smaller images, to the point where in smaller images this overhead is no longer visible in the figure.

Another important aspect to note is that the cryptographic stage in the secure world also has some implementation differences which affect the execution time of the secure world execution flow. In order to keep the data protected from the normal world, the secure world receives the envelope via shared memory and allocates a secure memory buffer for the envelope. It then allocates another secure memory buffer to house the image data decrypted from the envelope and another to support the image resulting from the secure processing. This means that the secure world has two additional buffer allocations and two additional data copying operations, which allied to the less optimized memory management imple-

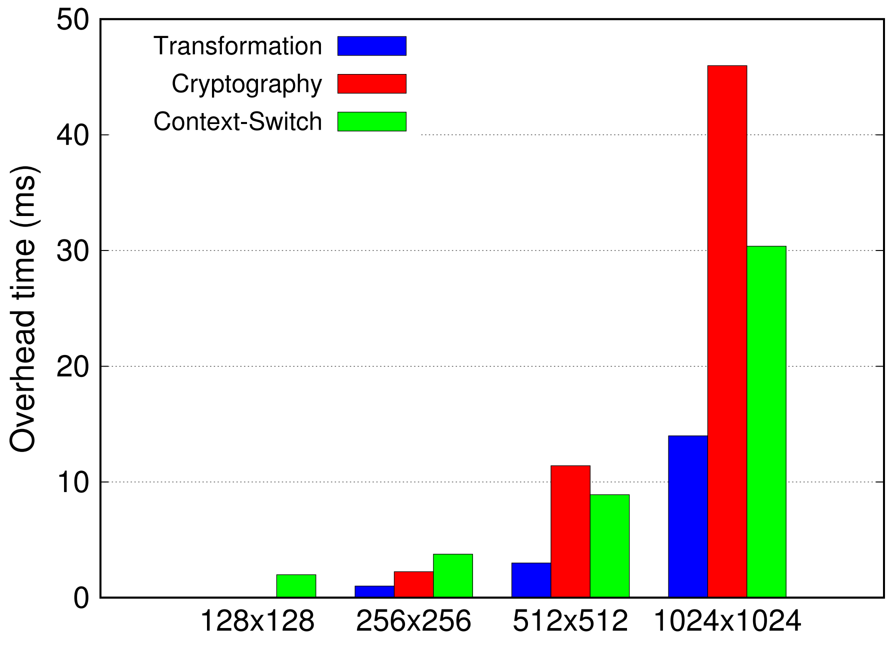

Figure 5.3: Execution time overhead of each stage.

mentation of the *base-hw* microkernel, can have a noticeable impact on the performance of the system. It is this impact that we observe in Figure 5.2.

In order to evaluate the impact of each of these three stages on the overall performance, Figure 5.3 shows in detail the additional execution time spent by each stage for every image tested. By analysing this figure, where in the Y-axis we have the overhead time in milliseconds for each stage, and the X-axis represents the images tested for this evaluation, we can observe that the overhead of the image processing stage is negligible (less than 15 milliseconds for the largest image) and is associated to the memory management implementation of *base-hw* microkernel. This has to be the case, since the code implemented in the normal world and secure world is the same and were both compiled with the highest level of compiler optimizations (O3).

Additionally, during our evaluation we noticed a high discrepancy in the execution times of the image processing stages, which lead us to analyse the binary code of the compiled objects of the normal world application and Darkroom. From this analysis we concluded that the normal world application was using the Floating Point Unit (FPU) for float-point operations and was getting a significant advantage when performing image transformations. We fixed this discrepancy by compiling both the secure world and normal world application with soft floats, i.e., to emulate the FPU operations via software. This is because Genode's `libc` implementation was not compiled to support this board's FPU. Note that the use of soft floats has an overall impact on the performance of the system, but in this evaluation we are more concerned with the differences between the performance of a strictly normal world implementation and our prototype, instead of being concerned with the absolute execution times. This, allied to the fact that both the normal world application and secure world are compiled with the same optimizations and are running the same code, reinforces the fact that the negligible overhead of the transformation stage must be associated with the less optimized memory manager of Genode's *base-hw*.

The cryptographic stage, as explained before, has two additional memory allocations and two ad-

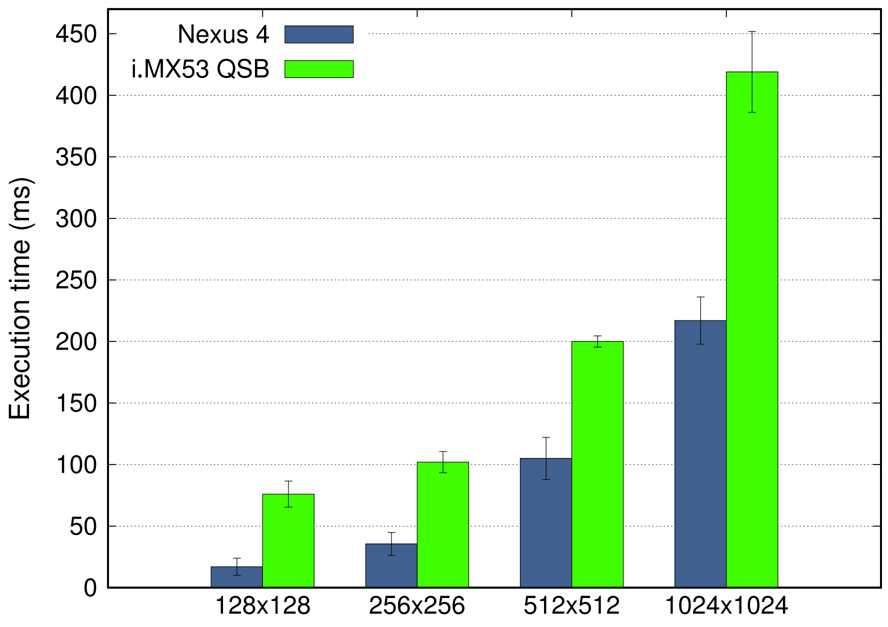

Figure 5.4: Execution time of the image decoding process at the client and at the server.

ditional copying operations, while the context-switching stage has only one memory allocation and two copying operations. For this reason we expected the cryptographic stage to demonstrate a higher overhead than that of the context-switching mechanism. On one hand, Figure 5.3 reinforces this idea by showing a larger overhead for the cryptographic stage on larger images. On the other hand, for smaller images the cryptographic stage is surpassed by the context-switching stage. This happens because the context-switch mechanism performs memory operations which are dependent from the size of the image, but also performs additional operations which are independent from the image size. Operations such as calling the `smc` system call, translating the shared buffer's virtual address and performing the world-switch itself have a constant cost independent of the size of the image. This constant cost is what we witness in the measurements for smaller images.

5.4 Image Envelope

In this section, we start by studying the impact of performing image decoding on the client instead of performing it on the secure world runtime, i.e., to send the raw pixel data (e.g.: bitmap image) to Darkroom instead of sending the original image format (e.g.: jpeg), which might be compressed. We then compare the size of encoded images with the size of these same images decoded into raw pixel data. This allows us to evaluate the impact of using uncompressed bitmap images in Darkroom instead of the original compressed format. We then study the overhead of using the image envelope structure, such as the size overhead, to securely send and store the sensitive images.

5.4.1 Image Decoding Performance

As explained in Chapter 4, the secure world runtime expects the image envelope to contain the pixel data of the sensitive image, instead of containing the encoded image data which might be subject to compression techniques. This allows us to avoid porting an existing image managing library for each supported format and thus maintain a smaller trusted computing base. But in order to understand whether this has a significant impact on the performance of the system, we measured the time it takes to decode an image on both the client Android application and on the server platform.

Figure 5.4 shows the execution time in milliseconds, represented in the Y-axis, of the decoding process of different-sized images, represented in the X-axis, for two different platforms, the Nexus 4 on the left bar and the i.MX53 QSB on the right bar. We can observe that the amount of time needed to decode images on the development board is much larger that the time needed by the mobile device to perform the same operation. This is because the development board used for our prototype has a less powerful processor than the device used to run the client application. In this particular instance, decoding the image on the client side yields a better result for the overall performance of the system. Although for cloud servers used today these results would not represent the truth, since servers are generally more powerful than commercial mobile devices, our architecture is based on ARM, which is still not widely used for commercial datacenters.

From this figure, we can also observe that the absolute execution time for the larger 1024x1024 resolution image is less than 250 milliseconds. This means that decoding image data on the client, with commonly used sizes, is achievable without having a noticeable impact on the performance of the device. Thus, by leveraging a low impact on device performance and allowing the secure system deployed on the cloud server to avoid implementing an image library with a large trusted computing base we accomplish a balanced implementation equilibrium while achieving the goals of maintaining a small TCB and a reduced overhead.

5.4.2 Comparison Between Encoded and Decoded Images

Encoded images are often subject to compression techniques which allow them to maintain a small size while representing a large amount of data. These techniques are specially useful when images are transmitted through the network among several entities, because the low size allows these images to be sent under constraining bandwidth conditions without representing a high impact on performance. The disadvantage of this approach is that the end nodes must be capable of decoding these images in order to display or process them. Darkroom was not fitted with this decoding capability in order to maintain a small TCB. For this reason, images sent to Darkroom must be in its decoded (raw pixel) format.

Figure 5.5 shows the difference in size between the encoded JPEG images tested before and their equivalent decoded Bitmap format. In this figure, the Y-axis represents the size of an image in kilobytes while the X-axis represents the different-sized images tested. We can observe that the left bar, which represents the encoded JPEG format, has a much smaller size than that of the right bar, which represents the decoded bitmap image. In fact, for the 1024x1024 resolution image we have an increase of 365% of the image size when using the decoded bitmap format, and the remaining resolutions also show increases of approximately 300% in size. The consequence is that this increase

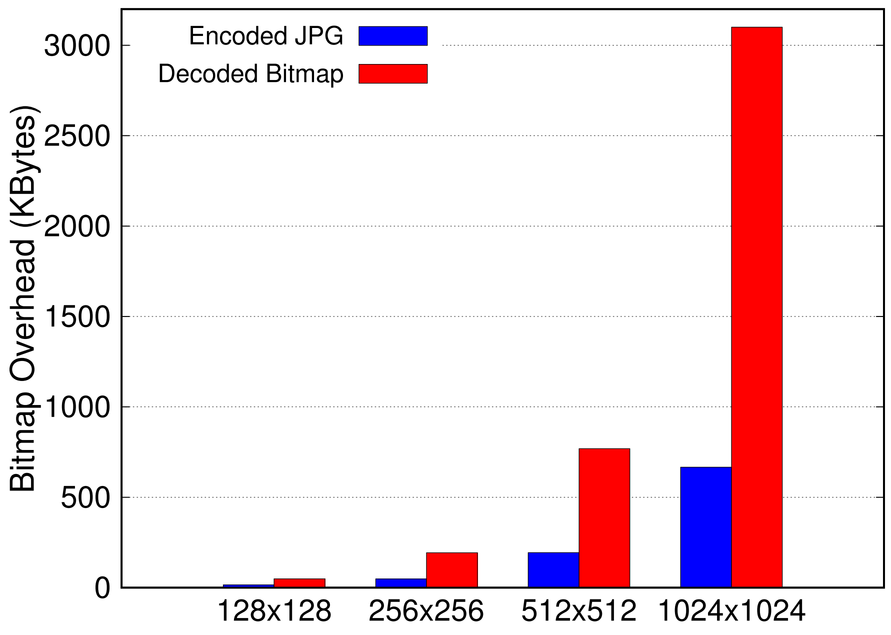

Figure 5.5: Size comparison between encoded JPEG and decoded Bitmap.

may be responsible for a lower performance when uploading and downloading the images to the secure service on the cloud.

But, even though we can observe a large increase on the size of the image we want to send to Darkroom via the client application, we still believe the use of a decoded image format is wise because it allows us to maintain a small trusted computing base. As an example, we measured the code base of the two most popular libraries for processing the JPEG and PNG encoded image formats, `libjpeg` and `libpng`, and concluded they have large code bases of approximately 106 KLOC and 43 KLOC, respectively. On the other hand, by allowing Darkroom to directly process decoded pixel data, we were able to implement six transformation functions with a code base of approximately 1300 LOC. More details regarding Darkroom's trusted computing base are described in Section 5.5.

5.4.3 Envelope Size Overhead

In order to securely send and store the sensitive image data in the normal world operating system of the TrustZone-enabled platform, we developed a structure called envelope. This structure is responsible for holding the encrypted image data as well as the symmetric key used to encrypt it, along with the message authentication code for integrity checking by Darkroom. To evaluate whether using this structure represents an impact on the system, we measured the size of the tested JPEG images and compared it to the size of the envelope which contains these images. Note that for this experiment we are not using the decoded pixel data for creating the envelope, but rather the encoded image since we are interested in assessing the difference between the data and the envelope containing the data, and for this the type of data used is irrelevant.

Table 5.2 shows the evaluation results of measuring the size of the image and the size of the enve-

Image	JPEG	Envelope	Overhead	Percentage
128x128	14380	14544	164	1.14 %
256x256	48617	48784	167	0.34 %
512x512	198500	198656	156	0.08 %
1024x1024	681500	681664	164	0.02 %

Table 5.2: Size comparison between the image and envelope (Bytes).

lope containing the encrypted image inside. As we can observe, the difference between the size of the envelope and the size of the data sealed inside it is very small. We can state that the envelope overhead is independent of the size of the original image, because different-sized images show a constant size overhead of approximately 165 bytes. This increase represents the size of the symmetric cryptographic key and the message authentication code inside the envelope. Since the image is encrypted with a symmetric key algorithm, the difference in size between the original image and the corresponding encrypted image represents only the padding necessary for the cryptographic algorithm to work. The 512x512 image needs smaller padding (12 bytes) compared to the remaining images, which is way the size overhead is smaller for this image. Considering the small impact of using the envelope structure for the size of the data to be sent, we consider this overhead negligible.

5.5 Trusted Computing Base Size

One of the requirements for Darkroom discussed in this dissertation was developing a secure data processing service while maintaining a small trusted computing base. In order to evaluate the TCB of our Darkroom prototype, we counted the number of lines of code present in the prototype and compared it to other systems', namely similar microkernels, a full-featured kernel and popular libraries. We also analyse the trusted computing base of the Darkroom kernel when compared to the *base-hw* microkernel from which it was adapted.

Table 5.3 shows the measurement of the code base of several systems. The first system is the Linux kernel, version 2.6.35, which was used as the normal world kernel. This kernel comprises a large and complex code base of approximately 10400 KLOC [1]. The second system is an image library which, although it is a library for reading and writing JPEG image files, has a large code base of approximately 106 KLOC. The third system is the Fiasco.OC microkernel adapted by the Genode developers. This system has a code base of approximately 60 KLOC and starts to approach the small code base size desired for a secure world kernel. Lastly we have Darkroom and the *base-hw* microkernel.

As we can see, both Darkroom and the *base-hw* microkernel have a much smaller code base than that of the other mentioned systems, with 25.5 and 20 KLOC respectively. This happens partially because the *base-hw* microkernel was built from scratch specifically for this TrustZone-aware platform while the other systems are built to be deployed on different platforms. Additionally these systems have implemented features which are not necessary for Darkroom or for the *base-hw* microkernel. Some of these systems also do not have the particular requirement of maintaining a small trusted computing base, such as the case of the image libraries.

[1] `https://www.linuxcounter.net/statistics/kernel`

System	Code Base Size
Linux kernel	10400 KLOC
Libjpeg	106 KLOC
Fiasco.OC	60 KLOC
Darkroom prototype	25.5 KLOC
base-hw	20 KLOC

Table 5.3: Code base size comparison.

Component	Code Base Size
TZ VMM	815 LOC
Miscellaneous code	792 LOC
Image Processing Engine	1364 LOC
Cryptographic Engine	2516 LOC
Total	5487 LOC

Table 5.4: Darkroom's component code base size.

We can also see that Darkroom has a larger code base than the *base-hw* microkernel from which it was adapted. This is because, in order to support the design of Darkroom, our prototype must implement additional features which are not supported by the initial implementation of the *base-hw* microkernel. Note that the original VMM implemented by Genode developers is not included in the *base-hw* microkernel's code base size measurement, since it was originally implemented as a client application running on top of it. In addition to the original *base-hw*, our Darkroom prototype also implements the modified TZ VMM, a cryptographic engine and an image processing engine.

To understand how the additional 5.5 KLOC code base is organized, we counted the code base of the different components implemented in Darkroom. Table 5.4 shows the code base size of the components which comprise Darkroom. We started by measuring the code base related to the TZ VMM. This includes the VMM implementation, a secure memory manager and the ATAGs for the normal world Linux kernel. The TZ VMM code sums up to 815 LOC. We then proceeded to measure miscellaneous code implemented on top of the *base-hw* kernel which does not fit any of the components described above. This miscellaneous code sums up to 792 LOC, mainly for reading and writing to block and serial devices. We continued to measure the code base of the Darkroom prototype. The image processing engine implements an image processing manager, which is responsible for handling the image data and selecting the correct transformation requested by a remote client. Additionally, it also implements the six transformation functions supported by this prototype. This component sums up to 1364 LOC.

Lastly, we measured the largest component (excluding the *base-hw* microkernel), the cryptographic engine. This component implements a cryptographic manager which is responsible for encrypting and decrypting the image envelope sent to the secure world. Additionally, this component also implements the cryptographic algorithms of AES-128 (adapted from mbedtls), RSA and HMAC. The code base size for this component is 2516 LOC. Combining the code base size of all these components we have a code base of 5487 LOC, which when added to the original *base-hw* microkernel, sums up to 25.5 KLOC.

Even though this prototype's TCB is small, the use of a software cryptographic manager which implements the three cryptographic protocols supported by Darkroom has a high impact on the size of this prototype's code base. This is because it is the largest and most complex component present in

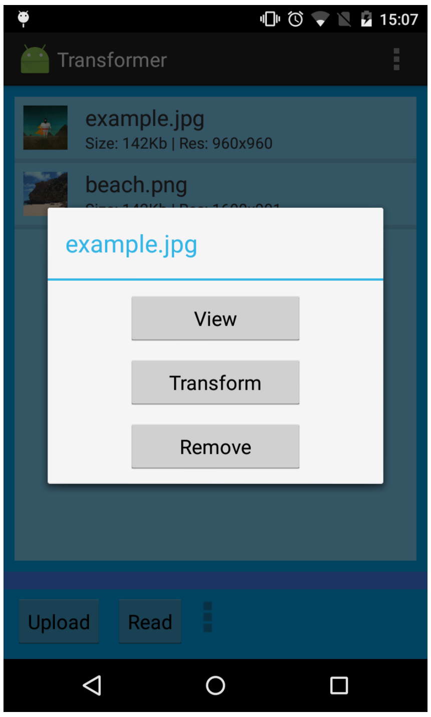

Figure 5.6: Screenshot of the Android client application.

the prototype, aside from the *base-hw* microkernel. The use of the Symmetric/Asymmetric Hashing and Random Accelerator (SAHARA) hardware module present in the i.MX53 QSB would allow us to save the additional 2.5 KLOC of the cryptographic engine while providing a better performance.

5.6 Use Case Application

The remote client application implemented for testing purposes is a really simple Android application for sending and receiving image envelopes and requesting image transformations from the image processing service running in Darkroom. The application was developed for a LG Nexus 4 Android device and has a simple execution-flow. Using this application, the user can access the photo gallery in order to select images for the upload process. After selecting an image for the upload process, the application identifies the image format and retrieves the pixels of that image to a buffer. These pixels are what the application is going to send to the image processing service via an envelope. Then a symmetric key is generated for encrypting the image. In our implementation the server's public key is hard-coded in the application, but for a ready to deploy system this key would be downloaded in a certificate from the cloud provider. Then, as explained in the design, the encrypted image data and the symmetric key encrypted with the server's public key are sent, together with the HMAC of the message for integrity checking, to the normal world of the Darkroom-enabled server. This envelope is stored in the normal world until an image transformation is requested.

The user is then given a list of uploaded images. By selecting an image the user can choose to request a transformation for that image or delete it from the server. Additionally, he can choose to upload more images to the Darkroom server. Figure 5.6 shows a screenshot of the application where we can see the options given to the user when an image is selected. In this case the image is called `example.jpg` and in the background we can see the list of uploaded images and an option to upload additional images. When the user selects the transform option, a list of transformations available in the server is shown. By selecting any of these transformations, a packet with the transformation identifier and the image ID is sent to the Darkroom server which proceeds with the transformation. After this transformation is completed, a new list of images is sent to the client which includes the new transformed image. The user can then select the newly transformed image and download the envelope from the server. This envelope is then opened at the client and the image can then be visualized by the user.

Developing the client application is easy using mobile development kits. Darkroom expects the application to decode the image before sending it, meaning that the application must send the raw pixel data to Darkroom instead of sending an image encoded with formats such as JPEG and PNG. Additionally, the client has to make sure the envelope is correct. Darkroom expects the envelope to be comprised of the encrypted image, the symmetric key used for encrypting the image (encrypted with the Darkroom's public key) and the message authentication code for integrity and authentication checking. Since today's mobile development platforms make it really easy to use features such as image processing libraries and cryptographic libraries inside applications, it is easy for developers to follow through with these requirements. For this application we used the `Bitmap` class from the `android.graphics` for decoding the JPEG images into arrays of pixel data and standard security libraries such as the `java.security` and `javax.crypto` for leveraging the cryptographic algorithms and keys.

5.7 Discussion on Different Implementation Attempts

We started with the goal of implementing a small runtime running in the secure world, just to support the needed cryptographic features for secure image transferring between the client application and the secure world runtime. After studying the boot process for an ARM platform and studying the TrustZone features, we implemented a small runtime running in the secure world. This runtime supported the context-switching between both worlds. This context-switch was supported via an assembly routine called when a software interrupt was triggered by the `SMC` instruction. This runtime also booted a simple runtime in the normal world, but was not capable of supporting a full-featured normal world OS.

We also implemented an image processing unit (IPU) driver adapted from a public on-board diagnostic suite for the i.MX53 QSB board. This IPU allowed us to display the images before and after processing, but was eventually considered unnecessary for Darkroom. Afterwards, we focused on supporting a normal world rich OS to provide network and storage capacity for Darkroom. The first stage was adapting a popular bootloader we knew worked for this platform. We adapted U-boot [2] to configure the secure world with our previously implemented secure runtime and also to launch a basic normal world runtime. We then tried launching a Linux kernel. However, because of our secure world configurations, the normal world Linux kernel hangs when it does not have access to secured critical hardware devices and as such it does not boot successfully.

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

We tried deploying the adapted Linux kernel from Genode directly in the normal world, without a VMM in the secure world, but the Linux kernel would not boot successfully. For this reason we experimented with Genode's *base-hw* microkernel. Since this experience yielded a successful boot of a normal world Linux kernel and a secure world microkernel, we decided to adapt this solution for Darkroom. Another obstacle to the implementation of Darkroom's prototype was the attempt to leverage our development board's cryptographic hardware coprocessor (SAHARA). This hardware component is proprietary and, in order to have access to the needed documentation, it is required to sign a non-disclosure agreement with Freescale, where we guarantee that the documentation is not made public. We contacted Freescale and proceeded with signing the non-disclore agreement. However, Freescale did not follow the agreement and we were never given access to the documentation necessary to use this hardware coprocessor. For this reason, we started exploring different solutions for our cryptographic support, namely a solution relying on cryptographic software, which ultimately leads to an increase of our prototype's trusted computing base.

5.8 Security Considerations and Limitations

In this section we discuss the security considerations and limitations of Darkroom. The attack surface of Darkroom comprises the `SMC` instruction exposed to the normal world OS via the Darkroom API and the shared memory region allocated for each world-switch request. The Darkroom API is relatively narrow, which limits the exposure of code vulnerabilities. The shared memory region allocated for each world-switch request is only used to read and write data, rather than executing possible code loaded to it. For this reason, the use of this shared memory region for attacks is unlikely, since the only thing the normal world OS can do is not to write the correct data to the shared memory region or to not write data at all, both of which the secure world detects by checking the HMAC.

As mentioned in Section 2.4, ARM TrustZone supports secure memory. This renders all memory allocated to the secure world as inaccessible by the normal world. Using this approach, TrustZone protects the secure world from a compromised normal world trying to infect it. But ARM TrustZone does not encrypt the secure world memory nor does it protect the system from bus monitoring attacks. This means that, by leveraging bus monitoring attacks, an attacker can make the entire memory contents vulnerable, including the memory allocated for the secure world.

When the normal world application requests a world-switch via the Darkroom API, a compromised rich OS can spoof or deny this request. The compromised OS can pretend it executes the world-switch and returns to the normal world application without proceeding with the secure data processing request. But this action can be detected by the client application, since the result of the requested processing will not correspond to the desired transformation. The same is true for denial-of-service attacks where the compromised OS ignores the request of the normal world application. In both cases the sensitive data inside the envelope is protected and never becomes exposed to the compromised components, meaning that the security properties guaranteed by Darkroom are still maintained.

Additionally, an attacker which successfully exploits a vulnerability in Darkroom's code is capable of not only compromising the code and data our system protects, but also compromise the entire normal world system. These attacks are out of the scope of this dissertation and are not considered.

Summary

In this chapter we evaluate the performance of our prototype implementation and we give a theoretical analysis and discussion on the initial implementation attempts as well as some general security considerations and limitations of the system. We start the chapter by describing the evaluation methodology used throughout. We then proceed with the analysis of the performance of image transformation functions in Darkroom. We also measured the overhead of the context-switching mechanism. We proceed with the evaluation of the impact of the image envelope and the trusted computing base size.

We end with an analysis on the use and development of the mobile client application and a discussion on previous implementation attempts of this system before discussing some security considerations and limitations of the system. The next chapter finishes this dissertation by presenting the conclusions and orientations for future work.

6 Conclusions

6.1 Conclusions

Nowadays the use of cloud services for processing and storing personal information is a growing trend and with it the chance of attackers exploiting vulnerabilities in the cloud provider's software infrastructure and disclosing sensitive user information increases. This data can then be used by attackers for malicious purposes such as espionage, blackmail, identity theft and harassment. For this reason there is a need to develop a system which allows sensitive data to be processed on the cloud without it being exposed to the untrusted and possibly compromised software components of the cloud provider's software infrastructure.

In this dissertation we introduce Darkroom, a system that leverages ARM TrustZone to enable security-critical processing operations inside a TrustZone-protected environment on the cloud. Darkroom provides clear isolation between a potentially compromised cloud server OS and a smaller trusted execution environment that guarantees that sensitive users data stored or processed in a cloud server is handled by a smaller trustworthy code base protected inside the TEE. For Darkroom, we designed a set of cryptographic protocols that build trust when cloud administrators add dynamic code in the server, and provides integrity and authenticity properties to data processing requests executed at cloud servers.

We implement a Darkroom prototype using real TrustZone hardware by applying the design to create a secure image processing service for the cloud. This use case can be used to demonstrate the use of Darkroom to support popular web services such as Instagram and Facebook. Through experimental evaluation of our system we observe that our solution adds a small overhead of approximately 16% increase of execution time in data processing when compared to computer platforms that require the entire operating system to be trusted. We also conclude that the goal of creating a system with a reduced trusted computing base was achieved by developing a Darkroom prototype with a code base of approximately 25.5 KLOC.

6.2 Future Work

In the near future, we want to overcome implementation challenges such as support for dynamic loading of image transformations by implementing a small interpreter inside the Darkroom kernel and using the security coprocessor SAHARA for leveraging hardware-based cryptography. This would allow us to reduce Darkroom's TCB even further and increase the overall performance of the system.

Beyond overcoming the implementation limitations stated, we believe Darkroom can be improved in three ways. The first is to implement Darkroom on Intel SGX hardware. The second is exploring the use of program verification strategies to verify the correctness of the secure world runtime. This would allow to prove the correctness of Darkroom and thus improve the guarantees given to the client. The third is to demonstrate its applicability for processing other data formats beyond images. Because Darkroom's design is generic, we believe there are a lot of sensitive private information being stored on the cloud which can be protected by Darkroom's secure processing runtime. Data such as sensitive medical records, music and videos would all enjoy from a secure processing environment such as that offered by Darkroom.

Bibliography

[1] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent 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*, volume 13, 2013.

[2] Android. Android Key Store. `http://developer.android.com/training/articles/keystore.html`, 2013. Accessed: 2016-01-07.

[3] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, and Ramarathnam Venkatesan. Orthogonal security with cipherbase. In *CIDR*. Citeseer, 2013.

[4] ARM. ARM Security Technology – Building a Secure System using TrustZone Technology. ARM Technical White Paper, 2009.

[5] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Proc. of OSDI*, 2014.

[6] BBC. FBI investigates 'Cloud' celebrity picture leaks, 2014. `http://www.bbc.com/news/technology-29011850`.

[7] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. DEPSKY: Dependable and Secure Storage in a Cloud-of-Clouds. In *Proc. of EuroSys*, 2011.

[8] Ferdinand Brasser, Daeyoung Kim, Christopher Liebchen, Vinod Ganapathy, Liviu Iftode, and Ahmad-Reza Sadeghi. Regulating smart personal devices in restricted spaces. Technical report, Rutgers University, 2015.

[9] Stefan Brenner, Colin Wulf, and Rüdiger Kapitza. Running ZooKeeper Coordination Services in Untrusted Clouds. In *Proc. of HotDep*, 2014.

[10] Stephen Checkoway and Hovav Shacham. *Iago attacks: Why the system call api is a bad untrusted rpc interface*, volume 41. ACM, 2013.

[11] Computer Weekly. Why unfair contract terms put end-user trust in cloud at risk, 2016. `http://www.computerweekly.com/blog/Ahead-in-the-Clouds/Why-unfair-contract-terms-put-end-user-trust-in-cloud-at-risk`.

[12] Comscore. Digital Future in Focus. Technical report, Comscore Inc., 03 2015. `http://www.comscore.com/Insights/Presentations-and-Whitepapers/2015/2015-US-Digital-Future-in-Focus`.

[13] Andreas Fitzek, Florian Achleitner, Johannes Winter, and Daniel Hein. Tthe andix research os-arm trustzone meets industrial control systems security. In *2015 IEEE 13th International Conference on Industrial Informatics (INDIN)*, pages 88–93. IEEE, 2015.

[14] Freescale. i.MX53 Multimedia Applications Processor Reference Manual, 2015. `https://cache.freescale.com/files/32bit/doc/ref_manual/iMX53RM.pdf`.

[15] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. Sprobes: Enforcing Kernel Code Integrity on the TrustZone Architecture. In *Proc. of MoST*, 2014.

[16] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.

[17] Craig Gentry, Shai Halevi, and Nigel P Smart. Homomorphic evaluation of the aes circuit. In *Advances in Cryptology–CRYPTO 2012*, pages 850–867. Springer, 2012.

[18] Global Platform. `https://www.globalplatform.org/`, 12 2011. Accessed: 2016-01-07.

[19] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, volume 8, page 9, 2010.

[20] Intel. Intel Software Guard Extensions Programming Reference (rev 1), 09 2013.

[21] Intel. Intel Software Guard Extensions Programming Reference (rev 2), 10 2014.

[22] Prerit Jain, Soham Desai, Seongmin Kim, Ming-Wei Shih, J Lee, Changho Choi, Youjung Shin, Taesoo Kim, Brent Byunghoon Kang, and Dongsu Han. Opensgx: An open platform for sgx research. In *Proceedings of the Network and Distributed System Security Symposium, San Diego, CA*, 2016.

[23] Stefan Kalkowski, Norman Freske, et al. The Genode OS Framework, 2006. `http://genode.org/`.

[24] Stefan Kalkowski, Norman Freske, et al. Genode - An Exploration of ARM TrustZone Technology, 2014. `http://genode.org/documentation/articles/trustzone`.

[25] Kari Kostiainen et al. On-board credentials: an open credential platform for mobile devices. Technical report, Aalto University, 2012.

[26] Wenhao Li, Haibo Li, Haibo Chen, and Yubin Xia. Adattester: Secure Online Mobile Advertisement Attestation Using Trustzone. In *Proc. of MobiSys*, 2015.

[27] Wenhao Li, Mingyang Ma, Jinchen Han, Yubin Xia, Binyu Zang, Cheng-Kang Chu, and Tieyan Li. Building Trusted Path on Untrusted Device Drivers for Mobile Devices. In *Proc. of APSys*, 2014.

[28] Xiaolei Li, Hong Hu, Guangdong Bai, Yaoqi Jia, Zhenkai Liang, and Pratiksha Saxena. DroidVault: A Trusted Data Vault for Android Devices. In *Proc. of ICECCS*, 2014.

[29] Yanlin Li, Jonathan McCune, James Newsome, Adrian Perrig, Brandon Baker, and Will Drewry. Minibox: A two-way sandbox for x86 native code. In *2014 USENIX annual technical conference (USENIX ATC 14)*, pages 409–420, 2014.

[30] Linaro. OP-TEE. `https://wiki.linaro.org/WorkingGroups/Security/OP-TEE`, 2014. Accessed: 2016-01-07.

[31] Dongtao Liu and Landon P Cox. Veriui: Attested Login for Mobile Devices. In *Proc. of HotMobile*, 2014.

[32] mbedTLS. mbed TLS, 2015. https://tls.mbed.org/.

[33] Brian McGillion, Tanel Dettenborn, Thomas Nyman, and N. Asokan. Open-TEE – an open virtual trusted execution environment. Technical report, Aalto University, 2015. Accessed: 2016-01-07.

[34] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP@ ISCA*, page 10, 2013.

[35] Naked Security. Google Drive security hole leaks users' files, 2014. https://nakedsecurity.sophos.com/2014/07/10/google-drive-security-hole-leaks-users-files.

[36] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proc. of SOSP*, 2011.

[37] PrivateCore. Trustworthy Cloud Computing with vCage, 2014. https://privatecore.com/vcage/.

[38] Samsung. Samsung KNOX - White Paper : An Overview of Samsung KNOX. Technical report, Samsung Electronics Co. Ltd., 04 2013. http://www.samsung.com/es/business-images/resource/white-paper/2014/02/Samsung_KNOX_whitepaper-0.pdf.

[39] Nuno Santos, Krishna P Gummadi, and Rodrigo Rodrigues. Towards trusted cloud computing. *HotCloud*, 9:3–3, 2009.

[40] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Using ARM Trustzone to Build a Trusted Language Runtime for Mobile Applications. In *Proc. of ASPLOS*, 2014.

[41] Nuno Santos, Rodrigo Rodrigues, Krishna P Gummadi, and Stefan Saroiu. Policy-sealed data: A new abstraction for building trusted cloud services. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 175–188, 2012.

[42] Felix Schuster, Manuel Costa, CÃ©dric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *Proc. of IEEE S&P*, 2015.

[43] Agam Shah. Intel faces a challenge in the server market with new ARM chips, 2016. http://www.networkworld.com/article/3077524/intel-faces-a-challenge-in-the-server-market-with-new-arm-chips.html.

[44] Alexander Shraer, Christian Cachin, Asaf Cidon, Idit Keidar, Yan Michalevsky, and Dani Shaket. Venus: Verification for untrusted cloud storage. In *Proc. of CCSW*, 2010.

[45] SierraTEE. http://www.openvirtualization.org/, 2012. Accessed: 2016-01-07.

[46] Rohit Sinha, Sriram Rajamani, Sanjit Seshia, and Kapil Vaswani. Moat: Verifying confidentiality of enclave programs. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1169–1184. ACM, 2015.

[47] Stephen Smalley and Robert Craig. Security enhanced (se) android: Bringing flexible mac to android. In *NDSS*, volume 310, pages 20–38, 2013.

[48] Zak Stone, Todd Zickler, and Trevor Darrell. Autotagging facebook: Social network context improves photo annotation. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on*, pages 1–8. IEEE, 2008.

[49] He Sun, Kun Sun, Yuewu Wang, and Jiwu Jing. Reliable and Trustworthy Memory Acquisition on Smartphones. *Transactions on Information Forensics and Security*, 10(12):2547–2561, 2015.

[50] He Sun, Kun Sun, Yuewu Wang, and Jiwu Jing. TrustOTP: Transforming Smartphones into Secure One-Time Password Tokens. In *Proc. of CCS*, 2015.

[51] He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Haining Wang. Trustice: Hardware-assisted isolated computing environments on mobile devices. In *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*, pages 367–378. IEEE, 2015.

[52] T6 TEE. `https://www.trustkernel.com/products/tee/t6.html`, 2014. Accessed: 2016-01-07.

[53] Richard Ta-Min, Lionel Litty, and David 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.

[54] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. Deepface: Closing the gap to human-level performance in face verification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1701–1708, 2014.

[55] Sai Deep Tetali, Mohsen Lesani, Rupak Majumdar, and Todd Millstein. Mrcrypt: Static analysis for secure cloud computations. *ACM SIGPLAN Notices*, 48(10):271–286, 2013.

[56] TLK. `http://www.w3.org/2012/webcrypto/webcrypto-next-workshop/papers/webcrypto2014\_submission\_25.pdf`, 09 2014. Accessed: 2016-01-07.

[57] Vic JR Winkler. *Securing the Cloud: Cloud computer Security techniques and tactics*. Elsevier, 2011.

[58] Bo Yang, Kang Yang, Yu Qin, Zhenfeng Zhang, and Dengguo Feng. DAA-TZ: An efficient DAA scheme for mobile devices using ARM TrustZone. In *Trust and Trustworthy Computing*, pages 209–227. Springer, 2015.

[59] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 203–216. ACM, 2011.

[60] Yan-Tao Zheng, Ming Zhao, Yang Song, Hartwig Adam, Ulrich Buddemeier, Alessandro Bissacco, Fernando Brucher, Tat-Seng Chua, and Hartmut Neven. Tour the world: building a web-scale landmark recognition engine. In *Computer vision and pattern recognition, 2009. CVPR 2009. IEEE conference on*, pages 1085–1092. IEEE, 2009.