Darkroom

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

(extended abstract of the MSc dissertation)

Tiago Luís de Oliveira Brito
Departamento de Engenharia Informática
Instituto Superior Técnico

Advisor: Nuno Miguel Carvalho Santos

***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 exposed to attacks. By exploring subtle vulnerabilities present in applications, OSs and hypervisors, an attacker may compromise the system and disclose sensitive user data hosted in it. This data can then be used 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. This paper 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 or Facebook. Through experimental evaluation we observe that our solution adds a small overhead to data processing when compared to computer platforms that require the entire OS to be trusted.**

I. 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. However, these services frequently handle user sensitive data which have 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 [1], and user document [2] 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 [3].

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 in scenarios where content needs to be processed by the cloud service. Notable examples include cloud services such as Facebook or Instagram which apply transformation functions to the images uploaded by the users, for example to rescale, rotate, or reencode 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 [4], enforce authentication mechanisms [5], [6], or provide OS introspection and monitoring capabilities [7], [8], [9]. One of the reasons this approach has been so popular in the mobile landscape has to do with ARM TrustZone [10], a technology that allows the implementation of TEE systems and is present in the majority of mobile device processors.

In this paper, we explore the adoption of ARM TrustZone technology to provide an isolated environment for processing data securely on the cloud. Specifically, we present the design of Darkroom, one of the first systems leveraging ARM TrustZone to offer a secure data processing environment for cloud-hosted services. In order to demonstrate the applicability of Darkroom for secure data processing, we focus this paper on the use case of secure image processing. We also implement a system prototype on real TrustZone-enabled hardware and a mobile application to act as a client for sending images, requesting transformations and receiving images to and from the prototype. Experimental evaluation of the prototype shows a reduced overhead (average 16% for the tested images), with a small TCB (approximately 25.5 KLOC), and is 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).

The rest of this document is organized as follows. Section II describes the related work. Section III describes Darkroom's architecture. Section IV describes the implementation of the Darkroom prototype. Section V presents the results of the experimental evaluation study. Finally, Section VI concludes this document by summarizing our findings and future work.

II. Related Work

Over the past years, extensive work has been focused on data security for untrusted cloud-hosted storage services. Systems such as Venus [11] or DEPSKY [12] 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 in the cloud. To overcome this limitation, researchers have studied ways to allow for encrypted data processing by leveraging advanced cryptographic techniques. CryptDB [13] is a representative example of such a system which employs 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.

An alternative approach to securing cloud processing is to leverage Intel's upcoming technology Security Guard Extensions (SGX). 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-enabled hardware deployed by the cloud provider, enclaves can run arbitrary functions at native processor speed, hence faster than homomorphic encryption 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 [14] and VC3 [15] 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 [14]. Thus, similarly to SGX, we take a TEE-based approach in the design of our system, but explore the use of ARM TrustZone technology, which is more mature than SGX and available in commodity hardware.

ARM TrustZone is a security extension present in ARM processors that provides a hardware-level isolation between two execution domains: *normal world* (NW) and *secure world* (SW). 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 SW. 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 Secure Monitor Call (`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 SW. 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 SW, at system boot time.

Given that the majority of mobile devices are equipped with ARM processors, ARM 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 [16], implementing OS introspection mechanisms [7], enabling secure data storage [4], providing secure authentication mechanisms [5], implementing one-time-password solutions [6], or providing forensic tools for trusted memory acquisition [8]. Other systems provide general-purpose frameworks for splitting mobile app code and running it in the TEE [17] or enabling trusted I/O between the user and TrustZone-based services [18]. Existing systems, however, are not directly applicable to the cloud setting due to the cloud's specificity in terms of application requirements and system administration model. Brenner et al. [19] took some first steps to using ARM TrustZone on the cloud by building a TEE-protected privacy proxy for Zookeeper. Their system provides a confidential coordination service for distributed applications, which constitutes a different application scenario than the focus of our work.

III. Design

This section describes Darkroom, a TrustZone-based system which allows for secure data processing on the cloud. To demonstrate the applicability of Darkroom for real world data processing, we focus this design on the image processing use case. Thus, this section describes a system capable of processing images without exposing them to untrusted components. This is done by leveraging TrustZone to securely process the images in isolation from the rich OS.

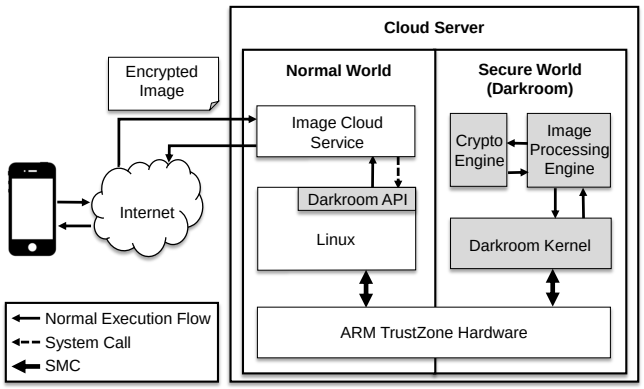

Figure 1. Architecture

A. Assumptions and Threat Model

To justify our design, we describe the assumptions and threat model considered for Darkroom. Our system aims to protect data processed on the cloud from being exposed to untrusted components. Since we assume an untrusted software platform, we can also assume an adversary which may acquire control of the provider's software infrastructure, interrupt the execution of the user program indefinitely (denial-of-service attacks are not considered by Darkroom) and falsify the results of calls made to the OS. 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 the processor is correct, this is, we assume TrustZone features supported by the processor are correctly implemented and cannot be compromised by an attacker. On the other hand, we do not consider side-channel attacks, since solutions to these attacks require hardware modifications which are out of the scope of this paper. Note that we are primarily concerned with potential breaches from software exploits to normal world programs. In particular, we want to prevent attackers that manage to take over control of the application, or the OS residing in the normal world, from violating the confidentiality of user images. We assume that both 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.

B. Architecture

Figure 1 shows the components of our solution and represents a 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 OS run in the normal world. 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 the `SMC`. This gives control to the secure world which decrypts the image data and executes the requested transformation on it. After processing, the image is encrypted again and sent to the normal world.

The first NW component which interacts with the sensitive data is the Image Cloud Service. This component represents image processing services such as Facebook or Instagram. This is an untrusted application which leverages the Darkroom API to enjoy the secure processing capabilities of Darkroom. This API is the component which sends the data to the SW and is also responsible for triggering the world-switch via the `SMC` instruction. Once the world-switch is triggered, the Darkroom kernel receives the encrypted data and redirects it to the two remaining SW components. The Crypto Engine is responsible for decrypting and encrypting the image while the Image Processing Engine is responsible for manipulating the unencrypted data.

ARM TrustZone specification [10] defines three main types of software architectures for TrustZone-aware systems. The first is a dedicated secure world OS, the second is a synchronous library of code inside the SW and the third is an intermediate option between the two. Our system needs the inter-world communication and small TCB offered by the library approach, but also needs the OS abstractions and flexibility offered by a secure world OS. For that reason we want to develop Darkroom as an intermediate approach between the two. This is done by building a small custom kernel which supports only the necessary features for Darkroom, such as memory management, threading and scheduling, while maintaining a small TCB.

C. Deployment on Cloud Servers

We envision Darkroom to be deployed on servers maintained by the service provider. The 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 OS or hypervisor. This ensures the trusted code is executed before the untrusted rich OS can compromise the platform.

In this process, the provider generates a *root key* for each server, which is a public key pair KR whose private part (KR^-) is bundled into the Darkroom image before it is flashed onto the firmware. The private part of this key will be accessible only by Darkroom and there will be a unique root key for each server. The public part KR^+ is certified by the provider to assert its authenticity. This key is fundamental to ascertain the authenticity of the Darkroom instance to a remote client. Note that, in addition to the root key, the Darkroom image contains public keys of cloud administrators authorized to perform security operations, such as setting up image transformations.

The secure boot mechanism offered by TrustZone ensures that the software running inside the secure world is authentic to that flashed by the provider and is not the result of an attack performed at boot time, the most critical point in time when the system is more vulnerable to attacks. Together, the secure boot and secure deployment features

guarantee to a remote client that the system is running a unmodified Darkroom instance, but do not guarantee that Darkroom is effectively running in the secure world. This is because, unlike Intel SGX, ARM TrustZone does not support hardware-based attestation.

D. 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 transformations (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 loaded code is trustworthy. This is because this code will have access to the raw image data submitted by clients and can potentially compromise that data. 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 meta-data that indicates the TF name and the type of input parameters, e.g., *rotation (int degree)*.

E. 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 afterwards. The second stage is to securely *upload* an image from the client to the server. In this process, we must ensure that the image is encrypted such that it can only be decrypted within Darkroom. The 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.). Forth 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.

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 a specific cloud service (whose logic runs in the rich OS and client). To this end, cloud administrators must generate the key pair KS and securely load it to the Darkroom runtime of each 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 2:

1. Initial setup:: To allow the Darkroom service and client to check the integrity of future exchanged messages, the client needs to share a secret key with the 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 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 wants to request 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

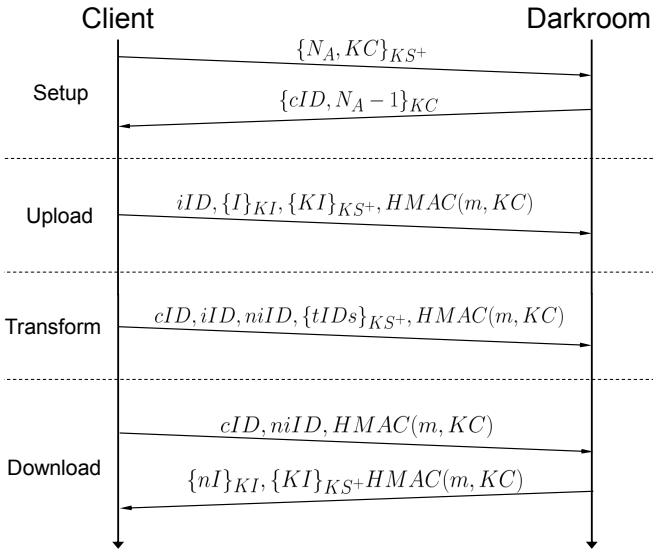

Figure 2. Communication of Darkroom processing stages.

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.

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 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.

IV. Implementation

We implemented a prototype of Darkroom for the Freescale NXP i.MX53 Quick Start Board. A fundamental concern when building our system was to keep 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. Next, we describe the most relevant implementation details of the components of our prototype.

A. 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 [20] framework to build our secure world kernel. Genode is a framework for building special-purpose OSs. 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. Leveraging the strictly necessary building blocks we can build a runtime with a low trusted computing base complexity yet support the features described above.

Genode offers a custom kernel called *base-hw* made for the i.MX53 QSB which runs in the secure world and has a codebase of 20 KLOC (thousand lines of code), much smaller than the Linux kernel. In addition, Genode implements a Virtual Machine Monitor (VMM) which manages a custom-made paravirtualized Linux kernel in the normal world. The use of 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 kernel. Taking this into account, the *base-hw* microkernel does not solve all low-level problems associated with Darkroom, such as the inter-world communication, but it significantly decreased the difficulties of implementing a microkernel from scratch.

The VMM is prepared to support basic context-switches requested by the normal world runtime. This TrustZone-aware VMM, or 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 NW. But it still lacks a mechanism to identify the source of the `SMC`, a mechanism to send arguments between worlds and a mechanism for the SW to receive an image envelope. To solve this problem we modified the original VMM to use CPU registers to send arguments form the NW to the SW. The secure world can read the registers from the saved CPU state of the normal world and interpret them.

Another important mechanism 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, meaning 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 send allows the secure world to access the shared memory region where the image data is stored. 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. This allows the secure kernel to access this shared memory region.

B. Normal World Components

To communicate with the remote client we implemented a normal world application. This application, built on top of the normal world Linux kernel, is responsible for the network communication (upload and download of image envelopes) and storing the envelope containing the image to be processed in the normal world operating system's file system. Additionally, this application is also responsible for sending the image envelope and corresponding desired transformation request to the secure world kernel through the use of the Darkroom API. The purpose of this application is to simulate a realistic server app, such as Facebook or Instagram, which can enjoy the security features introduced by Darkroom in secure image processing.

In order for untrusted applications running on top of the normal world operating system to use Darkroom, the rich OS needs to support a mechanism for these applications to request a world-switch. We modified the Linux kernel and implemented the Darkroom API as a flexible solution which allows user-level applications to request a world-switch to the NW rich OS. The Darkroom API offloads the world-switch related operations to the kernel by offering the new `smc` system call, which is responsible for allocating and preparing the shared memory region, proceeding with the world-switch request via the `SMC` interrupt and returning the results back to the user-level application.

To allocate the shared memory region inside the `smc` system call we leverage the `kmalloc` function and copy the content of the envelope to the newly allocated share memory region. We then call the `virt_to_phys` funtion to translate the virtual memory address returned by `kmalloc` to a physical memory address the secure kernel can use. 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.

Table I
DESCRIPTION OF THE TRANSFORMATION FUNCTIONS IMPLEMENTED.

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

C. 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.

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, the use SAHARA would allow us to support the needed 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. We managed to implement all the necessary cryptographic algorithms to support the features described in Darkroom's design by adapting the popular *mbed TLS* cryptographic library [21] for the AES-128 protocol and simpler implementations for the remaining RSA and HMAC algorithms. We had to resort to these simpler implementations for the remaining algorithms because *mbed TLS* relies on many Linux specific system calls for file management, input/output and for entropy gathering which the *base-hw* microkernel does not support. On the other hand, for a deployment environment, the system would be better-off using a hardware coprocessor like SAHARA. The use of this coprocessor would not only allow for a reduced TCB but also better performance.

D. Image Transformation Functions

The Image Processing Engine runs on top of the Darkroom kernel and manages the transformation functions loaded into the system. These functions are responsible for

```
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;
}
```

Listing 1. Sample code of gray-scale transformation function.

effectively processing the image data sent from the client in an isolated environment managed by the kernel. 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 I. All transformations offered by this component were implemented from scratch without having to rely on any image library. Listing 1 provides the sample code of a transformation function to change the color palette of the 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 TCBs 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 retrieve the pixel data to be sent to the Darkroom server.

In Darkroom's design we described dynamic loading of transformation functions. This feature allows a trusted cloud administrator, whose key is provisioned inside Darkroom, to add new transformation functions without having to reflash a new version of the system to the platform. But supporting dynamic loading of binary functions in low-level languages like the C programming language is not easy. On the other hand, if Darkroom could support a interpreter for a language such as the Python programming language, then the support for dynamic loading of image transformation functions would be possible. To do this, a client Python application can serialize a function's bytecode which the server will reconstruct and call it, thus effectively supporting dynamic loading of functions.

V. Evaluation

In this section we study the performance of our prototype through micro-benchmarking, the size of the trusted computing base and implementation specific features regarding implementation choices made during the prototyping phase.

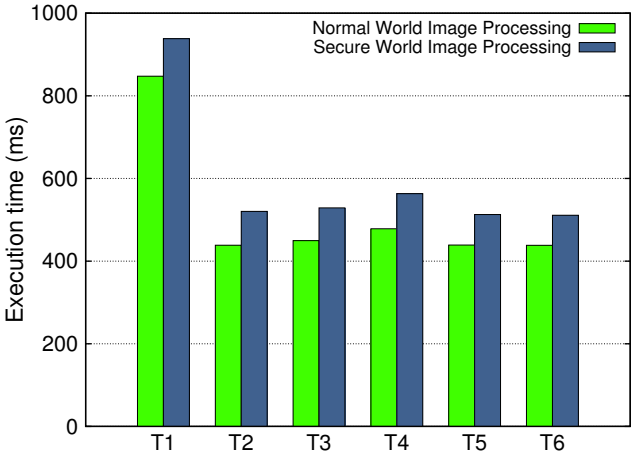

Figure 3. Execution time of all transformations.

A. 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 and an LG Nexus 4 device, featuring a Quad-core 1.5 GHz Krait processor, and 2 GB of DDR2 RAM memory as a host device for the client mobile app. The board executed our system from a mini SD card 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.

B. Performance of Image Transformation Functions

The goal of this evaluation is to measure the performance of our 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 of processing the same image transformation functions using Darkroom. When comparing these execution times we can conclude if there exists a penalty for the use of Darkroom features, such as the context-switch, and more specifically of components such as the `base-hw` microkernel.

Table I 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 3 shows the execution times of all these transformations.

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

Table II
OVERHEAD (IN MS) OF EVERY TRANSFORMATION FOR EACH IMAGE.

Image	Mean	STDEV	Mean (%)
128x128	0.17	0.41	15.7%
256x256	5.01	0.96	16.1%
512x512	20.15	1.71	16.4%
1024x1024	80.63	6.83	16.4%

penalty for the secure world execution flow, right bar, when compared to the same transformation executed in the normal world, the left bar. This is associated to the overhead of the context-switch, the implementation of the `base-hw`'s memory manager, scheduler and compiler optimizations.

We can observe that the penalty is constant across all transformations, even when the transformation has an execution time above that of other transformations. We also measured the same procedure for smaller images. Table II shows the difference between executing an image transformation in the normal world and executing the same transformation using Darkroom, across all images. We observe that the mean percentage of the overhead remains almost 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).

C. 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 4 shows the impact of T1 on the same images used for the previous test. 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 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. This is why larger images cause a more noticeable overhead than smaller images, to the point where

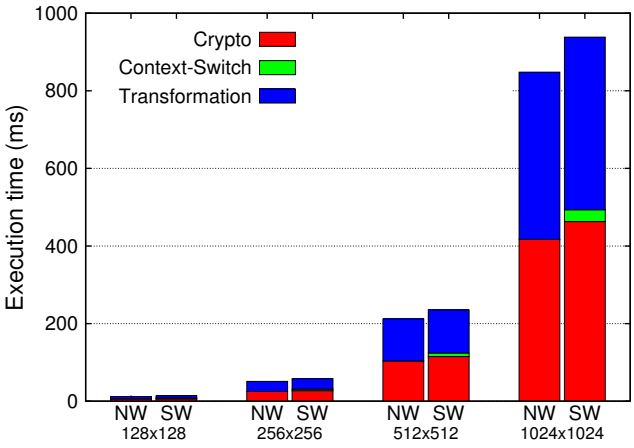

Figure 4. Execution time of the grey-scale transformation.

the context-switch overhead for smaller images 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. The secure world has more buffer allocations and more additional data copying operations to protect the image data in a secure memory region, which causes the overhead visible in the figure.

D. 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.

Encoded images are subject to compression techniques which allow them to maintain a small size while representing a large amount of data. These techniques are useful when images are transmitted through the network. 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 shows the difference in size between the encoded JPEG images and their equivalent decoded format. In this figure, we can observe that the encoded JPEG format has a much smaller size than the decoded bitmap image. In fact, for the 1024x1024 resolution image we have an increase of 365% of the image size. The consequence is that this increase may be responsible for a lower performance when

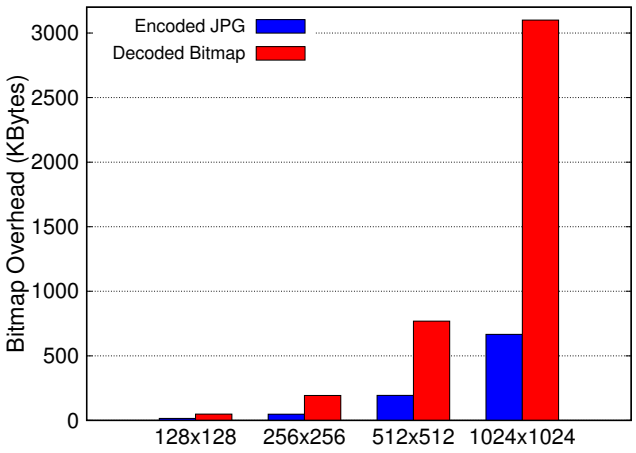

Figure 5. Size comparison between encoded JPEG and decoded Bitmap.

Table III
SIZE COMPARISON BETWEEN THE IMAGE AND ENVELOPE (BYTES).

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 %

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 believe the use of a decoded image format is wise because it allows us to maintain a smaller TCB. As an example, we measured the code base of the two most popular image libraries, `libjpeg` and `libpng`, and concluded they have large code bases of approximately 106 KLOC and 43 KLOC, respectively. 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.

We proceeded with measuring the size overhead of using the envelope structure. Table III shows in detail the added size between the original data and the envelope. We can state that the envelope overhead is independent of the size of the image because different-sized images show a constant penalty of approximately 165 bytes. This increase, which corresponds to the size of the image key and the message authentication code, is so small compared to the size of the original image that we can consider it negligible.

E. Trusted Computing Base Size

One of the major requirements for Darkroom is to maintain a small TCB. In order to evaluate the TCB of our prototype we have measured its codebase and compared it to the measurements of other systems, such as microkernels, a full-featured kernel and popular libraries. We also analyse the TCB of the Darkroom kernel when compared to the *base-hw* microkernel from which it was adapted.

Table IV shows the measurement of the codebase of several systems. As we can see, both Darkroom and *base-hw* have a much smaller codebase than that of the other mentioned systems. This happens partially because the *base-hw* microkernel was built from scratch specifically for this platform, while the other systems are built to be deployed on different platforms. Additionally these systems support features which are not needed by Darkroom or the *base-hw* microkernel.

Table IV
CODE BASE SIZE COMPARISON.

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

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. Note that the original VMM implemented by Genode developers is not included in the *base-hw* microkernel's codebase, because 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 TZVMM (815 LOC), a cryptographic engine (2516 LOC), an image processing engine (1364 LOC) and miscellaneous code (792 LOC) for reading/writing to block and serial devices.

F. Security Considerations and Limitations

The attack surface of Darkroom comprises the `SMC` instruction exposed 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, which makes its use for attacks unlikely.

ARM TrustZone supports secure memory allocated for the secure world. 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.

When the NW application requests a world-switch via the Darkroom API a compromised rich OS can spoof or deny this 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 may also compromise the entire normal world system. These attacks are out of the scope of this dissertation and as such are not considered.

VI. Conclusions

A. Conclusions

In this dissertation we introduce Darkroom, a system that leverages ARM TrustZone to bootstrap trust in a cloud based data processing service by executing the critical processing operations inside a TrustZone-enabled trusted execution environment. 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.

B. 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. We also 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 to verify the correctness of the secure world. And the third is to demonstrate the applicability of Darkroom for processing security critical data other then images.

References

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

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

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

[4] X. Li, H. Hu, G. Bai, Y. Jia, Z. Liang, and P. Saxena, "DroidVault: A Trusted Data Vault for Android Devices," in *Proc. of ICECCS*, 2014.

[5] D. Liu and L. P. Cox, "Veriui: Attested Login for Mobile Devices," in *Proc. of HotMobile*, 2014.

[6] H. Sun, K. Sun, Y. Wang, and J. Jing, "TrustOTP: Transforming Smartphones into Secure One-Time Password Tokens," in *Proc. of CCS*, 2015.

[7] X. Ge, H. Vijayakumar, and T. Jaeger, "Sprobes: Enforcing Kernel Code Integrity on the TrustZone Architecture," in *Proc. of MoST*, 2014.

[8] H. Sun, K. Sun, Y. Wang, and J. Jing, "Reliable and Trustworthy Memory Acquisition on Smartphones," *Transactions on Information Forensics and Security*, vol. 10, no. 12, pp. 2547–2561, 2015.

[9] B. Yang, K. Yang, Y. Qin, Z. Zhang, and D. Feng, "DAA-TZ: An efficient DAA scheme for mobile devices using ARM TrustZone," in *Trust and Trustworthy Computing*, 2015, pp. 209–227.

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

[11] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket, "Venus: Verification for untrusted cloud storage," in *Proc. of CCSW*, 2010.

[12] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, "DEPSKY: Dependable and Secure Storage in a Cloud-of-Clouds," in *Proc. of EuroSys*, 2011.

[13] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: Protecting Confidentiality with Encrypted Query Processing," in *Proc. of SOSP*, 2011.

[14] A. Baumann, M. Peinado, and G. Hunt, "Shielding Applications from an Untrusted Cloud with Haven," in *Proc. of OSDI*, 2014.

[15] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "VC3: Trustworthy Data Analytics in the Cloud Using SGX," in *Proc. of IEEE S&P*, 2015.

[16] W. Li, H. Li, H. Chen, and Y. Xia, "Adattester: Secure Online Mobile Advertisement Attestation Using Trustzone," in *Proc. of MobiSys*, 2015.

[17] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "Using ARM Trustzone to Build a Trusted Language Runtime for Mobile Applications," in *Proc. of ASPLOS*, 2014.

[18] W. Li, M. Ma, J. Han, Y. Xia, B. Zang, C.-K. Chu, and T. Li, "Building Trusted Path on Untrusted Device Drivers for Mobile Devices," in *Proc. of APSys*, 2014.

[19] S. Brenner, C. Wulf, and R. Kapitza, "Running ZooKeeper Coordination Services in Untrusted Clouds," in *Proc. of HotDep*, 2014.

[20] "The Genode OS Framework," http://genode.org/.

[21] "mbed TLS," https://tls.mbed.org/.