Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Master Thesis

Securing User-data in Android
A conceptual approach for consumer and enterprise usage

submitted by
Liviu Teriș

submitted
22.05.2012

Supervisor
Prof. Dr. Michael Backes

Advisors
Philipp von Styp-Rekowsky
Sebastian Gerling

Reviewers
Prof. Dr. Michael Backes
Juniorprof. Dr.-Ing. Christian Hammer

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 22.05.2012

Liviu Teriş

Acknowledgements

I would like to thank all the people who helped me with writing my thesis. First of all, I would like to thank prof. Michael Backes for giving me the chance to work on this cutting-edge topic. Also, I would like to express my deepest gratitude to my advisors Philipp von Styp-Rekowsky and Sebastian Gerling. Not only did they provide me with prompt support and advice, but they also created a pleasant working environment (and helped improve my German language skills).

Last but not least, I would like to thank my friends and family for their moral support and for helping me cope with the distance. Finally, I cannot thank enough my girlfriend Rucsandra for her patience and understanding.

Abstract

Nowadays, smartphones and tablets are replacing the personal computer for the average user. As more activities move to these gadgets, so does the sensitive data with which they operate. However, there are few data protection mechanisms for the mobile world at the moment, especially for scenarios where the attacker has full access to the device (e.g. when the device is lost or stolen). In this thesis, we tackle this problem and propose a novel encryption system for Android, the top-selling mobile operating system.

Our investigation of the Android platform leads to a set of observations that motivate our effort. Firstly, the existing defense mechanisms are too weak or too rigid in terms of access control and granularity of the secured data unit. Secondly, Android can be corrupted such that the default encryption solution will reveal sensitive content via the debug interface. In response, we design and (partially) implement an encryption system that addresses these shortcomings and operates in a manner that is transparent to the user. Also, by leveraging hardware security mechanisms, our system offers security guarantees even when running on a corrupted OS. Moreover, the system is conceptually designed to operate in an enterprise environment where mobile devices are administered by a central authority. Finally, we provide a prototypical implementation and evaluate our system to show the practicality of our approach.

Contents

1 Introduction **6**
- 1.1 Motivation . . . 6
- 1.2 Goals . . . 7
- 1.3 Contribution . . . 7
- 1.4 Outline . . . 8

2 Android **9**
- 2.1 Architecture overview . . . 9
- 2.2 The Android API and software development tools . . . 10
- 2.3 The application development model . . . 12
- 2.4 Content Providers . . . 13
- 2.5 Security features . . . 14
- 2.6 Security applications . . . 15

3 Requirements **17**
- 3.1 Attacker model . . . 17
- 3.2 A real attack . . . 18
- 3.3 The encryption system . . . 18
  - 3.3.1 Domestic usage (no central authority) . . . 19
  - 3.3.2 Enterprise usage . . . 20
- 3.4 Observations . . . 21

4 Hardware-enhanced security **22**
- 4.1 Trusted Platform Module . . . 23
- 4.2 Trusted execution . . . 24
- 4.3 ARM TrustZone . . . 25
  - 4.3.1 Hardware design . . . 26
  - 4.3.2 Software for the TrustZone . . . 28
  - 4.3.3 Related work and Implementations . . . 28
  - 4.3.4 Observations and Conclusions . . . 29

5 System design **31**
- 5.1 Local applications . . . 31
- 5.2 Control structures on the device . . . 32
  - 5.2.1 High-level overview . . . 32
  - 5.2.2 Control structures in detail . . . 33
- 5.3 Communication mechanisms . . . 36

5.3.1 Network communication . . . 36
5.3.2 On-device communication . . . 37
5.4 Common operations . . . 38
5.4.1 Installation/registration . . . 38
5.4.2 Login and access data . . . 38
5.4.3 Create user . . . 39
5.4.4 Access extension and revocation . . . 39
5.5 Observations . . . 39

6 Implementation details 40
6.1 Android content management . . . 40
6.1.1 Binders . . . 41
6.1.2 Content Providers . . . 41
6.1.3 Important applications and services . . . 42
6.1.4 Connecting to a Content Provider . . . 42
6.1.5 Running a query . . . 43
6.1.6 The problem with cursors . . . 45
6.2 The crypto application . . . 45
6.3 Architectural overview . . . 46
6.4 The SQLCipher library . . . 47
6.5 Remote SQLite encryption . . . 48
6.5.1 The native driver . . . 48
6.5.2 The Java interface . . . 49
6.5.3 The cryptographic service . . . 50
6.6 Android API related issues . . . 51

7 Evaluation 53
7.1 Test platform . . . 53
7.2 Integration with Android . . . 53
7.3 Performance measurement . . . 54
7.4 Observations . . . 56

8 Summary 58
8.1 Conclusion . . . 58
8.2 Limitations . . . 58
8.3 Future work . . . 59

Chapter 1

Introduction

The smartphone and tablet computer marked has grown constantly over the last years, and, as a result, the competition between vendors to pack even more complex functionality in these gadgets has intensified. The capabilities of mobile devices nowadays are getting close to those of personal computers, and, for the average user, a tablet or smartphone could easily replace a laptop for daily tasks that cover both work and entertainment.

The way in which people use the mobile phone has changed significantly: they use it as a personal digital assistant, for online shopping, to manage emails and so on. Together with these activities, the sensitive data they operate with has moved to the mobile devices: private information, online banking credentials, possibly confidential emails. However, most users do not realize that they walk around with mini-computers in their pockets, for example, it is uncommon for a laptop not to have a login password, whereas for most smartphones, users only use the simple "slide to unlock" screen to protect their devices[1].

1.1 Motivation

The market for mobile device operating systems is dominated by Android at the moment, with roughly 50% of all smartphones and tablets sold in 2011 running an Android distribution. This provides the first incentive behind our decision to focus on Android: its popularity also made the system a preferred target for attackers. Android comes as a proprietary software stack that runs on top of a Linux kernel, it offers a custom Java API that allows for development of third-party applications and the entire source code of the platform is available for download. The open nature of Android has encouraged the development of a variety of applications, but it also led to numerous vulnerabilities to be found and exploited.

Android is advertised as having been built with security in mind and this claim is supported by the very infrastructure of the OS: processes are sandboxed and isolated, such that malware can only produce limited damage. However, the protection offered against an attacker that has direct access to the device (if the smartphone is lost or stolen) is

[1] A survey conducted in March 2011 by Ponemon Institute for AVG Technologies, showed that less than half of the consumers use passwords or a similar type of locking for their mobile devices.

rather weak. This type of attack is referred to as *local* and it constitutes the very problem that we tackle in this thesis.

1.2 Goals

As stated in a previous paragraph, most users do not use any device locking scheme; even when such a mechanism is employed, it can easily be bypassed on some devices (see a real attack in Chapter 3). For extra protection, Android does provide its own partition encryption system, but there are several problems with this solution (discussed in more detail in Section 2.6), the most important of which is its lack of flexibility.

We seek to provide a better encryption solution that can be just as easily integrated with Android. Our encryption system focuses on the "Android way" of storing data, the Content Providers (more on these later), it comes with a user management and access control scheme, and it can easily be extended for use in an enterprise setting . The user thus has the possibility to decide what type of data is encrypted (contacts, emails etc.) and who has access to each of these containers (configure several user accounts).

This encryption system needs to be implemented such that it can be easily installed on a default Android image. Also, the changes brought to the implementation of existing Content Providers need to be minimal and third party applications that access Content Providers must not be affected in any way.

There is one extra goal that we set for our solution: it must offer security guarantees even when running on a tainted OS. In other words, if the attacker has full control over code execution and he has access to all the layers of the memory hierarchy, our device will still not reveal some secrets.

1.3 Contribution

In this thesis we provide the design and implementation details of such an encryption system for Android. We investigate the operating system in order to find the appropriate modules that need to be changed in order to secure the data containers of Android (the Content Providers) and we propose a design that puts emphasis on flexibility and transparency.

Moreover, we make the assumption that the Android software platform can be corrupted in all its layers. Therefore we turn our attention towards hardware security enhancements and investigate some solutions which allow us to move code and data to an execution environment that is isolated from the untrusted operating system. Our design is based on such a technology that is already available on many mobile devices.

Finally, we provide the conceptual design in which our encryption system is extended to be used in an enterprise environment: several mobile devices can communicate with a central server that coordinates access control policies and also these devices have the possibility to exchange information from their encrypted containers.

1.4 Outline

The first chapter of this thesis gives an overview of the Android platform with a focus on the application development environment, the Content Providers and the inbuilt security features. We then formulate a set of requirements, what is to be expected of an encryption system for Content Providers. These requirements point out some vulnerabilities that the system may have and this leads to investigating what the hardware has to offer in terms of protection. The next chapter describes the conceptual design of an application that matches the aforementioned requirements. This is followed by a detailed description of a software module that implements the part of the encryption system that is set to run on the mobile device (the encryption library) and we conclude by offering a performance analysis of our implementation.

Chapter 2

Android

Since its introduction in 2008, the Android operating system has had a constant growth on the mobile devices marked, covering roughly half of all smartphones and tablets delivered in 2011. This, on the one hand, spawned an entire industry of third-party applications for Android, and on the other hand, increased the number of attacks aimed at mobile platforms.

Android is in fact more than an operating system as it offers an entire software stack built on top of a modified Linux distribution. The entire platform is open-source, so all layers of Android can be extended, the new image can be compiled from source and flashed on Android-ready devices. Moreover, a Java API offers the possibility to develop user applications that can be easily deployed on running Android instances.

In this chapter we will give an overview of the system architecture, list some of the tools used in code development, explain the security features of Android, and also discuss some third-party security-related applications.

2.1 Architecture overview

The Android software stack is composed of four layers (see figure 2.1): a Linux kernel, a set of native libraries, a Java application framework, and on top of that, a set of core applications [2]. A minimal understanding of these layers is required since we need our system to integrate transparently with Android.

Android is based on a modified Linux kernel version 2.6. The kernel performs core tasks (such as memory management, process scheduling) and it acts as a hardware abstraction layer for the software that runs on top of it (it offers a set of drivers for all system components). The differences from the mainline kernel mostly consist of implementing support for tasks that are specific to embedded devices, for example power management.

The kernel functionality is made available to the upper layers via a set of user-space native libraries and services. This layer also contains the so-called Android runtime: every application runs in a separate process in such a self-contained environment. This runtime loads a set of core libraries as well as a custom Java virtual machine, the *Dalvik VM* (more on

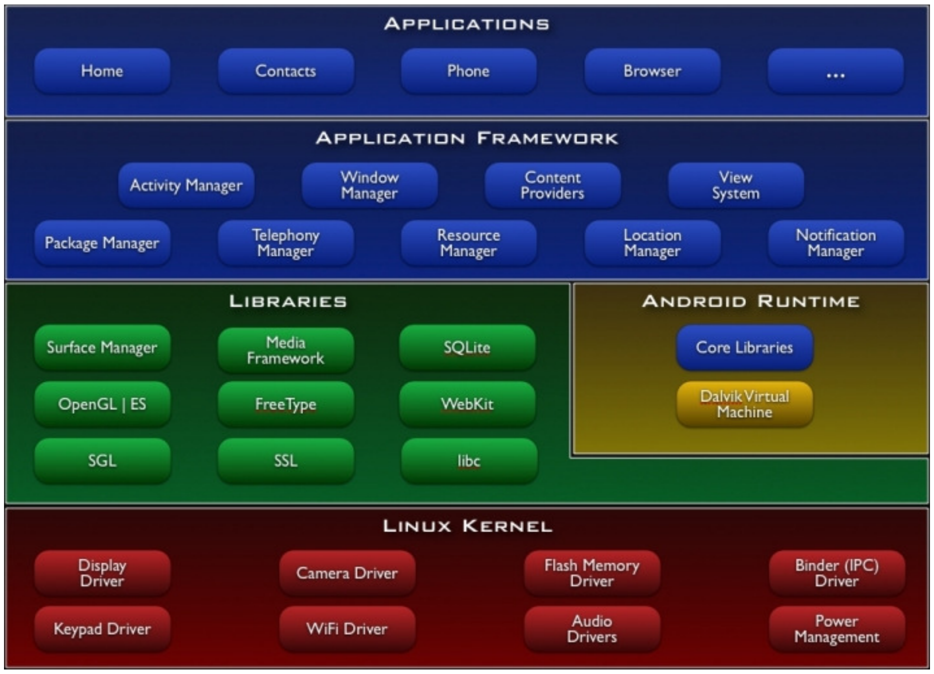

Figure 2.1: The architecture of Android [2]

this in the next sections). As a result applications are isolated with respect to accessing each other's system resources (memory and file system objects). If two applications need to communicate they can do so by using a dedicated inter-process communication (IPC) mechanism, the *Binder*.

The next layer, the application framework, consists of a set of Java libraries (jar packages) that use the native layer and extend its functionality. Communication with the underlying native libraries is achieved via Java Native Interface (JNI) calls. In the design and implementation of our system we will be working with these two middle layers (the native libraries and the application framework).

Finally, Android is delivered with a set of ready built applications such as a contacts manager, an email client, a browser and others. All these applications are implemented in Java with support from the framework layer. The applications layer will not be affected in any way by the presence of our encryption system, i.e. existing applications do not require modifications in order to benefit from the functionality of our security extensions.

2.2 The Android API and software development tools

In this section we give an overview of the tools employed during development in order to show how our system will be built and how it can be set to run on Android. Also, we offer some insight into how data can be retrieved from a device connected to a PC. We will be referring to these concepts in chapters 6 and 7 where we also provide more details on the limitations imposed on our implementation by the API.

One of the reasons behind the popularity of Android, is the support that it offers for software development in Java as well as in C/C++ (native code): an emulator for various platforms[1], a Java Software Development Kit (SDK), native compilers for the ARM architecture, debuggers and profilers. An overview of the main concepts used in application development is given below.

The Android API comes in the form of a java archive (the *android.jar* file) that the user-defined Java code links against. The Java code is compiled to byte-code, it is stored in *class* files (one such file per Java class), and these files are then merged into a single *.dex* file. The format of the dex file is proprietary to Android and it is especially crafted to run on the Dalvik VM, a register-based virtual machine (as opposed to conventional Java machines that are stack-based) optimized for low memory requirements.

For performance reason, some applications may choose to implement part of their functionality in native code. This code will be compiled with the *ARM gcc* toolchain and packed in a shared library file (a .so file) that needs to be dynamically loaded from the Java code. The dex file and the so libraries (if any) are packed in an *application package (apk)* together with a *manifest* file. The manifest lists Android specific settings, amongst which the most important are the permission of the application, the system resources that it has access to: internet access, the right to interrogate certain Content Providers and so on. This file can also specify the user under which the application runs (the Linux UID) and it can set the new module to receive special signals called *intents.*

The new apk can then be sent to a device (this can also be the emulator) via a dedicated interface, the Android Debug Bridge or *adb.* This tool runs a client application on the development machine and it connects to the target device via USB. In order for the corresponding adb service to run on the Android machine, it needs to be explicitly enabled from the Android settings menu; once the connection is established, the client can start issuing commands in order to install applications, copy files to and from the device or open a shell on the Android machine.

Opening a shell on the device allows for Linux applications to be executed from command line. However, the user has limited access to the platform since, on most devices, root access is disabled. The process by which the Android image on a device is modified such that some applications (e.g. the shell) are set to run as root is called *rooting.* There are numerous reasons for which a device would be rooted, the most common being the desire to overcome restrictions imposed by the mobile network carriers. Rooting requires for the device to be reflashed (some Linux partitions are overwritten), an operation that can be performed only if the bootloader allows it. The means by which the bootloader is unlocked vary greatly with the device: for instance, on Motorola Xoom it can be unlocked by first pushing a combination of keys that puts the phone in the so-called recovery mode, whereas on HTC Desire unlocking is blocked by hardware.

[1]This is actually an ARM emulator and the supported platforms can be configured in terms of display size and memory size.

2.3 The application development model

We have seen how to build and install applications, however we have not mentioned how to develop applications in the first place. In this respect, the Android API imposes a specific model, that all software modules need to follow; the components of this model are: *Activities*, *Services*, *Binders*, *Intents* and *Content Providers* (see figure 2.2).

There are two main ingredients to Android applications: *activities* and *services*. The main difference between the two categories of application components is that activities are attached to a user interface, whereas services are designed to run in the background; services can either be spawned from an application in order for some work to be performed in the background, or they can be long-term-running tasks that answer requests coming from remote applications. Typically, an application is composed of several interconnected activities and services.

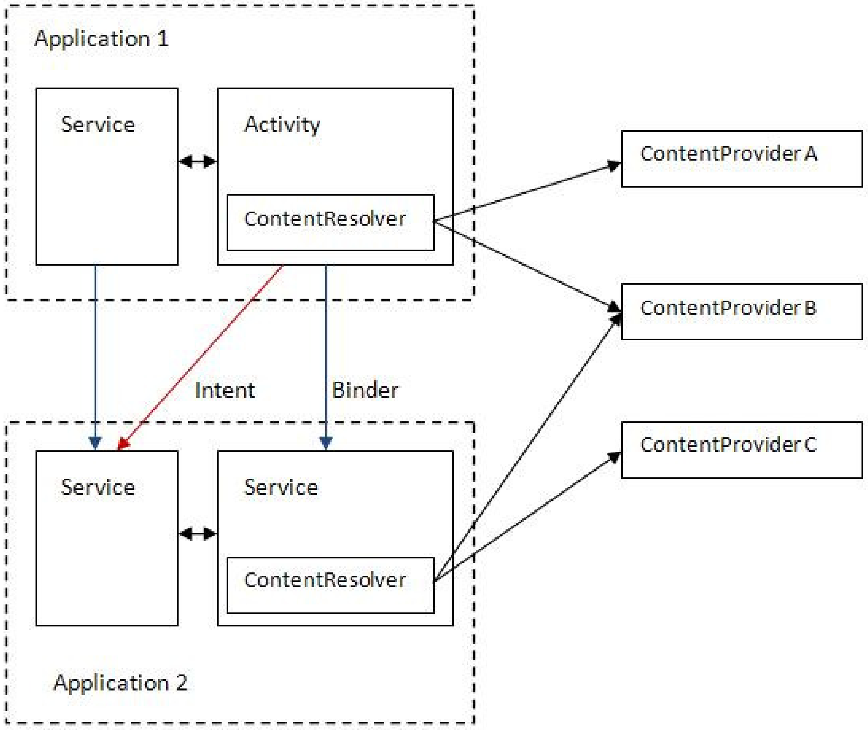

Figure 2.2: Android applications

In the case of remote services, applications can establish connections using the Binder mechanism: a service declares the methods that it wishes to export in a special format, the *Android Interface Definition Language*(aidl). A dedicated build tool generates stub java classes from these aidl files and the service needs to provide the implementation of each method. In order to call these methods, an application needs a local copy of this aidl file, and once bound to the service, it can issue *remote procedure calls (RPC)*: method parameters are packed in objects called *Parcels*, they are sent to the service in a special message format, and the application blocks waiting for the execution of the remote method to complete. The communication details of this mechanism are handled by the Binder, thus, from the point of view of the client application, the these remote calls are no different from local calls.

Android offers a second IPC mechanism, *intents*; they provide an asynchronous message passing scheme. The intent itself is an object that encapsulated the description of a task to be performed, or a notification of an event that has already occurred. Applications can both generate intents and register as receivers of intents - a special entry is added to the manifest and a user-defined callback function needs to be registered. Intents also provide the default mechanism by which applications connect to services in the first place, in an operation called *binding*: the details of this operation are not relevant, but it is important to notice that the initial connection to the service is non-blocking. Before the application can start issuing synchronous RPC calls to the service, it needs to wait for the notification that the connection has been established.

2.4 Content Providers

Content Providers are a core component of the Android software development model as they are the main mechanism by which data can be written to permanent storage such that it can be shared between applications. The Content Provider actually defines the software module, the dedicated service, that manages data access to such a shared container. The term, however, is sometimes used to denote the data container itself. More details are given in Chapter 6, but it is important to see what they do since they lie at the center of our encryption system.

The main idea behind Content Providers is that they group data by type, in a way that is convenient for the applications that operate with this data. For instance, Android is delivered with a set of Content Providers for data such as contacts, emails, browser settings and so on. Also, the API exposes the necessary mechanisms for the user to define custom containers. Another advantage of this concept is that it offers a standard way of accessing data, i.e. a set of data management functions that hide the actual implementation details. For this reason, there is no actual constraint on how data is stored and retrieved in the Content Provider, however the most convenient and the most frequently used alternative is to employ the *SQLite library* included in the application framework.

Moreover, an application that request data from a container does not have to cope with the details of locating the data and connecting to the appropriate provider. Every Content Provider is associated with a system-wide unique identifier, the so-called *AUTHORITY*. If an application wishes to access data stored under this identifier, it can do so by the use of an object, the *ContentResolver*, that is loaded by the system in the runtime environment of the application. Since the Content Provider can run in a different process than that of the calling application, the Content Resolver handles all inter process communication in a transparent manner.

2.5 Security features

Android is advertised as having been designed with security in mind [3]. There are several features that increase the level of security for applications and the most important are listed below.

Application isolation, or *sandboxing*, is a key feature of the Android model. This is achieved by the fact that, by default, every application is installed under a different UID and every Android application runs in its own process. This approach relies on the underlying Linux layer to enforce access boundaries between processes: a process cannot directly access filesystem resources that were created by a different user.

This sandboxing model comes with data access limitations, therefore, if applications need to cooperate, they will have to make use of Binders, Intents and Content Providers[2]. Android offers the possibility to set access rules for these IPC mechanisms in code or in the manifest files of applications that use them. For example, when an application sends an intent, it can specify that the receiver needs to have certain properties set in its manifest (only applications that have the property will receive the intent). Also, Content Providers can set access rules in their manifests. Binders however, are not declared in the manifest, but they can explicitly check permissions at runtime: if an application calls a remote method on a service, the service can first verify if the caller has certain attributes set in its manifest.

Manifests play an important role in setting application permissions. It is important to note that all these permissions are set in a static manner: the manifest rules are parsed when the application is set to run and they cannot be changed dynamically, at runtime. Also, the set of requested permissions is displayed when the application is installed and it is the exclusive task of the user to inspect these permissions, decide if the application is trustworthy and if it safe to give it the required privileges. The main problem with this setting is that, even if the application is known not to be malware (the vendor is trusted), it may still have bugs and thus expose certain vulnerabilities. Nevertheless, the user will grant all requested privileges to this trusted application. An ill intended application can then be installed and, in order for it not to raise suspicion, it will not ask for any sensitive privileges, but instead use the trusted vulnerable application to access whatever resources it needs.

In addition to listing all required privileges in the manifest, all applications need to be signed with a certificate whose private key is held by the developer. There are two main reasons for which signing is required: first of all, the certificate identifies the application vendor. It is however perfectly acceptable, and quite common, that the certificates are self signed; Android will not check the entire chain of trust, it will just test if the certificate matches the signature. Secondly, if two apk's are signed with the same key they are allowed to freely share system resources and they can be set to run in the same process.

[2]UNIX-type communication objects like local sockets or signals can still be used, but Android recommends using the mechanisms provided by the API.

2.6 Security applications

By the means of its core design features, Android offers application isolation such that, if a malicious application is downloaded on the device it will do as little damage as possible. In a scenario where the attacker will try to retrieve personal information from the phone, sandboxing will prevent a newly downloaded application from accessing Content Providers (if, of course, the permission is not set in the manifest, assuming that the user will not install an application that requires access to private data in the first place). This type of protection isn't of any help for an attacker who has physical access to the device and, as shown below protection against such a scenario is rather weak.

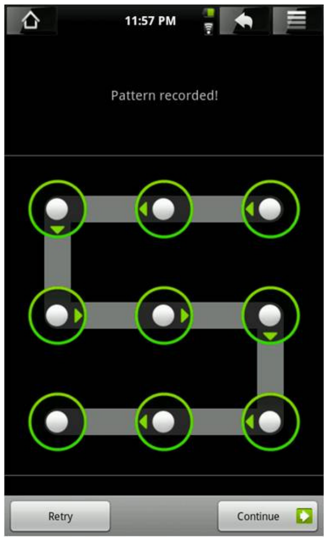

Figure 2.3: Unlock pattern

Android offers the possibility to protect the phone with a password or unlock pattern (see figure 2.3). These mechanisms are not enabled by default and in some cases they can be bypassed by simple "smudge attacks": finger traces may be visible on the touch-screen and thus reveal the unlock pattern or password. Even with a locked screen, if the debugging interface is left enabled on the device, the adb client has access to a large part of the Android filesystem. Moreover, if the device is also rooted, the entire content of the flash memory and the SD card is exposed via adb.

To cope with this issue, starting with Android 3.0, a block encryption mechanism is provided (Android 3.0 is a tablet-only version introduced in February 2011). The encryption solution is also included in Android 4.0, a smartphone version released in October 2011, but the number of devices that support this release is still rather low in comparison with the ones that use Android 2.3.

This block encryption system uses the *dm-crypt* Linux kernel encryption module and a custom frontend; this frontend will ask for a password to be introduced when the device is first encrypted (the key is actually derived from this password) and this same password needs to be provided in order to unlock the screen. This encryption module only works at block level, so it is not possible to pick only the sensitive data to be protected. The Android filesystem comes with the following set of partitions, each having a designated purpose:

- boot - the bootloader and the Linux kernel.
- system - all system components with the exception of the kernel.
- recovery - a reduced OS image that is used in order to fix problems that may appear with the main image.
- data - data used by the applications.
- cache - applications may choose to use space on this partition as a buffer at runtime.
- sd-card - the (optional) SD card.
- sd-ext - this partition has the same role as the cache partition, only that it is located on the SD card.

Out of these partitions, only data and sd-card are encrypted. This may pose a problem if some sensitive data is left on the cache or sd-ext. Also, the fact that encryption is performed at block level (the entire partition is encrypted) can incur performance issues for every data access, even when retrieving public information from the flash memory.

Moreover, the data partition is not unmounted when the screen is locked (the data partition is first mounted when the device boots). If the debug interface is enabled, the adb client will have full access to the data stored on the flash, without the need to input the password used in the encryption: the filesystem driver will decrypt data before it is passed to the adb service that runs on the device.

Another problem with the integrated encryption solution is that it offers little flexibility: the user cannot decide what data is sensitive and needs to be encrypted. There are third-party applications that tackle this problem, such as Secrets for Android [5]; they offer the possibility to enter data in a special form and use the Java security API to encrypt it with a key derived from a password.

However, the encryption solution that comes with Android 3.0 is too coarse (works at partition level) and third party encryption applications only offer protection for data that is input via the application. The problem that we are tackling in the rest of this document is to find a way to offer encryption for the dedicated data storage mechanism in Android, the Content Provider.

Chapter 3

Requirements

In this chapter we will give details about the requirements for out Content Provider encryption system. We will begin by discussing the threats that our system needs to defend against and then describe a set of usage scenarios for our security enhancements.

3.1 Attacker model

Before we set off to designing the operation scenarios, we must establish the type of threats that we are trying to mitigate. There are two types of attacks that could target an Android device: *local* and *remote*, where, in local attacks the attacker has physical access to the device (as opposed to remote attacks).

A remote attack doesn't necessarily imply that it is being run from a location outside the device; malware running on the phone can also be considered remote since it had to be downloaded from a remote location in the first place. This is the type of attacks for which application isolation as enforced by Android offers protection and we will not discuss them any further.

Our Content Provider encryption system focuses on scenarios where the attacker has full access to the device. A typical situation for which we want to offer protection is that where the phone is lost or stolen, and, as discussed in the previous section, the screen locking application is not of much help. Therefore, we need to look into solutions that encrypt the data stored on the device.

In a scenario where full physical access to the phone is granted, there are three main types of attacks, according to the Trusted Computing Group whitepapers [9]: *hack attacks*, *shack attacks* and *lab attacks*. The first category contains software attacks: the device can be connected to a PC and the ill-intentioned user will try to exploit vulnerabilities in the Android stack in order to get access to the data stored therein. Shack attacks are those that use low-budget hardware in order to read electrical signals of various ports and try to reconstruct data that is being exchanged between various hardware modules on the phone. Finally, lab attacks are the most intrusive type: it can be assumed that the attacker has the proper resources to slice chips open and analyze them at transistor level.

Given that mobile platforms are delivered as System-on-a-Chip (SoC) - the CPU together with memories and other modules are packed in a single chip - a shack attack would have small chances of success (there are not many open lines to tap into and inspect). Also, a lab attack falls out of the scope of our protection system. We will assume that data of such value that it would determine someone to perform a lab attack in order to retrieve it, should not be stored on a mobile device.

Our Content Provider security system only deals with the first type of attacks, those done by hacking. We will assume that a device is found with the screen locked and the only extra tools that an attacker will use are a PC and an USB cable. The device can also have the debug interface enabled (the user has at some point downloaded some data from the PC via USB, using the adb tool, and forgot to disable the debug interface). Moreover, the device can be rooted and the bootloader can be unlocked.

3.2 A real attack

The degree of effort that needs to be invested in performing a local attack highly varies with the device. We will give an example of how data can be retrieved from a Motorola Xoom tablet running Android version 3.2. The only restriction to our attack is that the bootloader is unlocked, but, given that this device allows for the bootloader to be unlocked and relocked from software, the attack scenario is still valid (the user may have unlocked the phone to try a custom Android image). It is important to have this assumption as unlocking the bootloader also performs a factory reset - it wipes the entire content of the data partition, thus removing all user data.

At this point the attacker can boot the device and enter the bootloader menu (by pressing the volume rocker and the power button at the same time) with the phone connected to a PC via USB. The attack then proceeds by reflashing the recovery partition with one where the shell application is installed with root privileges - this operation will not affect the other partitions, so user data is left untouched. At this point the attacker can mount other partitions and dump the entire content of the data partition, or it can change the UID of the shell application installed on the system partition such that it runs as root. This final operation would put the entire system at the attacker's disposal, even after the device is rebooted. This attacks will of course not obtain any valuable information if sensitive data is encrypted.

3.3 The encryption system

So far we have established the attack vectors that we want to defend against. We next need to establish the type of user for which we are designing the application; the tasks that the system needs to fulfil are detailed in the rest of this section.

We start by mentioning that the security system allows for multiple users to be registered and they can log in on a mobile device without having to restart the operating system. Also, the basic unit that is secured is the Content Provider, or, to be more specific, the data container managed by a provider. This means that there is a unique key per container and that encryption is atomic with respect to a container - either the entire data in the container is encrypted or no data entry at all. Also, access rights are defined per user with respect to a set of containers.

The encryption targets two types of usage scenarios:

1. *Domestic or standalone usage* - the device acts independently of any external authority with respect to the container encryption system.

2. *Enterprise usage* - all devices are registered with a *Central authority* that manages the set of users. A given user account can be declared system-wide, i.e. it can use some credentials to log in on every device and its access rights are also valid on all registered devices. If user A has access to container C on a device, it will have access to the respective container on any registered device.

3.3.1 Domestic usage (no central authority)

In this scenario, the device operates independently of any other device or central authority. Access policies are set locally via a special admin application. The incentive behind allowing several users on a device comes from the fact that some containers may be considered more sensitive than others. For example a user may want to have access to the encrypted email container only when he is at work and disable this access by switching to an account with less privileges outside office hours.

This usage scenario requires two types of accounts to be installed on the device:

1. An *admin account* needs to be created when the encryption system is first installed. The only purpose of this account is to allow for various configuration options to be set via a dedicated *monitor application*: add or delete user accounts, decide what containers to encrypt and assign these user accounts to the existing containers (allow and revoke access to the containers). This application can also set login policies, i.e. decide when the user needs to enter the login credentials (e.g. on screen unlock or at least every 30 minutes).

2. A set of *user accounts* is created by the aforementioned monitor application. In order to access any of the encrypted containers, the user needs to login with the credentials of such an account and, once the device is unlocked, data access tasks will perform cryptographic operations in a transparent manner. The user does not need to be aware that any encryption is performed in the background, he will only notice this when he tries to access some container for which he does not have the proper authorization.

The entire encryption system interacts with the user via two dedicated interfaces:

1. A *monitor application* that requires admin credentials and offers a graphical interface for all administration operations.

2. A *login application* that asks for the login credentials and unlocks the containers.

As far as the end-user is concerned, he only needs to input the proper login credentials before accessing a container and he needs to have access to the admin credentials if he wishes to modify access policies.

3.3.2 Enterprise usage

This second mode of operation is targeted at the business environment: a set of devices are connected to some remote server, the central authority (CA), and have the possibility to exchange data from their encrypted containers.

In this operation scenario user identifiers are unique across the entire systems i.e. user A on device D1 is the same as user A on device D2 meaning that they have the right to access the same containers.

The system also supports user groups; this design decision is motivated by the fact that in a business environment, users will change more often that their functions. With this feature, access rights can be configured per group and a user will always inherit the privileges of the groups to which it is assigned, but it can also be endowed with extra access rights.

The main purpose of the central authority is to impose an extra degree of control over the set of users and their access privileges and also to ensure that these access rights are enforced on all registered devices. A device does not have to be connected to the CA at all times, but it needs to communicate with the CA at given intervals in order to receive updates for the access control lists (for instance the CA may have decided that a user account is revoked, so this change needs to be propagated to all registered devices in order to prevent the respective user from further accessing information). This means that any change in access policies can only be done when the device is connected to the CA.

Devices also have the possibility to share encrypted content i.e. some entries can be exported in an encrypted form and assume that the receiving device has the proper mechanisms to decrypt it.

It is desirable to also have the possibility to declare local users (user accounts that are only known to the local device and are not valid on other devices in the system). This can be useful if, for instance, the device is used for personal purposes outside the business environment; a local user account will be declared such that it allows only for non-critical encrypted containers to be accessed. The CA still needs to be informed when such a user is created, in order to verify the type of access the new account requires, but it will not inform the other devices of this change.

Similar to the previous scenario, there are two main type of accounts: an *admin account* and a *regular user account*. Their purposes are the same as in the case of the domestic type of usage, with the exception that the admin account is also responsible for the proper setup of the communication with the CA. Also, local users can be created on the device using admin credentials, in which case the proposed change needs to be first approved by the CA. Regular global users are however only created and revoked by the CA and broadcasted to all registered devices[1].

The device also needs to have a *monitor application* and a *login application* for regular users, which need to fulfil the same tasks as in the case of the domestic usage scenario. In the enterprise scenario, there is a third application in the system, the CA; it can be implemented as a web service and it needs to be aware of the configuration of all access lists on all registered devices. This service also needs to handle change requests from registered devices and broadcast these changes to the other members.

3.4 Observations

In this chapter we only offered an overview of what is expected from the encryption system. However, some implementation challenges can already be pointed out. For example , devices are able to share data, which means that they need to also share some keys. If these keys are exposed, they need to be changed on all devices registered with the CA. This operation can be expensive and also, retrieving the key from a device can make all registered devices vulnerable. In conclusion, we need a better mechanism to store these keys such that they never leave the device, even when running with a corrupted Android OS.

[1]The devices need to check for updates on a regular basis. If the local configuration is outdated and the device does not connect to the CA, the encryption system will simply refuse to perform any data decryption operations.

Chapter 4

Hardware-enhanced security

A complex software system like Android is prone to vulnerabilities. Given the fact that Android is also relatively new, and that it runs on a multitude of devices with (slightly) varying hardware architectures, it is quite likely that it hides a set of exploitable faults. The main idea behind hardware security enhancements in general is that they move some sensitive code or data out of the exclusive control of the (possibly faulty) OS. Regardless of the specific implementation, such a security mechanism will isolate code or data in a hardware-protected location. No assumption can be made on the degree of protection Android offers for the content of the internal memory, therefore, the following scenario needs to be taken into consideration: a malicious user can dump and investigate the content of the entire flash memory and even take snapshots of the address space of random processes.

In this section we will be looking at what the hardware security enhancements have to offer and how these mechanisms can be employed in our encryption system. More precisely, since we want the devices to be able to perform crypto operations locally, we need to guard a set of encryption keys as best as possible. In other words, we need a hardware mechanism that will *hide these keys*: the hardware prevents this data from ever being exposed to the operating system.

In order to actually use the hidden encryption keys there are two design possibilities. In the first case, the cryptographic operations are implemented in hardware on the same chip that guards the keys and they run in order to serve encryption requests that come from outside the chip. The second choice is to ensure that the hardware creates an isolated virtual environment, outside the operating system, where some code is allowed to run and use these keys. In both cases, a *trust* relationship needs to be established: in the first case the encryption chip needs to trust the external components that send requests. In the second case, the actual code needs to be trusted i.e. it needs to be implemented such that it does not leak any valuable information to the outside world (outside its virtual environment) and also, the hardware platform needs to guarantee that the code has not been changed by a malicious user. More details on how trust is achieved are given later in this section.

It is important to note that hardware solutions alone are by no means expected to eliminate all vulnerabilities from a system - the mere fact that part of the execution path moved in hardware, or is protected by hardware, does not mean that the rest of the code should be less rigorous with respect to security (a hardware implemented decryption function that can be called by any user offers just as much protection as not having any encryption at all). These hardware security enhancements, if properly employed, only offer an extra layer of security to a software system built with security in mind.

In the next sections we will give an overview of three technologies that address this problem, and that correspond to three different approaches: the Trusted Platform Module (TPM) proposes an all-in-one solution (a separate chip that handles all your sensitive operations), Intel Trusted Execution Technology (Intel TXT) combines the virtualization support of the CPU with a TPM in order to secure its virtual environments, and, finally, ARM TrustZone moves the focus on processor-supported virtualization and removes the TPM chip entirely .

4.1 Trusted Platform Module

The Trusted Platform Module (TPM) is the most well-known of the hardware security solutions, mainly due to its popularity on the PC market - since 2006 it is common for laptops to be delivered with such a TPM chip. It is also used extensively in disk encryption systems such as Microsoft BitLocker [1].

The TPM is a hardware device compatible with the requirements of the Trusted Computing Group [10]: security, privacy, interoperability, portability, controllability and ease-of-use. This document is accompanied by a set of specifications comprised in a design principles document [19], a description of the structures used by the TPM [20] and the set of supported commands [21]. These documents comprise an architectural description, the functional requirements, and standardize the TPM to host device communication.

The most important features of the TPM are its ability to:

- Store and hide security tokens (passwords, certificates and encryption keys). These tokens never leave the chip in plain-text form, they can however be packaged and exported for use on other TPM chips.
- Offer various cryptographic functions implemented in hardware. In order to ensure interoperability, there is a minimal set of algorithms and protocols that all TPM chips need to implement: RSA, SHA-1, HMAC.
- Store platform measurements in so called Platform Configuration Registers (PCR), thus ensuring that the host device has not been tampered with. There are two types of PCRs: static PCR values can only be reset upon platform boot[1] whereas dynamic PCRs can be reset during system runtime - measurements can be taken across all software layers, these values are hashed and combined with the hashes stored in the static PCRs; these new values will reflect the changes that have occurred in the system since the last reboot. The static PCRs are used by BitLocker: when

[1]This occurs during the initial boot sequence that runs from BIOS.

the system is installed, some code in BIOS will take a measurement of the system configuration, hash this value and send it to the TPM. The TPM will store this hash in a PCR, seal a secret using this value, and only reveal the secret in the future if the same hash is provided.

These features can be easily employed in order to provide access control for common security-related processes such as digital signing - the software will never have direct access to the signing key and the TPM will refuse to sign some document if the user is not properly authenticated with the TPM or if the state of the platform has changed without the TPM having been noticed.

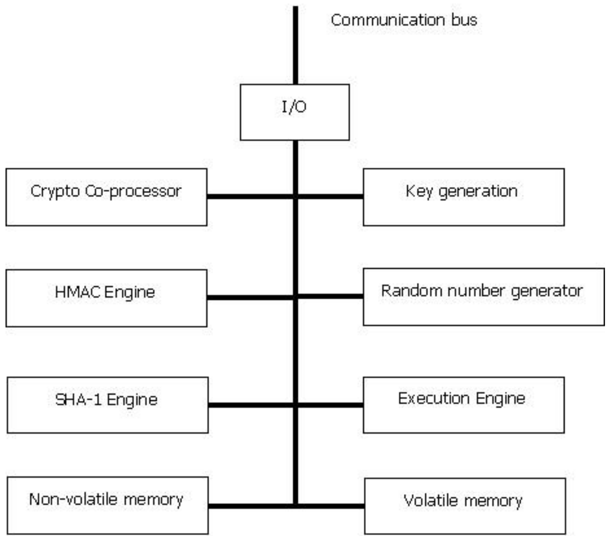

Figure 4.1: TPM Architecture

The functionality of the TPM chip is reflected in its hardware architecture, as depicted in figure 4.1. Note that every supported algorithm corresponds to a separate silicon area. This also means that the more functions are supported, the more space and power the chip requires, which makes it incompatible with the restrictions of mobile platforms. This observation is confirmed by the fact that, while the TPM technology has been widely adopted in the PC industry (100 million devices with TPM chips sold in 2007, according to the Trusted Computing Group [22]), there are hardly any mobile devices that offer it.

4.2 Trusted execution

Several CPU vendors have developed extensions to the TPM in order to provide support for trusted hardware-enhanced virtualization. The aim of these technologies - Intel TXT (Trusted Execution Technology) and AMD Secure Execution Mode - is to provide the means to create a trusted execution environment on top of the normal execution mode. In other words it is possible to have a trusted application launched from a possibly tainted operating system. They do so by having the CPU work in conjunction with a TPM chip such that the state of the system can be checked for consistency against a previous configuration that is known to be trusted before certain operations are performed.

Intel TXT technology [17] creates so-called Measured Launched Environments (MLE) that only run if the system is in a trusted state. Intel TXT can launch MLEs at any time without having to reboot the platform; it does so by making use of a specific CPU instruction and the dynamic registers in the TPM. This instruction, *SENTER*, triggers an interrupt that acts as a reboot without actually reseting the platform [15]. This instruction broadcasts messages to the chipset and other (physical and logical)processors in the platform that, in response, perform basic cleanup.

When a MLE is set to run, two code blocks are loaded in memory: the MLE code itself and an authenticated code module (AC) that is digitally signed by the chipset vendor. This AC will test for various chipset configurations and if the platform is found to have an acceptable configuration, it measures the MLE code and passes the control to this code block.

Intel TXT can only work in conjunction with a TPM chip that it uses in order to store measures of the MLE code; it also makes use of the Intel Virtualization Technology for Directed I/O (VT-d) in order to make sure that the MLE code runs in an environment that is isolated and protected from DMA accesses (this technology allows for specific physical memory pages to be blocked).

In a nutshell, this type of hardware offers the possibility to run trusted code on top of an untrusted software platform: this code cannot be corrupted in an undetectable manner and it will run in its own safe virtual environment. It is, however, important to notice that the system only ensures that the running code has not been tainted. It is the programmer's duty to make sure that the code cannot be exploited during runtime. This can be achieved by making the attack surface as small as possible and by maintaining a small trusted code base that can be easily inspected for vulnerabilities.

Since Intel TXT requires the presence of a TPM chip, it cannot be ported entirely to a mobile processor. However, the main principle of operation, secure hardware enhanced virtualization, does not require the entire functionality of the TPM and can be implemented in a lightweight form on mobile processors.

4.3 ARM TrustZone

Given the space and power constraints on mobile devices, having a dedicated security processor is infeasible. The alternative offered by the TrustZone technology is to have the CPU create a protected virtual environment: all system resources are split between a so-called *Normal world* and a *Secure world* (hardware enhanced virtualization) [9].

Any CPU that has a Memory Management Unit (MMU)[2] and offers at least two operation modes (a normal mode and a protected mode) can be used to implement software

[2]A hardware component integrated in the CPU that handles physical to virtual address translations.

virtualization. A special process, the hypervisor, runs in the protected mode and isolates memory accesses and possibly other IO resources amongst the processes under its control. In what security is concerned, without proper hardware support, other modules that control the BUS (such as the DMA or the GPU), can bypass the hypervisor.

The main idea of the TrustZone is that it moves this hypervisor in hardware and redesigns some of the platform components such that two virtual environments are created, and one of them is not only isolated, but also trusted (i.e. the code running in this environment has not suffered any unauthorized modifications). This privileged virtual environment is called the *Secure world*, whereas the other one is referred to as the *Normal world*.

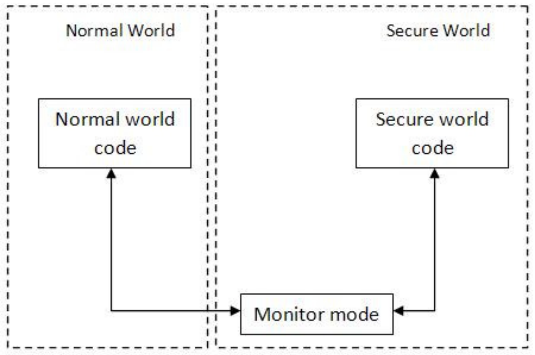

Figure 4.2: The TrustZone virtual cores [9]

4.3.1 Hardware design

ARM platforms and in general mobile solutions are delivered as System-on-a-chip (SoC) - the main CPU together with memory controllers, some RAM and flash memory modules, other dedicated processing units and the main bus (see figure 4.3). The changes required by the TrustZone occur in several of these components as follows:

- There are two main bus lines on the platform: a system bus, the *Advanced eXtensible Interface (AXI)*, and a peripheral bus, the *Advanced Peripheral Bus (APB)*. Both of these are modified such that no Secure world resources can be accessed by the Normal world components.
- The CPU runs code from both worlds in a time-sliced fashion.

The extended BUS comes with an extra control signal for the channels of the main system bus, a Non-secure(NS) bit. The peripheral bus (APB) itself does not support this extension in order to maintain compatibility with legacy peripherals, but the AXI-to-APB bridge (see figure 4.3) ensures that the NS bit is properly handled when communicating outside the AXI bus. All bus masters set this signal when making a transaction and the logic in the secure slaves checks for security violations. This is equivalent to having 33-bit addresses that map over a 32-bit physical address space for secure transactions and a similar address space for non-secure transactions (possible cache coherency problems need to be handled by the operating system).

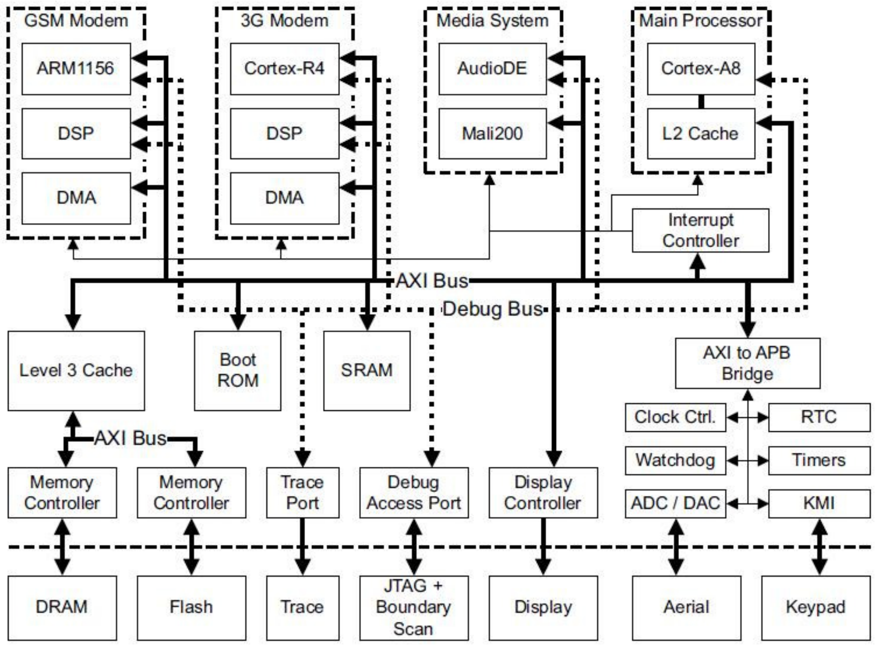

Figure 4.3: The architecture of an ARM SoC [9]

The CPU provides the two virtual cores and a context switching mechanism, the so-called monitor mode. For each data access operation the NS bit on the bus is set according to the currently running core: the Non-secure virtual core will only be allowed to access Non-secure system resources, whereas the Secure core has access to all resources. The two cores execute in a time-sliced manner and all context switches pass through the monitor mode: a specific interrupt puts the processor in this mode and the code in the monitor (always running in the secure world) will handle the switch.

Each of the two virtual processors is equipped with a virtual MMU, therefore each core can independently manage address translations. Translation tables descriptor formats include an NS (Non-secure) field - the secure virtual processor uses the field in virtual to physical address mappings (in order to access non-secure addresses), whereas the non-secure core ignores the field. Entries in the Translation Lookaside Buffer (TLB) also include the NS-bit, allowing addresses from both worlds to coexist in the same tables and enabling faster context switching (no need to flush the tables). Also, any attempt to access a page with NS set to zero (a protected page) from the Normal world would result in a processor fault.

There are two possibilities for the Normal world to communicate with the Secure world. The first mechanisms uses *interrupts*: the Normal world calls a dedicated CPU instruction that results in an interrupt to be triggered in the TrustZone monitor. An interrupt table set in the monitor will then point to the appropriate handler code that runs in the Secure world. The underlying mechanism uses interrupts, but from the point of view of the developer, this communication scheme behaves like Remote Procedure Calling where parameter values are placed in the CPU registers and passed to the Secure world.

Since the secure world has access to the entire address space, the second communication mechanism uses *shared memory.* An application can be designed to allow the Secure world to directly map non-secure memory pages. The Normal world will write data to the shared memory area, the Secure world will perform some operation and return the new data in the same memory area. This is achieved with no loss of performance since this memory sharing scheme can occur at any level in the memory hierarchy: the applications in the two worlds will even share the same cache lines.

So far we have established that the resources of the Secure world are out of the reach of the Normal world. However, an attacker can attempt to replace the Secure world on-flash software image on a powered down device. At boot time the platform needs to check the Secure world software for authenticity. This operation is performed by the bootloader stored in the on SoC ROM: a hash of the Secure world image is signed with a vendor private key and verified with the matching public key stored on the device. The key pair is generated at device manufacture (not unique per vendor) and the public key is stored using a one-time-programmable (OTP)[3] memory. To be more precise, the on SoC ROM has a small size so it will only verify a second level bootloader that in turn verifies the rest of the code. This mode of operation establishes a chain of trust rooted in the OTP key and it can be extended to an unlimited tree of applications that need to be verified by their parent before they are allowed to run.

4.3.2 Software for the TrustZone

There is no standard software architecture for TrustZone enabled platforms, but, as ARM suggests, there are two main design options:

- A dedicated operating system running in the Secure world in parallel with the Normal world OS. Concurrent execution can be simulated in the Secure world and complex tasks can be split across intercommunicating processes.
- A synchronous library: sensitive operations can be implemented in the Secure world and they would perform tasks only when requested by the Normal world.

However, ARM has designed a software API (TrustZone API) to standardize communication between the two worlds. It enables a Normal world client to authenticate with and send commands to a security service and also use shared memory for data exchange.

There are no restrictions on how to implement the software that runs in the Secure world, however, a rule of the thumb dictates that the code should be kept small and it should maintain a small interface for the outside world (reduce the trusted code base and the attack surface).

4.3.3 Related work and Implementations

The TrustZone technology is deployed with the ARM Cortex-A processor family that in turn are integrated in some of the best-selling mobile platforms (Qualcomm Snapdragon,

[3]This memory is based on fusible resistors that can only be burnt once such that the value stored on this chip cannot be changed by any means.

based on Cortex-A8, was market leader in 2011). However, there is a small number of software solutions that make use of this technology.

A first solution is offered by Green Hills Software; they extended their Integrity Multivisor to use TrustZone [11]. This solution offers a hypervisor that comes with its own real-time micro-kernel, such that all other operating systems run on top of it. On TrustZone-enabled platforms, Android can run in the Normal world whereas the hypervisor, together with security-critical code will run in the Secure world. The Normal world will act as a virtual machine under the control of the Secure world. The hypervisor provides an implementation of the TrustZone communication API, i.e. it enables the Normal world software to sent requests to the Secure world. However, no ready built Secure world software is offered.

A second solution was proposed by Giesecke & Devrient in 2010, their Mobicore platform for Qualcomm Snapdragon processors [14]. This is a full fledged operating system that runs in the Secure world and handles the execution of secure applications called trustlets as well as the communication with the Normal world. The OS comes with an open API that allows for the creation of user defined trustlets; a platform can run Android in the Normal world and use custom tasks running on Mobicore to perform security-sensitive operations requested by Android applications.

Both aforementioned solutions are commercial. There are several academic implementations such as the one proposed by Santos et al.[18] - they design a .NET-based custom framework that allows for parts of an application to be run in the TrustZone. The sensitive parts of a .NET solution can be moved in a class called a trustlet and the framework takes care of running this class in the Secure world inside a minimal language runtime based on the .NET Micro Framework. The two worlds communicate via a secure procedure call mechanism that also handles context switches (trigger an interrupt, enter monitor mode and have the monitor jump to the appropriate interrupt handler in the trusted framework driver). The application however is meant to be a proof-of-concept and it does not yet support shared memory. Data exchange between the two worlds is only done via CPU registers, which is a serious limitation if the TrustZone is used by applications that operate with big data blocks.

4.3.4 Observations and Conclusions

TrustZone technology is delivered with many ARM chips, but it is often left disabled by the firmware. There are however some development boards with full TrustZone functionality, such as the Freescale iMX53 [13]. Also, ARM Development Studio 5 [4] offers full emulation of this technology.

It is also important to notice that the TrustZone only offers protection at runtime. If an attacker had the proper resources to extract the flash chip from the device and inspect its content he would find the Secure world software image unencrypted and he would be able to retrieve the encryption keys. This is what the TrustZone whitepapers define as a lab attack and are considered beyond the scope of an encryption system for mobile devices - if some data were valuable enough for some attacker to go through this amount of trouble to retrieve it, this data should not be stored on a mobile phone in the first place.

The implementation of a software module for the TrustZone is a complex task and it falls out of the scope of this thesis. However, given the attention that this technology has drawn over the past years, we chose to design the Content Provider encryption system around this concept, i.e. some sensitive operations are considered to be executed in the Secure world.

We will make the assumption that some encryption keys cannot be retrieved even if the Android instance installed on the machine is corrupted. An attacker would use root privileges to dump flash and RAM content, but the keys would be stored in secure memory areas and the Normal world Android would not be able to access them. Also we designed the system such that all sensitive operations were moved to a separate service and we modified the libraries used by the Content Providers to forward all data encryption(decryption) requests to this service using shared memory. Porting this implementation to a TrustZone-enhanced OS would only require the addition of the proper TrustZone API calls that switch the CPU from the Normal to the Secure world mode.

Chapter 5

System design

This chapter discusses technical details of the Content Provider encryption system i.e. the exact means by which encryption keys and user configurations are stored and managed and how data is encrypted.

This system is designed with the TrustZone technology in mind. To be more precise, it is assumed that some components will run in the Secure world and they can only be accessed from the Normal world in a manner that is controlled by the hardware platform. However, as shown in Chapter 4, this technology is disabled by the firmware on most devices, and the system should work under these conditions as well, at least in the domestic mode (where no communication to a CA is required and exposing a container encryption key would not cause damage to more than one device). For this reason, some data structures that are only accessible to the Secure world (and are therefore protected by hardware mechanisms) are nevertheless encrypted. In other words, our system offers the strongest security guarantees under the assumption that the TrustZone is active, otherwise, if the OS is corrupted[1], the encryption system is also vulnerable.

In the design of our system we will assume that the enterprise mode only works with TrustZone-enabled devices. Certain restrictions, that will be pointed out in this chapter, constrain us to only use this operation mode if the hardware platform supports the TrustZone technology. On the other hand, the domestic mode can be installed on devices that do not have a TrustZone, but in this case there are some vulnerabilities that will also be discussed.

5.1 Local applications

There are two main software modules that run on the mobile device and ensure data access: a *monitor service* and a *cryptographic service*. The monitor service is in charge of all account management operations: it ensures that the user provided the proper authentication credentials and that it is entitled to access the container that it requests. The cryptographic service only encrypts or decrypts data as requested by the user and is under the control of the monitor service. This means that it is only given access to the

[1]More precisely, if a malicious process finds a way to inspect the memory image of the process that manipulates encryption keys, these keys can be revealed.

necessary keys by the monitor service and, as long as it is not notified by the monitor that it must no longer answer queries coming from the user applications, it will do so without any extra access verifications. These two services will run as threads in the same process, thus there is no IPC mechanism employed when they need to share data. In fact, they could be merged into one single service. The decision to have two such services is only motivated by the wish to obtain a clean design, where each service corresponds to one of the major tasks of our system: manage users and encrypt/decrypt data.

Ideally, both these services would run in the Secure world. However, if the device is not endowed with this technology, the container encryption key will appear at some point in the memory image of the cryptographic service and, since there is no hardware memory protection, an attacker with root privileges will be able to inspect this image. At this point the system will have to rely on the fact that the service cannot be brought in a running state unless the end-user introduced the proper login credentials. In other words, without a valid password, that is not stored anywhere on the device, encryption keys will not be loaded in memory. Even with the limitation of not having a TrustZone, our encryption system offers better functionality than the default Android encryption: access control can be configured and screen locking will prevent the adb from reading decrypted data.

5.2 Control structures on the device

In this section we give a detailed overview of the internal mechanisms that manage the users and access control lists on the device.

5.2.1 High-level overview

On-device access control is based on a hierarchy of *Critical Data Blocks (CDB)*[2] associated to users, groups and containers. Every user is mapped to one *User CDB (UCDB)* and to a number of *Group CDBs* (if the user is part of one or several groups). Group CDBs have the same structure and are employed in the same manner as the User CDBs, i.e. they give the list of containers that every user account is allowed to access. In a similar manner, every encrypted container is associated with one *Container CDB (CCDB)* that holds the encryption information (encryption keys) for exactly one container.

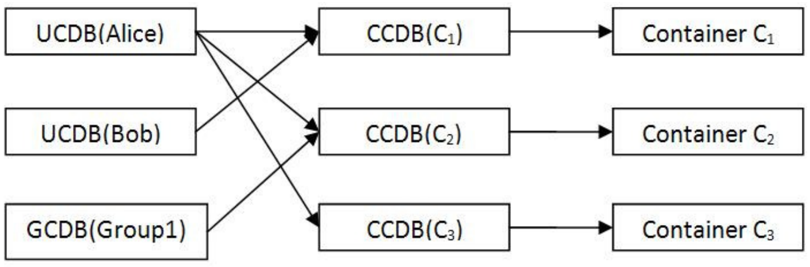

Figure 5.1: Control structures

[2]This name is used in the FreeOTFE [12] disk encryption system, where such a structure is associated with each encrypted drive. Our system uses Content Providers instead of drives as the encrypted unit and, also, these blocks have an entirely different structure.

The purpose of this three tier architecture depicted in figure 5.1 is to allow for user addition and revocation without the need to re-encrypt the data stored in the containers. Also notice that in this design, the encryption keys stored in the Container CDBs are the most sensitive (they create the most damage if exposed) and for this reason they must never leave the Secure world. They are never explicitly returned to the user requesting to access data, but instead, a dedicated service running in the Secure world will handle cryptographic operations on behalf of the user.

There is a second reason for which the container encryption keys need to be strictly guarded: in enterprise mode, our system offers the possibility to *share encrypted content* amongst registered devices. The easiest way to implement this data sharing scheme is to actually use the same key for a given container on all devices (e.g. contacts are encrypted with the same key); these keys will be further referred to as having *global scope*, as opposed to keys that are only valid on a given device and therefore have *local scope*. These keys are set on the device in a special setup phase and they are only used from the Secure world. In the enterprise usage scenario, these keys are also known to the *Central Authority* that, as a result, can choose to broadcast encrypted information to all devices (a new contact entry needs to be made available to all registered devices).

For this data sharing scheme to work (more precisely the usage of system-wide keys), it is important to point out that the system setup phase is critical and it can only be performed in a safe environment. Again, the TrustZone provides the proper support for this trusted setup: a system administrator will have the signing key for the code and data that is written to the TrustZone[3] and it is only this person that can change this data.

Another advantage to this simple data sharing scheme comes from the fact that no extra, possibly expensive, cryptographic operations are performed before data is exported, which is an advantage with respect to time and energy consumption.

With this CDB structure in place, and assuming that the device makes use of the TrustZone to perform the sensitive cryptographic operations, we can extend the functional requirements of our system. If a user was revoked, this user can access restricted content only for a limited time, before the revocation announcement is received by the phone. After the revocation decision has been enforced on the system, the revoked user has no further access to protected content as the container encryption key was never revealed to the user in the first place.

5.2.2 Control structures in detail

Container CDB (CCBD)
The purpose of this structure is to allow for easy user addition and revocation without the need to change the master encryption key. This is also the most sensitive structure in the system since the master keys are are set at install time are never supposed to change. The structure itself contains the *Master key (MK)* that is used to decrypt container data

[3]Recall from Chapter 4 that the TrustZone comes with a public key stored on a OTP chip and that only the entity that holds the corresponding private key can push valid data to the Secure world.

and a hash of that value; this structure is encrypted with *$KSM_{CCDB}(C)$*, a symmetric key with local scope generated by the monitor service.

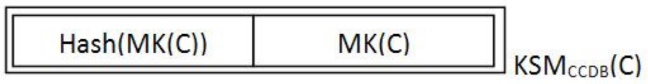

Figure 5.2: Container critical data block

The purpose of *$KSM_{CCDB}(C)$* is to offer protection if the system runs without a TrustZone - the master key is never stored in the flash memory unencrypted, such that if an attacker has access to the content of the entire flash memory, he will not be able to find the key in plaintext. Finally the purpose of the hash is to check that the provided *$KSM_{CCDB}(C)$* is correct: when the CCDB is decrypted, check that the hash of the decrypted master key matches the stored hash.

User CDBs (UCDB)
These structures form the second layer in the key storage scheme (see figure 5.3); they are control structures that hold all the *$KSM_{CCDB}(C)$* keys for all the containers that a user has access to. Every user is associated to such a structure as well as every group of users. When the user logs in, the monitor service will find the UCDB for the current account and, in enterprise mode, the GCDBs for all groups in which the user is a member; the monitor then sends these structures to the crypto service. From this point onwards it is the task of the crypto service to use this structure and load the appropriate container master keys (MK(C)) when some data request is received. The idea behind this mode of operation is that the master keys are only loaded in memory when they are used (alternatively, the monitor service could send the master keys to the crypto service, but this would mean that the master keys are at all time in the memory image of the monitor service).

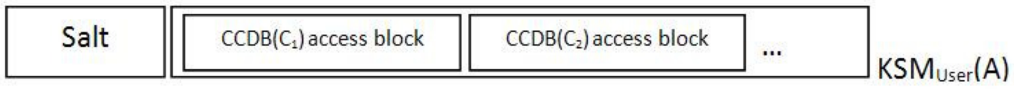

Figure 5.3: User critical data block

where a CCDB(C_i) access block has the following structure:

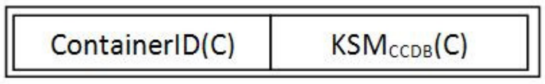

Figure 5.4: User access block

The UCDB itself is protected by $KSM_{User}(A)$, a symmetric key derived from the user password and a random cryptographic salt - the key is obtained directly from the password entered in the user interface and the derived key will be sent to the crypto application along with the UCDB. This key also provides the authentication mechanism for the corresponding user: if the crypto service fails to decrypt the UCDB[4] with the $\text{KSM}_{\text{User}}(\text{A})$ value that it received from the monitor service, it will not answer any incoming data requests.

The password is never stored in plaintext so an attacker with full access to the flash memory would not be able to obtain the container master key only by inspecting the data stored there. However, with full memory access and a known password, nothing stands between the attacker and the master keys.

Finally, groups are only available in enterprise mode: they contain keys that are used by several users, thus the encryption key for the GCDB structure cannot be derived from the password of one user. Also, when running without a TrustZone, it is not safe for the GCDB encryption key to be placed in persistent storage. However, in enterprise mode, when making use of a TrustZone, GCDBs do not have to be encrypted since they will never leave the Secure world.

Monitor TTL(Time to Live)
This is a special token that dictates the points in time when the monitor service needs to connect to the Central Authority to retrieve configuration updates. This does not necessarily have to be a time interval, it can also be a given number of login operations performed by the user (or a count of some other operation).

This validity expiration mechanism must not only notify the monitor that it needs to update, but rather force it to do so. For a TrustZone-enabled device, a TTL token can be stored in a region controlled by the Secure world and is therefore unforgeable (from the Normal world). However, for a platform that does not support TrustZone and runs a possibly corrupted Android, storing this token can be problematic[5]. For this reason we will restrict the usage of our encryption system to only run in domestic mode for platforms that do not support the TrustZone (there is no need to update the control lists in this case).

Key wallet, ACL and user groups
These two structures are only used by the monitor service and they centralize all the access rights. The key wallet is a list of Container CDB encryption keys ($KSM_{CCDB}(C)$) and in the access control list each user is mapped to a list of containers that it is allowed to access. Both these structures are protected by a key derived from the *Administrator password* that is configured at install time and is never stored in plain-text on the device.

[4]The CCDB holds a "magic value" in a separate field and this value can be checked after decryption.

[5]It could be stored in a file that contains the token and an HMAC of the token. In order to do this, the key for the HMAC needs to be stored somewhere in the flash, but, since we agreed that the entire content of the flash memory can fall in the hands of the attacker, the key can be retrieved, rendering the whole validity checking mechanism useless.

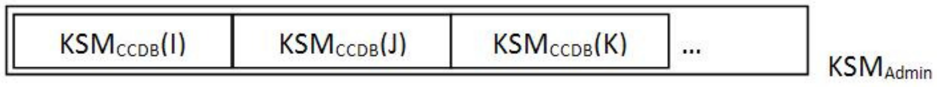

Figure 5.5: Key Wallet

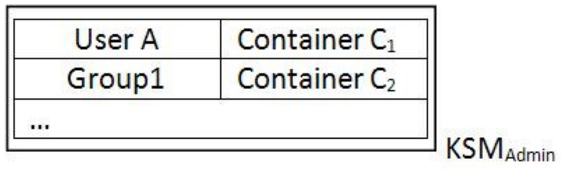

Figure 5.6: Access Control List

In enterprise mode, these structures are synchronized periodically with the CA and are also used to create and propagate changes to the User CDBs in a lazy manner (see details in the Common operations section).

The monitor service also manages a user groups structure, a mapping that contains the list of members of each group. This mapping is used in enterprise mode by the monitor to send the appropriate GCDBs to the cryptographic service.

5.3 Communication mechanisms

In the first half of this section, we discuss the communication protocols employed when the device connects to the CA for updates or when devices share encrypted container entries. This type of communication only pertains to the enterprise mode, so we can also assume that there is a TrustZone available. In the second part, we detail the mechanisms used in on-device communication: client applications running in the Normal world exchange data and commands with the security services that run in the Secure world.

5.3.1 Network communication

Device to CA message exchange occurs when the device connects to the CA in order to ensure that its access policies are consistent with those stored on the CA. In this case, secrecy does not necessarily have to be enforced, but the communication protocol needs to ensure that the communicating parties are properly authenticated and that the messages are unforgeable. Authentication needs to be established both ways: the device must only receive messages from the real CA, otherwise an attacker could inform the device that a fake user was created and use that new account to access some restricted data. Also, the CA needs to be aware of the state of the entire system, therefore it must know that acknowledgement messages come from the real registered device.

For this task, a simple protocol can be devised: a message can be sent along with a HMAC that uses a secret key shared between the Secure world of the device and the CA (this key can be set at install time). However, given the fact that this type of communication doesn't occur very often, the extra time and power consumption of a more complex protocol like TLS [16] would not be a problem. Also, TLS is known to be stable, it is widely used in practice, and the necessary configurations can be easily done on our system: again, the Secure world on the device and the CA can share a pair of certificates when the system is first installed on the mobile device.

There is also the case when devices exchange content from encrypted containers. In this case we rely on the fact that it is only the devices that know the container encryption keys, so container entries can be directly copied (encrypted with the container master key) and sent over the network with no extra encryption. This model can also be extended such that the CA can broadcast container entries to the registered devices - the CA also needs to have the encryption key for that given container.

5.3.2 On-device communication

This data exchange scheme applies to the enterprise mode of operation, where a TrustZone is available. Client applications that request data access operations run in the Normal world and they need to communicate with the monitor service and the cryptographic service that run in the Secure world.

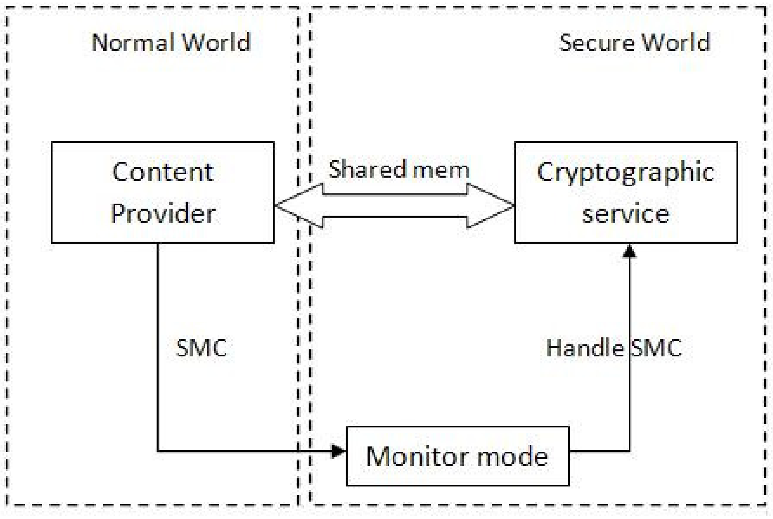

Figure 5.7: Communication with the TrustZone

For this task we will use the two communication mechanisms offered by the TrustZone technology: *interrupts* and *shared memory*. Before a Content Provider can access data for the first time (in this login session), it has to agree on a shared memory area with the cryptographic service (see figure 5.7, where the Monitor mode corresponds to the special CPU mode offered by the TrustZone, not the monitor service). This is done by the use of interrupts: the Normal world application will reserve a memory area, place the address of this area in a register and trigger an interrupt[6]. The interrupt will be handled in the Secure world by the cryptographic service that will map this memory region in its address space and use it to receive data requests.

This memory area is used by the client Content Provider to copy bulks of data that needs to be decrypted/encrypted and the cryptographic application returns the result in the same buffer. The two communicating process will synchronize[7] via semaphores that can be placed in the same memory area.

If the TrustZone is not available, this would reduce to a scheme with two processes that use remote procedure calls and shared memory to communicate. The mode of operation remains the same, only that there is no Normal to Secure world context switch that accompanies every RPC call.

5.4 Common operations

In this section we clarify the mode of operation by describing some common tasks that occur in the system when installed as an enterprise application (the tasks performed in the domestic mode are a subset of the tasks that occur in the enterprise mode).

5.4.1 Installation/registration

The system is installed on the device only in the presence of the CA to which it connects via a trusted medium (e.g. a direct connection via USB). The mobile device needs to receive the keys for container encryption, the admin account is configured and also, the set of user account that exist in the system are configured on the device. The device downloads the access control lists and creates the UCDBs for all existing users and groups. In this step also the containers are encrypted.

5.4.2 Login and access data

In this encryption system there can only be one user logged in at a given time. The login procedure is handled by the monitor service, and, if the user changes, the cryptographic service needs to be notified such that it can suspend all operations requested by the previous user. The monitor service will compute $\mathrm{KSM}_{\mathrm{User}}(\mathrm{A})$, the key that secures the UCDB for that given user, and send this key to the cryptographic service. The cryptographic service will verify that it can decrypt the UCDB and, if this operation succeeds, it is ready to receive data request connections. It will serve these request as long as the user does not change and the session time for the user has not expired (the monitor can set a login validity interval, after which the user needs to enter the login credentials again).

As long as the user is logged in, the cryptographic service will answer all requests for

[6]This interrupt is done via the *Secure Monitor Call (SMC)* CPU instruction. The Normal world can place arguments in the CPU registers and trigger this instruction that, in turn, will yield the processor to the Secure world.

[7]The client application needs to signal that fresh data needs to be processed and the cryptographic service will signal that the data is ready.

containers to which the user has access: it will receive interrupts that set shared memory areas and then continue to decrypt/encrypt data using these shared buffers. When the login session expires, the monitor service will signal the cryptographic service that it must block until it is again notified to resume operation.

5.4.3 Create user

These operations can only be initiated by the Central Authority. This is a typical enterprise setting: if a new user needs to be added, the central administrator is notified, that person will create the necessary updates and broadcast them to all registered devices. The CA sends the update information over the air together with a hash of the new user password (the login credentials for the new account are sent to the user via some other channel).

This operation mode also applies to the case when a user account is deleted or it appartenance to a group changes, with the exception that no further credentials need to be sent.

5.4.4 Access extension and revocation

It is also the CA that initiates changes in the access rights for a given user (add or remove access to some container) - the update message is broadcasted to all devices.

The changes are propagated to the access control structures in a lazy manner: the password for the affected user is needed in order to write the changes to the corresponding UCDB, but that user may never be active on a given device (the users have system-wide validity, but a person holding a device may only use one account and not have the credentials for other accounts). Therefore, the changes to the UCDB are made when the affected user logs in: the monitor will ask for the Administrator password as well and it will use the user password to decrypt the UCDB and copy new keys to this structure or remove the keys for the revoked containers.

5.5 Observations

The system is designed to run on a platform that supports the TrustZone technology; for enterprise usage this is a strict requirement, whereas for domestic usage this restriction can be overseen (with some risks). Regardless of the operation mode, the system has a common weak spot, the user password.

If the platform is endowed with the TrustZone technology and a password is obtained by the attacker, the master passwords for container encryption will still not be revealed. If this technology is not present, a lost password could lead to all container keys for that user to be revealed. In order to secure the password itself, some external hardware token (a smart card) would need to be employed together with the password. Such a *two-factor authentication* mechanism is technically feasible (connect over USB or Bluetooth with the mobile device), the only problem being its usability: the user needs to carry the security token at all times in order to access any of the encrypted containers.

Chapter 6

Implementation details

In this section we will discuss Content Providers in depth (and the most important Android modules that they use) in order to find an appropriate design for an application that would enhance these providers with the possibility to encrypt their data. We will then describe the implementation details of a library that achieves this objective and it does so with minimal changes to the Content Provider software stack.

6.1 Android content management

Before we set off to designing our encryption system, we first need to establish the directions in which the application will extend the functionality of the existing data management system. There are two main tasks that our module needs to perform whenever data from a Content Provider is requested:

1. Enforce extra permissions. Access rights for a regular Android application are set in the manifest file of the application; these privileges are preserved but they are combined with a set of per-user access rights (see Chapter 5). A dedicated software component must handle these dynamic permission verifications - we will further refer to this module as the *monitor service.*

2. Handle encryption/decryption of the requested data. These cryptographic operations must be transparent to the applications that connect to the Content Provider i.e. these applications do not have to be modified in any way in order to accommodate encryption and, ideally, the cryptographic operations do not have a noticeable impact on the overall user experience.

The first task requires implementing *dynamic access control* on Android. Currently, Android only handles this operation in a static manner: access privileges are set in the manifest of an application, if an application has the proper rights when it starts, it will retain these privileges as long as it runs. On the other hand, our security system needs to cope with scenarios where several users (with different access rights) can log in without having to restart the device. In other words it needs to check access rights dynamically, whenever data is retrieved from (or modified in) the containers.

For the second task changes need to be made to the specific mechanisms used by the

Content Providers to manage data. There are no restrictions on how this can be done on user defined providers - a Content Provider can define and use its own data structures. However, the Android SDK offers support for a more convenient solution: using SQLite databases. For this reason we chose to concentrate on adapting the Android SQLite library to fit our purposes.

6.1.1 Binders

The *Binder* inter-process communication (IPC) mechanism constitutes one of the core components of Android and is also a basic ingredient of the Content Provider implementation. It abstracts a message passing communication scheme and it is defined in the Android API by the `IBinder` interface that is in turn implemented by the `Binder` class (the remote object, where the actual operations are performed) and the `BinderProxy` class (the object located on the client side). When a client requires a remote operation, the client that holds a BinderProxy object calls `BinderProxy.transact` which is reflected in the `Binder.onTransact` method on the server side. Transaction requests and their results are marshalled in `Parcel` objects before being passed between the communicating entities.

Android Content Providers are implemented as remotable objects (i.e. reachable from other processes) and this is achieved with support from the Binder IPC: Content Providers in Android extend the `ContentProvider` class which in turn extends the `Binder` class. The observation that the Content Provider itself serves as an IPC mechanism points out the fact that there are several processes involved in operations such as retrieving data from a container. We will be inspecting these processes in the next sections.

6.1.2 Content Providers

Android is delivered with a set of ready-built Content Providers, the so-called system containers - the most frequently used are listed below together with a short description:

- ApplicationProvider: the list of installed applications as used by the application launcher[1].
- SettingsProvider: various system settings.
- CalendarProvider: alarms and the widgets that use them.
- ContactsProvider: the list of contacts along with the call history.
- DownloadProvider: information about downloaded applications.
- MediaProvider: the content referred by the Media Store application.
- TelephonyProvider: information on the current telephony operator and various communication settings.
- UserDictionaryProvider: the dictionary that is used for typing suggestions whenever the keyboard is employed.

[1]The application launcher is a special Android service that initiates the runtime environment for an application that is set to execute.

- BrowserProvider: browser search history.
- EmailProvider: the email container.

Every Content Provider (including the system providers) is associated with at least one identifier, the so-called AUTHORITY. This identifier can be seen as the name of the container and it is used by Android to find the Content Provider (the application) that handles the data stored in the container. A provider is usually instantiated when some process requests data from the respective authorities for the first time, it resides in a designated service (specified in the manifest), it never dies (if a crash occurs, it is restarted by the OS) and it is contacted by remote applications whenever they need to retrieve or update data.

Most of these Content Providers store data in SQLite databases[2] and therefore are backed up by a SQLite driver. This observation supports our decision to design the encryption system around the SQLite driver for Android, as a great number of system providers could easily be adapted to benefit from this service.

6.1.3 Important applications and services

As mentioned earlier, the answer to a data request on a Content Provider is the result of an inter-process communication scheme involving the following entities:

- The *system_server*, a core Android process, runs two important services:
  1. ActivityManagerService: keeps track of all running Activities. It also creates the system Content Providers upon request and it stores `ContentProviderProxy` objects for all running providers. These proxy objects will be passed to applications that require access to the respective containers.
  2. ContentService: dedicated objects called `ContentObservers` register with this service in order to be notified when changes are made to the Content Provider that is being observed.
- The process where the Content Provider resides: this differs with the provider type. For example the ContactsProvider resides in *android.process.acore*(the Android core process), and the SettingsProvider runs in *com.android.settings*.
- The process where the application that runs the data request resides. Each such application has a main thread, an instance of the `ActivityThread` class. This class handles operations such as contacting the `ActivityManagerService` in order to obtain `ContentProviderProxy` objects.

6.1.4 Connecting to a Content Provider

Before any data can be retrieved from a container, a process must connect to the associated Content Provider i.e. obtain a proxy to the process that manages the respective container.

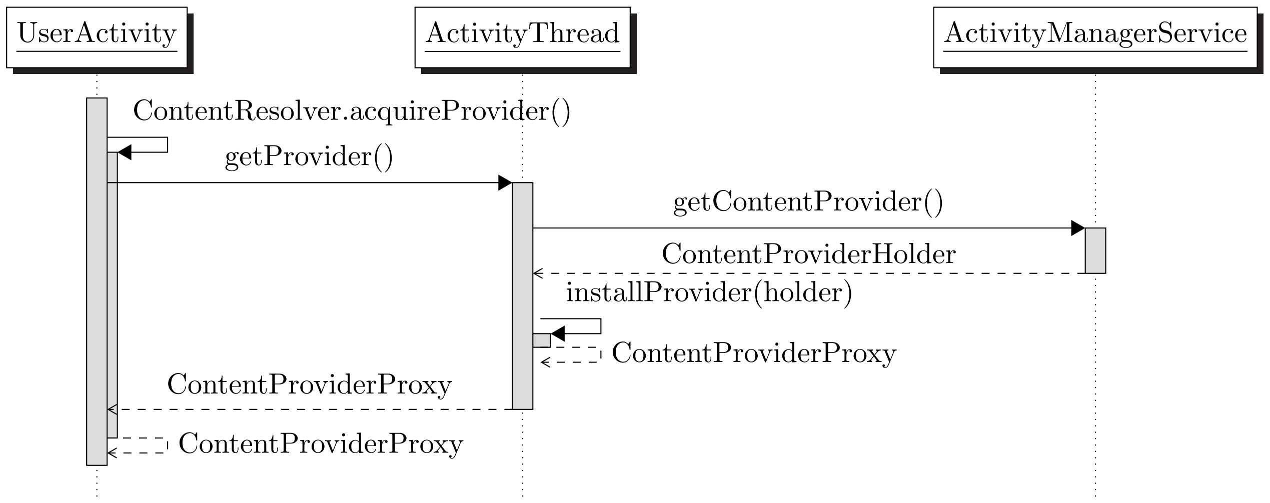

Figure 6.1: Gaining Access to a Content Provider

The user Activity and the ActivityThread run in the same process (every Activity is attached to such an ActivityThread) whereas the ActivityManagerService runs in the system_server process. This Activity is an instance of the `Activity` class and it contains the user defined code - this is the only part that the developer is concerned with, creating the corresponding ActivityThread is the responsibility of the Android platform. If the Activity wishes to retrieve data from a given container, it only needs to provide the proper AUTHORITY name and formulate a request using a method exposed by the ActivityThread. When the user Activity runs such a request for the first time (for a given container) , its ActivityThread contacts the ActivityManagerService and obtains a descriptor for the Content Provider (the `ContentProviderHolder`) that is in charge of the required AUTHORITY. Using this descriptor, the ActivityThread installs the provider i.e. depending on how the Content Provider is set to run, it actually starts a new Content Provider instance in the current process or it just retrieves a proxy to an already running remote provider. This proxy is then passed to the UserActivity that requested the Content Provider.

The call to `ActivityManagerService.getContentProvider` results in a call to `ActivityManagerService.getContentProviderImpl` - in this method an initial *access privileges check*[3] is performed in `checkContentProviderPermissionLocked` before the `ContentProviderHolder` is returned.

6.1.5 Running a query

Once a proxy to the Content Provider is obtained, the user Activity can start retrieving data. In diagram 6.2, ContentProcess depicts the process in which the Content Provider runs. A call to the query function on a `ContentProviderProxy` results in the request

[2]The only exception is the Media Provider where the actual content is stored in media files, but a SQLite database keeps an index of these files.

[3]The privileges set in the manifest file are inspected. The right to access the specific Content Provider needs to be explicitly stated in this file.

parameters being marshalled in Parcels and a remote request being sent over the Binder interface. On the server side of the Content Provider, the request is unmarshalled and the call is being forwarded to the actual Content Provider implementation. The result of the query is a Cursor object that is in turn packed into a Parcel and sent to the party that initiated the query call.

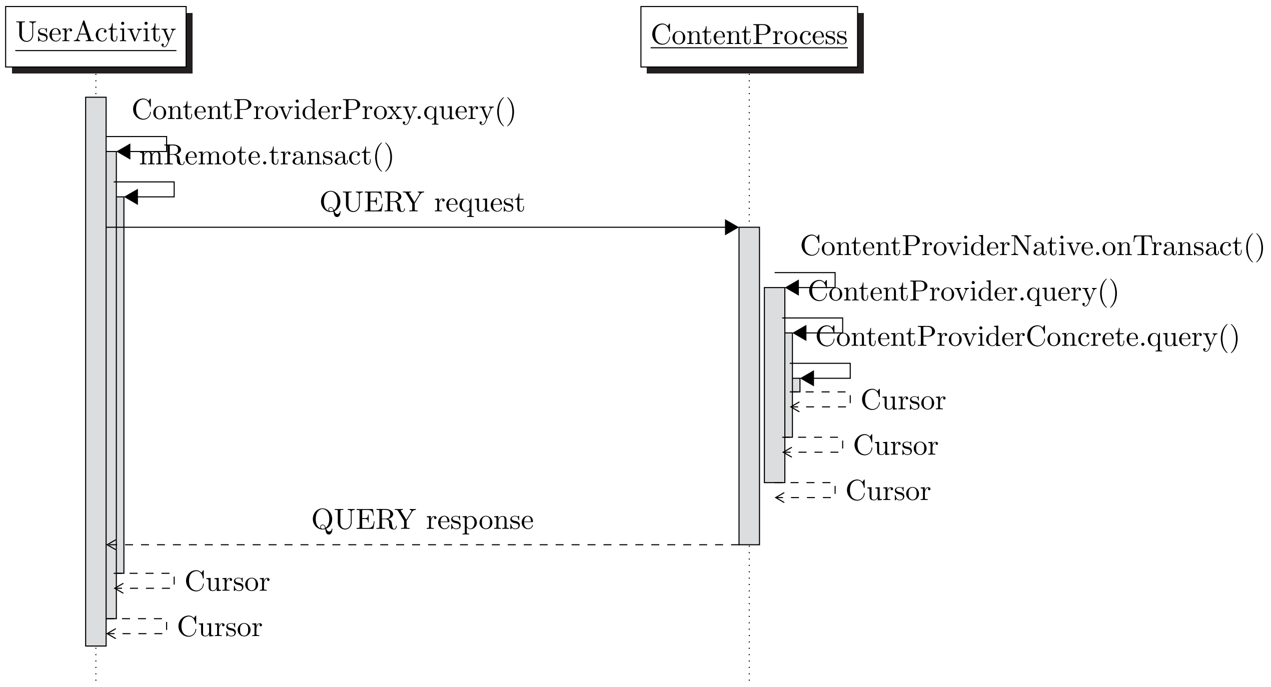

Figure 6.2: Running a query on a Content Provider

All Content Provider implementations stay within the lines of the following class hierarchy: `ContentProvider` extends `ContentProviderNative` and, in turn, the actual Content Provider implementation (in the diagram, `ContactsProviderConcrete`) extends the `ContentProvider` class. On the server side of the Content Provider, all data requests go through the `onTransact` method that, based on the transaction request, dispatches the call to the corresponding function: query(), insert() etc.

This is also the spot where Android makes a second *access check* via calls to `enforceReadPermission(uri)` and `enforceWritePermission(uri)`. The URI that is sent as a parameter to these functions contain the name of the AUTHORITY. These methods contact the ActivityManagerService in order to check the access rights of the calling activity: they use the Binder to retrieve and send the PID and UID of the UserActivity to the ActivityManagerService, that checks if the manifest of the user application does indeed have access to the AUTHORITY listed in the URI. In order to enforce extra permissions these two verification methods can be extended to also connect to a dedicated monitor service.

6.1.6 The problem with cursors

A call to the query method of a Content Provider returns a `Cursor` object. The main purpose of Cursors is to cache query results for future uses. They are in fact more that mere containers since they are implemented as self-refreshable remotable objects. This means that the Cursors that the User Activity receives, holds the requested database entries, but it also serves as a proxy for a `BulkCursor` object that resides in the Content Provider process. Whenever a new Cursor is returned by the Content Provider implementation, this Cursor is registered with the `ContentService` so that it is notified whenever other threads make changes to the database to which it is linked. If such a change is triggered, the Cursor updates itself - it runs a requery operation on the remote Cursor object, without the Content Provider ever being notified.

This entire refresh mechanism stays within the lines of the Android static permission checking mechanism - if an application was able to retrieve the Cursor in the first place, it can never lose access to the Content Provider and it can refresh the retrieved data at will. However, this behavior may pose problems in our dynamic access enforcement mechanism - if the current user changes, the new user must be forbidden from further accessing data in a container before a new privilege check is performed, which means that the ContentService needs to be aware of what Cursors need to be invalidated.

6.2 The crypto application

A quick look at the implementation of Content Providers offers an idea on how to integrate our encryption system with Android: force the ActivityThread to check with the monitor service when first instantiating a provider, force calls to `ContentProvider.onTransact` to connect to this service as well, disconnect all invalid cursors when the user changes, and modify the SQLite driver to support encryption.

We designed an encryption library that can be easily employed by the existing Content Providers. The purpose of this library is to offer a proof-of-concept for the system described in Chapter 5: it is only a demo application and we leave a complete implementation for future work. This reduced encryption system concentrates on the part that runs on the mobile device and shows how Android can be modified to accommodate our security mechanism.

The library is built such that an existing Content Provider can be rewritten to support encryption with minimal changes and the modifications necessary to the Android software stack, are minimal. Also, the design stays within the lines of the TrustZone model - all sensitive operations are isolated in a service and the Content Providers communicate with this service using shared memory.

The two tasks that the system needs to fulfill are implemented as follows:

- The encryption/decryption operations are handled by a modified SQLite driver.
- Access verifications accompany every request. The Content Provider sends packages

to be decrypted by the cryptographic service; if the currently logged in user is not allowed to access the desired container, the service will refuse to decrypt and will signal an error. This error code is set in the first bytes of the response message delivered by the crypto service and it is the duty of the Content Provider to handle this exception. A more elegant (and considerably more complex) solution would also consider disconnecting Cursors that are no longer valid (prevent them from requerying the database). In general, a complete solution would implement a mechanism which notifies all the objects that interact with the database that they may not be valid when the current user changes.

6.3 Architectural overview

The solution that we propose introduces two new Android system services: a *monitor service* and a *cryptographic service*. Both these services are started at boot-time and processes can connect to them using the *ServiceManager*, a dedicated service that runs in the system_server process, handles all Android system services, and offers a simple interface that allows processes to retrieve proxies for the services under its control.

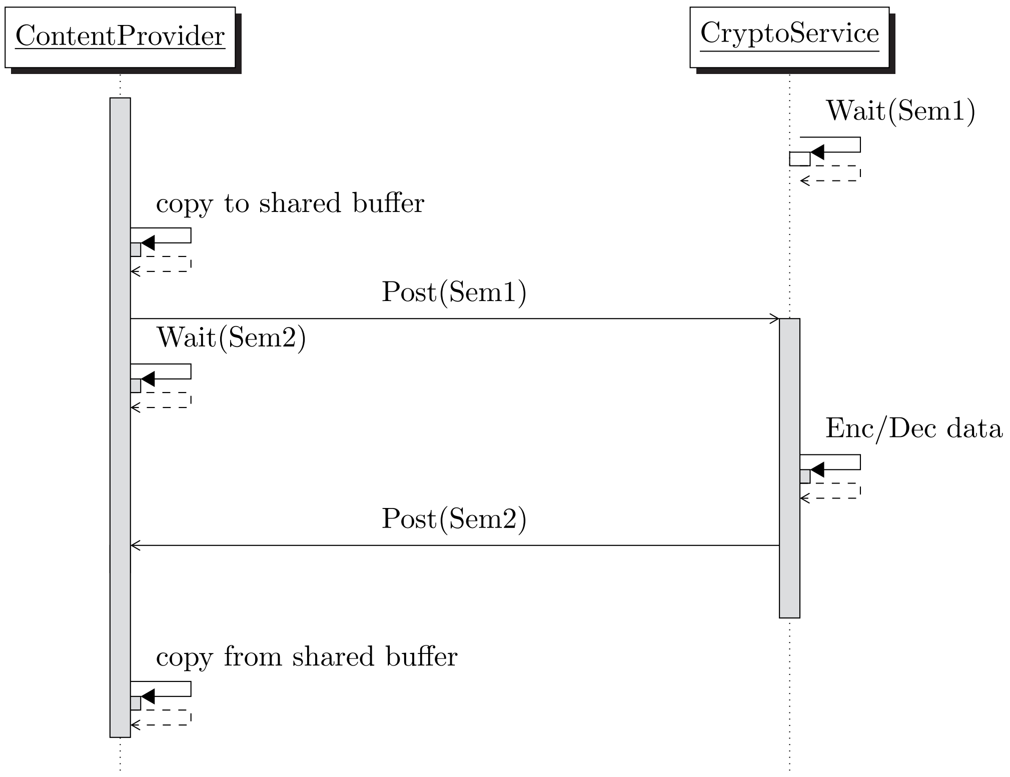

Figure 6.3: Remote data encryption

The *monitor service* manages the access control lists and exposes user management methods to a *login application* and a *monitor application*. The user login application pops-up when the screen is unlocked (it receives the `android.intent.action.USER_PRESENT` intent) and connects to the monitor service in order to check user credentials. If the user changes and some containers are no longer available, the monitor service notifies the cryptographic service that some requests must not be serviced any longer. The monitor application offers a set of user management functions - user addition/deletion, container addition/deletion and adding/removing the access right to a container for a given user.

As stated earlier, most Content Providers manage their data via a SQLite driver. A SQLite solution for Android that supports encryption is offered by the SQLCipher library [8] - this package was modified such that instead of encrypting data in the process where the library is loaded, it connects to a designated service and it uses a shared memory area and two semaphores to place encryption requests (see figure 6.3). In order to keep the synchronization scheme simple, every time a new Content Provider connects to this service, a new encryption thread is spawned. These threads run independently and can only be interrupted by the monitor service: if the user changes, the monitor service notifies the threads that serve containers which are no longer available to the user. These threads will signal an encryption error on all subsequent incoming requests, but are kept alive and will be re-enabled once a user with proper privileges logs in.

6.4 The SQLCipher library

SQLite [7] is an open-source software library that implements a SQL database engine. It is written in ANSI C, which makes it easy to cross-compile, and it offers a self-contained (in that it only uses libc functions), serverless solution: a database management application loads this library and works directly on the database files. The fact that it is also quite small (350 KB) and its simple structure make it a desirable data management solution for mobile platforms.

Moreover, the overall memory consumption of a running SQLite driver is kept low. This is achieved not only by the fact that an entire database is packaged in a single file, but also by having databases partitioned in one KB[4] chunks called pages. The internal mechanisms of this driver organize data in B-trees of pages such that whenever some data is requested, an index is queried and only the required pages are loaded in memory.

The SQLite pager mechanism offers another handy feature: so-called *codecs* can be added to the SQLite library such that custom operations can be performed on a data page right before it is written to the database file or after it has been read from the file. The SQLite library offers a dedicated API for the addition of such codecs and also allows for the addition of custom PRAGMA calls - special instructions that can be employed in order to send configuration parameters to the database driver.

The SQLCipher [6] project makes use of the codec API and adds encryption operations to the SQLite pager - the SQLite code is linked against an OpenSSL library that performs 256-bit AES encryption upon page retrieval and insertion. Encryption keys are derived from user provided passwords that are fed to the driver via a `PRAGMA key` instruction.

[4]This value can actually be configured at compile time, but the default size is 1024 bytes.

The Android API offers its own implementation of the SQLite library - it consists of a Linux SQLite shared library compiled for the underlying platform (for most Android distributions this is an ARM processor), a Java interface, and the Java Native Interface (JNI) adapter layer. It provides a convenient way to manipulate transactions and cursors from Java code. Most applications that use the SQLite module will only interact with this Java interface and rely on the hidden native implementation for low level data management tasks (use Java calls to open a database and manage transactions).

The Guardian Project [8] ported the SQLCipher library to Android. Their solution comes in the form of a native SQLCipher driver enhanced with a Java interface that replaces the default Android SQLite implementation and adds the necessary API calls for cryptographic operations. The changes that this new Java library brings to the Android database API mostly consist of having added a password parameter to the function that opens the database - all other functionality exposed by the API remains unchanged.

6.5 Remote SQLite encryption

The SQLCipher for Android library does not fulfill all the needs of our Content Provider encryption system since all cryptographic operations are performed in the local process (in our case this is the process where the Content Provider resides). However, it does provide a good starting point. This library was modified such that, instead of encrypting data in the local process, it receives as a parameter the name of a service to which it sends pages to be encrypted or decrypted.

6.5.1 The native driver

In the native library, libsqlcipher.so, the codec function is modified as follows: when a page is read from persistent storage, it needs to be decrypted before it is returned. Instead of performing the decryption operation locally, the ciphertext is written to a shared memory area, the post function is called on the first semaphore (Start_semaphore), and then the thread waits on the second semaphore (Done_semaphore). The post operation notifies the remote crypto service that fresh data is available; the crypto service processes the data, writes it back to the same shared memory region and posts the Done_semaphore, signaling to the SQLite driver that the requested operation has completed.

```
ctx->shared_mem_address[0] = (char)mode;
*((unsigned short*)(shared_mem_address + 1))
                                        = (unsigned short)size;
memcpy(shared_mem_address + 2, in, size);
sem_post(start_sem_address);
sem_wait(done_sem_address);
```

Listing 6.1: The codec function - snippet

As shown in the attached code listing, the first three bytes have a special meaning: the first byte depicts the requested operation (encryption or decryption) and the next two bytes store the length of the page in little endian format[5] (the maximum page length supported by SQLite fits in an ANSI C unsigned short). If an error occurs on the server side, this will be signaled in the first byte of the return message and it will be propagated upwards on the call chain.

The shared memory address and the addresses of the two semaphores are created in the Java layer of the SQLite driver (more details in the next section) and are passed to the native code via PRAGMA calls. The SQLite driver offers this dedicated interface such that various setting can be sent to the driver in key-value pairs - for user defined pairs, the appropriate processing code needs to be added to the native library.

6.5.2 The Java interface

The Java interface provided by the SQLite for Android library seeks to replace the original Android SQLite packages entirely. In fact this implementation copies the code in the `android.database` and the `android.database.sqlite` packages and operates the necessary changes. The most notable modification comes in the form of an extra parameter, the password, being sent to the constructor of the SQLiteDatabase class (this constructor sends the password to the native driver via a "PRAGMA key = password" call).

Since our remote encryption driver does not need the password to be set locally, this parameter can be removed. Instead, the SQLiteDatabase constructor needs to receive the address of the shared memory region along with the address of the two semaphores. These three addresses are sent to the underlying driver using three newly added pragma calls, one for each of the required addresses.

It is also this Java layer that connects to the cryptographic service and sets the communication primitives. It does so by using a call to the ServiceManager to retrieve a proxy to the cryptographic service (see details in the next section). As soon as the connection is established, the identifier of the container for which crypto operations are requested is sent to the cryptographic service - this will allow the service to pick the appropriate key in all subsequent operations. It then creates three shared memory areas, obtains the file descriptors for these areas and uses the Binder to marshall these descriptors in ParcelFileDescriptors[6]. These values are sent to the cryptographic service using the Binder remote procedure call mechanism. Once both processes have agreed on these primitives, no further Binder calls are used - communication is only done via shared memory and semaphores.

Android comes with a proprietary implementation of shared memory, the ashmem (anonymous shared memory) Linux driver; the main difference from the POSIX implementation is that it is not associated with an unique identifier that other processes can use in order

[5]If this message is processed in Java code on the receiving side, the big endian JVM needs to rebuild this value correctly.

[6]Linux file descriptors are in fact integer values that are associated to a file structure - this mapping is only valid in the process that opens the file. Fortunately, the Binder offers this mechanism to share file descriptors between processes, such that, when unmarshalled on the receiving side, the file descriptor maps to the same file as in the first process.

to refer to the same memory area[7], two processes need to use the Binder in order to pass the associated file descriptor. The Android API also offers a Java wrapper for shared memory, the `MemoryFile` class that facilitates operations such as data read and write.

In what semaphores are concerned, Android offers POSIX semaphores, with the mention that they cannot be shared amongst processes based on their identifier. Instead a semaphore object (a `sem_t` structure) needs to be placed in a shared memory area such that both processes can call POSIX functions (`sem_open`, `sem_post`, `sem_wait` etc.) from the JNI layer, using the address of this structure.

6.5.3 The cryptographic service

The cryptographic service is the core of the entire security system. On a TrustZone enabled platform, this module along with the monitor service would run in the Secure world as we need these services to be trusted and untouchable during runtime. It works together with the monitor service (in fact they share the same process) that provides the encryption function with the appropriate key.

The mode of operation is kept simple: whenever a new connection is established, the service spawns a new encryption thread that will be associated with the container identifier received from the party that initiated the connection. All these threads run independently, communicate with their peers using the aforementioned semaphores and shared memory, and are monitored by the monitor service - if the user changes, the monitor service can decide to suspend[8] the threads that serve requests for containers that are no longer available.

Given its importance, this service was implemented as a so-called system service. As opposed to regular user-defined services (extensions of the `Service` class), system services are registered with the ServerManager process, they are never killed by Android and they can receive connections in a synchronous manner. The default way to define a service only allows for connections to be established asynchronously: the process that wishes to open a connection sends an intent and registers a callback function that will be notified when the proxy to the service is ready; after this callback function has run, the process can call the methods defined in the service aidl (see Chapter 2 for details). The problem with this mode of operation is that the calling process is blocked until the connection notification arrives in its message queue[9]. Moreover, the process does not busy-wait for the notification, instead, it sleeps and it will only inspect newly arrived messages when it is again brought in a running state by some external action (the service receives another remote call). With a system service this is not the case: a proxy to the service can be obtained with a blocking call to (`ServiceManager.getService(service_name)`).

[7]The POSIX implementation actually creates an entry in the /dev/shm folder, that other processes open as a file before mapping the memory area it corresponds to. Given that in Android processes run with different UIDs, other processes will not be allowed to open this mock-file, therefore some other mechanism needs to be used in order to do the actual resource sharing.

[8]The thread is notified that it should answer all incoming requests with an error message.

[9]Every Activity or Service is associated with a message queue that is managed by a dedicated thread called Looper.

6.6 Android API related issues

We will begin by pointing out that some components of Android, though fully functional, are not exposed in the API; this is the so-called hidden and internal API. Most of this hidden functionality covers parts of the API that are considered unstable or vulnerable at the time of release, parts that may change in the future (thus generating backwards compatibility issues for the applications that use them), or that are considered irrelevant for regular application development. Also, it must be pointed out that we developed our application for Android 2.3.5 and some hidden parts of this API have already been disclosed in Android 4.0. In this section we will list some of the issues that we came across while developing the encryption system and the means by which they can be overcome.

One example of an issue with the hidden API is the fact that the methods in the `ServerManager` class cannot be directly accessed from user-defined code. Another problem arose when using the shared memory mechanism: the Java wrapper, the `MemoryFile` class, opens a new memory area in read-write mode, but it allows a remote process to map the area only in read mode, although the Android implementation of the Linux `mmap` function is perfectly capable of mapping the memory region in write mode. There are several more locations where our code had to cope with such limitations, but all these problems can be overcome by the use of Java reflection (as mentioned earlier, this functionality is there, only that hidden).

A second problem is given by the fact that one of the purposes of our encryption system is to provide extra security to the existing Content Providers. These Content Providers are delivered as apk's located in the `/system/app` folder in the Android image and they are installed when Android boots. In order to replace these providers, root privileges are required and also the new apk needs to be signed with a specific key, the shared key[10]. This key can be found in the Android source tree and it can be replaced with some custom key before Android is built. It is however necessary to replace the Android image on the device with the one containing the new key. Also this key is not available to the end user for the original Android image delivered with the device.

There is a second reason for which the device needs to be reflashed when the encryption system is installed: recall from the previous sections that the cryptographic service is a system service. This means that it must run under the shared UID called "system" so it needs to be signed with a specific key, the platform key. As in the case of the shared key, it is not available to the user for the Android image provided with the device.

A potential problem can come from the fact that the Android API does not provide a way to define modal dialogs. As a result, our login screen can be bypassed, but since the previous user is invalidated on screen lock, this doesn't create a security issue: if the user decides to bypass the login screen, there will be no access to any of the encrypted containers.

[10]This is actually only needed for the providers that run under the special UID called "shared" (e.g. the Contacts Provider)

Finally, we start our cryptographic service at boot time by registering a receiver for the BOOT_COMPLETED intent. However, this approach will not work in Android 4.0 - the service is not put in a running state, it needs to be started explicitly by some user action. This can be overcome by placing the code for the service in a dedicated folder, adding a new entry in init.rc (the file where all system services are listed) and recompiling the Android image. This, of course, requires for the entire OS image to be reflashed.

Chapter 7

Evaluation

In this chapter we provide an evaluation of the encryption system in terms of usability and performance. A first requirement of our system is that it integrates easily with Android and that it does not affect existing applications. Also, one of the concerns with using an encryption system is the impact that is has on the overall performance of the platform. This concern is even more justified on a mobile device, where processor speed, memory limitations and power consumption play an important role in the design of applications.

7.1 Test platform

During the development of the encryption application we used an Android 2.3.5 image compiled from source, and running on an emulator. This tool provided together with the Android SDK is an actual ARM emulator (it emulates the behavior of the ARM CPU and other peripherals and it runs a standard Android image on top of this virtual platform). All our tests run the Android emulator on commodity hardware: a personal computer with an Intel Core i5 CPU and 3 GB or RAM. In terms of user experience, the performance offered by the emulator is comparable to that of using a real device.

7.2 Integration with Android

In order to demonstrate the versatility of our encryption library, we built a modified version of the Contacts Provider that uses encryption and replaces the default provider. The changes consists in linking the system provider with our SQLite encryption library and adding the proper exception handling code that manages disconnections form the encryption service. This package was built in an apk, the manifest of this package set to associate the new provider with the Android AUTHORITY string for the Contacts Provider, signed with the *shared key* for our Android image, and deployed on the emulator. Once these steps are performed, the Contacts and Call Log applications (the applications that run when the call button is pressed) will automatically start using the new provider, with no change to their code. Moreover, the encryption system runs transparently with respect to the user. The encryption system only becomes noticeable when the user changes and the new account does not have the right to access the contact list: no contacts are displayed in the application and no new contacts can be added.

7.3 Performance measurement

We ran performance measurements on a custom Content Provider that manages a set of notes[1]: entries with a title and a body. We created two such Note Providers, one that uses the default SQLite library and one that uses our encryption SQLite library. A test application then connects to both providers and performs queries, inserts, and updates on a similar set of entries (short notes with a title of roughly five characters and a body of ten characters).

Table 7.1: Decryption time for one page - samples [ms]

Server	Client
0.391	2.518
0.339	9.134
0.229	0.794
0.212	2.013
0.211	1.575
0.193	0.900
0.187	0.797
0.182	0.922
0.161	1.989

Table 7.2: Encryption time for one page - samples [ms]

Server	Client
0.907	3.423
0.787	2.646
0.740	2.135
0.661	1.717
0.660	4.589
0.622	4.781
0.582	2.723
0.277	0.917
0.271	1.701

Table 7.3: Average time to encrypt/decrypt a page [ms]

Server decrypt	Client decrypt	Server encrypt	Client encrypt
0.410	2.118	0.308	1.902

The encryption service operates with units called *pages* of 1024 bytes each. We measured the time in milliseconds necessary to decrypt a page and to encrypt a page; a few samples are listed in table 7.1 and 7.2 respectively. Also, table 7.3 lists the average time for cryptographic operations calculated from a set of 100 measurements for each of the columns. In these tables, the *server time* measures only the interval for the cryptographic operation performed in the encryption service, whereas the *client time* is the entire interval necessary to obtain a page, as seen from the perspective of the Content Provider. This interval includes the time spent in order to copy a page to shared memory, notify the encryption service that new data is present, and wait for the result to show up in shared memory. For a fair result, the operation that copies the data from shared memory back to the client buffer is not included, as a similar operation would also occur if the default

[1]In fact this provider was used throughout the entire development phase in order to keep the structure of the database simple.

SQLite library was used (copying data from the database file to the client buffer).

There are noticeable inconsistencies in the client time for the samples in 7.1 and 7.2: the difference between the client time and the server time should be constant as this difference is caused by the same sequence of data sharing and synchronization operations that accompany every request. These inconsistencies can be explained by the fact that the operating system may choose to schedule other tasks in this interval: the client process posts a semaphore and then sleeps waiting for the other semaphore. This sleep operation triggers the Linux scheduler to run some random task that was waiting, not necessarily the encryption service. Moreover, the underlying Linux kernel in Android is preemptive so both the client application and the encryption service can be interrupted at any time (holding the semaphore does not disable preemption).

Table 7.4: No encryption [ms]

Row #	Insert	Update	Query all
10	116.758	87.344	17.453
20	224.122	173.469	14.424
30	379.712	322.023	15.183
40	453.505	573.115	16.805
50	610.037	662.340	16.445
60	834.528	927.414	20.935
70	934.562	1071.927	17.883
80	1087.395	1358.585	18.395
90	1230.477	1528.834	19.082
100	1202.021	1742.441	20.351

Table 7.5: With encryption [ms]

Row #	Insert	Update	Query all
10	186.774	95.586	30.519
20	356.225	211.782	22.957
30	489.417	328.163	32.480
40	676.117	671.346	39.912
50	826.060	714.512	49.439
60	1075.279	1040.313	59.189
70	1188.590	1081.876	56.164
80	1366.597	1462.620	62.267
90	1635.070	1628.988	62.449
100	1719.279	2170.913	66.288

Given that our tests run on an emulator, the encryption/decryption time intervals may not provide an exact measure of performance by themselves. Therefore, for a second test, we measured the time in milliseconds for common database operations in a Content Provider that uses encryption (table 7.5) and one that uses the default Android API (table 7.4) for a variable number of database entries. In this case, the ratio of the time for each operation when using the encryption system to that obtained with no encryption, is expected to be preserved when the system runs on a real device: the emulator runs ARM instructions, thus, on a real device, the exact same code would run, only that at a different frequency. One important observation that can be made on these values is that they stay within the same order of magnitude for a given number of entries.

Also, in these tables, the number of page encryption/decryption requests necessary for each database operation cannot be established with precision. This number is highly dependent on the internal mechanisms of SQLite. For example, to serve a database query requesting a table row, the SQLite driver will first decrypt an internal index table and locate the necessary pages. If the requested data happens to reside in one single page, there will be only one decryption request, otherwise all the necessary pages need to be decrypted.

The encryption system induces an overhead of roughly 40% for insert operations, 11% for updates, and 150% when all database entries are retrieved. However, user experience does no suffer a noticeable deterioration. For example, there are no lags when using the Contacts application with our encryption system instead of the default application.

7.4 Observations

We have not included in our tests the Binder calls that are performed when the connection between the client application and the encryption service is established; the overhead introduced by these remote procedure calls is negligible since they only occur when the connection is first established or when the user changes.

Synchronization operations that use semaphores, on the other hand, occur for every page access. Android user-space semaphores copy the standard Linux implementation, with one minor detail that actually has an impact on performance, namely that they do not use Futex[2] objects. However, this is not a problem with our synchronization scheme, since lock contention is kept at a minimum: every client process communicates with its own encryption thread. Actually, these semaphores are used as a notification mechanism rather than a mutual exclusion scheme. In other words, every time *wait()* is called on a semaphore we want the calling process to sleep and be woken up when the requested operation has finished.

This last observation takes us to a second point: even though there is a thread created for every connection, these threads sleep most of the time. They are only active when some

[2] *Fast user-space mutexes* are optimized user-space mutexes that, unlike normal mutexes, only enter kernel-mode when they need to sleep: a counter value is tested in an atomic manner form user-space and, if the mutex can be acquired, the process enters the critical zone immediately. Otherwise they trigger the necessary system call so that the process can sleep and be rescheduled at a later point.

Content Provider requests data, for instance when the contacts application is used. There is also an overhead associated to the creation of each thread, but this only occurs when the encryption connection is first established.

Finally, we need to take into consideration how running on a processor with TrustZone would impact performance: when the connection to the encryption service is first established, a specific CPU instruction needs to be triggered in order to call functions that run in the Secure world. However, once the shared memory area is set, there should not be any communication overhead: the MMU makes the pages available to the Normal world application, these pages are mapped in the virtual address space and used as if they were part of the Normal world memory.

Chapter 8

Summary

8.1 Conclusion

In this thesis we discussed the design and implementation details of a novel encryption system for Android Content Providers.

While looking for a proper design for the encryption system, we gave an overview of the type of support that dedicated hardware platforms have to offer for security-related applications. We discussed the advantages and challenges of using the ARM TrustZone technology and decided to base our design on this special CPU feature. The next two chapters detailed the design of the enterprise encryption system and then we moved onwards to the implementation details of an application that provides parts of the functionality in this design.

Our defense system is designed such that it can be used in an enterprise environment and offers security guarantees even when running on a corrupted OS. Moreover, our implementation of the SQLite encryption library runs transparently with respect to the user of the encrypted Content Provider, thus achieving one of the main goals set in the requirements section: the developer of third-party applications needs not be aware that the containers are encrypted. Also, the encryption library is implemented such that it can be ported to a TrustZone-enabled platform with little effort: the parts that need to be moved to the Secure world are well isolated. Finally, we give a set of performance measurements for our implementation and offer an analysis of its runtime behavior, showing that user experience will not deteriorate when our system is in use.

8.2 Limitations

There are two main points to be discussed in terms of limitations: first of all, if the encryption system runs on a TrustZone enabled processor, conceptually, it is not possible to retrieve the master encryption keys for the container. This is true only if the code that runs in the Secure world is properly implemented. In other words, the TrustZone technology will guarantee that it is not possible for an unauthorized process running in the Normal world to randomly access system resources of tasks that run in the Secure world. However, if a Secure world process decides to publish some sensitive information (or rather, it is tricked by a Normal world application into doing so) there is nothing that would prevent this action. This is not a limitation of the TrustZone itself, but rather a general coding guideline: the Secure world code is isolated and cannot be changed without proper authorization, but it is the duty of the developer to check for confidentiality and integrity (preferably by running some code analysis that checks for improper flow of data).

There is a second limitation to our encryption system: the end-user is the weakest point in the system. If the user decides to make the password public, there is nothing standing between the attacker and the confidential information stored on the device. This, however, is a problem with every encryption system and it can be mitigated by the use of an external authentication token.

8.3 Future work

As discussed in Chapter 6, our demo implementation only covers part of the functionality and it needs to be extended in two directions: in a first step, the application needs to be ported on an Android version that has support for the TrustZone[1] and the encryption service needs to be implemented as a process in the Secure world. Also, in order to make the system suitable for enterprise usage, the Central Authority together with the communication module on the device need to be implemented.

[1]At the moment there is ongoing work from Linaro (www.linaro.org) in this direction. Their latest Android distribution released in April 2012 is compiled for the Freescale iMX53 board and the kernel build configuration has the TrustZone interrupt controller enabled.

References

[1] BitLocker Drive Encryption Technical Overview. http://technet.microsoft.com/en-us/library/cc732774(v=ws.10).aspx#BKMK_SystemDesign, 2009. [Online; accessed 14-April-2012]. 4.1

[2] Android developer web page. http://developer.android.com, 2011. [Online; accessed 14-April-2012]. 2.1, 2.1, 8.3

[3] Android Security web page. http://source.android.com/tech/security/index.html, 2011. [Online; accessed 14-April-2012]. 2.5

[4] ARM Development Studio. http://www.arm.com/products/tools/software-tools/ds-5/index.php, 2011. [Online; accessed 14-April-2012]. 4.3.4

[5] Secrets for Android. http://code.google.com/p/secrets-for-android/, 2011. [Online; accessed 14-April-2012]. 2.6

[6] SQLCipher web page. http://sqlcipher.net/, 2012. [Online; accessed 14-April-2012]. 6.4

[7] SQLite web page. http://www.sqlite.org/about.html, 2012. [Online; accessed 14-April-2012]. 6.4

[8] The Guardian Project - SQLCipher for Android. https://guardianproject.info/code/sqlcipher/, March 2012. [Online; accessed 14-April-2012]. 6.3, 6.4

[9] ARM. *ARM Security Technology - Building a Secure System using TrustZone Technology*, April 2009. 3.1, 4.3, 4.2, 4.3, 8.3

[10] TGC Best Practices Committee. Design, implementation and usage principles, version 3.0. Technical report, TGC, February 2011. 4.1

[11] David Kleidermacher. Green Hills Software Integrity Multivisor. http://www.iqmagazineonline.com/current/pdf/Pg26-29-IQ30.pdf, 2011. [Online; accessed 14-April-2012]. 4.3.3

[12] Sarah Dean. Freeotfe v5.21 - a free on-the-fly transparent disk encryption program. Technical report, FreeOTFE, February 2010. 2

[13] Freescale. *Hardware Reference Manual for i.MX53 Quick Start-R*, 2011. 4.3.4

[14] Giesecke & Devrient GmbH. Mobicore Secure OS. http://www.gi-de.com/gd_media/media/documents/brochures/mobile_security_2/MobiCore.pdf, February 2012. [Online; accessed 14-April-2012]. 4.3.3

[15] David Grawrock. *The Intel Safer Computing Initiative: Building Blocks for Trusted Computing*. Intel Press, 2006. 4.2

[16] IETF. The transport layer security (tls) protocol version 1.2 (rfc 5246). Technical report, IETF, April 2008. 5.3.1

[17] Intel. *Intel Trusted Execution Technology (Intel TXT) Software Development Guide*, March 2011. 4.2

[18] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Trusted language runtime (tlr): Enabling trusted applications on smartphones. *Proceedings of HotMobile, Phoenix, AZ*, March 2011. 4.3.3

[19] TGC. Tpm main part 1 design principles, version 1.2. Technical report, TGC, March 2011. 4.1

[20] TGC. Tpm main part 2 tpm structures, version 1.2. Technical report, TGC, March 2011. 4.1

[21] TGC. Tpm main part 3 tpm commands, version 1.2. Technical report, TGC, March 2011. 4.1

[22] TGC. Trusted platform module (tpm) summary. Technical report, TGC, March 2011. 4.1

List of Figures

2.1 The architecture of Android [2] 10
2.2 Android applications 12
2.3 Unlock pattern 15

4.1 TPM Architecture 24
4.2 The TrustZone virtual cores [9] 26
4.3 The architecture of an ARM SoC [9] 27

5.1 Control structures 32
5.2 Container critical data block 34
5.3 User critical data block 34
5.4 User access block 34
5.5 Key Wallet 36
5.6 Access Control List 36
5.7 Communication with the TrustZone 37

6.1 Gaining Access to a Content Provider 43
6.2 Running a query on a Content Provider 44
6.3 Remote data encryption 46

Listings

6.1 The codec function - snippet 48