Secure Key Storage and Secure Computation in Android

Master Thesis

June 30, 2014

Author:
T.J.P.M. (Tim) Cooijmans

Supervisors:
dr. ir. E. (Erik) Poll
prof. dr. E.R. (Eric) Verheul
drs. T. (Ties) Pull ter Gunne (SNS Bank)

Radboud University Nijmegen

Executive Summary

The increasing usage of smartphones also gains the interest of criminals who shift their focus from attacking, for example, internet banking in a browser to attacking mobile banking using an application on a smartphone or a tablet. Of course this is not limited to banking applications, also other mobile services are expected to see increased fraudulent activities in the coming years. Two important solutions to protect against attacks on mobile devices are secure key storage and secure computation. Secure key storage provides an environment where secret values needed for securing communication between parties are stored. Secure computation offers a secure environment within a device where trusted applications can run and handle sensitive operations such as asking for a PIN-code.

Recent Android phones have hardware support, called ARM TrustZone® technology, to create a secure environment, isolated from the Android OS, that attackers can not access. The secure environment is called the Trusted Execution Environment or TEE. This technology is used on recent Android devices to provide secure key storage and secure computation. The Android documentation claims that this ensures that the secret data can not leave the phone. However, in this thesis we describe an issue that allows an attacker to, under certain conditions, assign the secret data to another application. The effectiveness of this solution is almost nullified by this issue. In this thesis we provide recommendations to the developers, who use or intent to use secure key storage in their application, how to mitigate the risks of this issue and how to use the key storage features.

Next we analyze both currently used and possible future use cases of the secure computation. Secure computation is currently used to secure digital media delivery (movies, music) to devices such that the media can not be played on different devices. The secure computation possibilities currently available on Android OS can be used to improve the security of use case, such as mobile banking, mobile payments and secure communication. The TEE can, for example, be used to securely ask the user to confirm a mobile banking transaction in such way that it is guaranteed that the confirmation originated from a specific device.

The conclusion is that while there are some problems, either security or cost related, with both secure key storage and secure computation on Android, both solutions can provide additional security properties that may be very welcome to secure mobile applications. Secure computation can be used in more complex use cases that can not be solved by only secure key storage.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

1.1 Problem statement

The use of mobile platforms such as smartphones has grown enormously in the last five years and with it also the number of mobile applications (also called *apps*) on these platforms. Nowadays most corporations have created or use mobile applications to either support their personnel or their customers.

A result of this is that sensitive data is processed by or stored on mobile platforms, not only on phones issued by companies to employees that the companies can control but also on their customers phones over which they have virtually no control.

Banks are example of this. Already in 1986 the Dutch bank Postbank introduced a banking application that could be installed on a computer [27]. Almost 30 years later nearly every bank has a number of applications on all commonly used mobile platforms for customers to manage their funds. According to recent[1] figures by ING 51% of their customers uses a mobile banking application on their smartphone. More interestingly 29% of ING's customers exclusively uses the mobile banking application.

Naturally this also gained the awareness of criminals who started creating malware for the mobile platforms [24]. In December 2012 the Eurograbber SMS trojan intercepted SMS messages on Android phones containing *Transaction Authentication Numbers* (TAN) to confirm banking transactions. The attack was employed on both the computer and the Android phone of a banking customer [17]. According to the Cisco 2014 Annual Security Report 70% of all Android users are affected by malware [10]. Cisco TRAC/SIO researchers observed that 99 percent of all mobile malware target Android devices [10].

This in turn forced the manufacturers of the mobile platforms and the application developers to invest in securing their mobile platform and the apps on it. Already around 2008 both Android [22] and Apple iOS [39] incorporated additional security measures.

Two security concepts that we will research in this thesis are *secure computation* and *secure key storage*. Secure key storage provides an environment where secret data needed for securing communication between parties is stored and that can be used for standard cryptographic operations. Secure computation offers a secure environment within a device

[1] `http://www.adformatie.nl/blog/mm/2013/11/mobiele-app-verandert-bankiergedrag.html`

where trusted applications can run and handle sensitive operations such as asking for a PIN-code or running a specific cryptographic algorithm. Note that secure key storage is a specific, commonly used, case of secure computation.

To implement secure computation and secure key storage on mobile platforms hardware solutions were invented. One commonly used solution for secure computation and secure key storage is the Secure Element [28]. This is a smart card like tamper resistant platform that can be embedded in systems as a chip or integrated in UICC cards (SIM cards).

Another solution is ARM TrustZone Technology in ARM processors. ARM processors are used in almost every smartphone regardless of the operating system. In 2012 ARM announced that they would include ARM TrustZone Technology in every processor design that they license to manufacturers [38]. As a result many smartphones today are equipped with a TrustZone compatible processor. ARM TrustZone Technology is a hardware-based solution embedded in the ARM cores that allows the processor cores to run two execution environments, also called worlds: The *normal world* where for example Android OS or any other operating system runs and a special *secure world* where sensitive processes can be run. Both worlds can run interleaved.

ARM TrustZone Technology provides the basis for a Trusted Execution Environment (TEE). The TrustZone hardware features, together with some software, ensure that resources from the secure world and some specific devices can not be accessed from the normal world. The TEE offers secure computation (and as a consequence also secure key storage). However, a TEE also provides a way to securely communicate with a user as we will see in chapter 2. This is not possible for a secure element.

This thesis focuses on the security features provided by the Android mobile operating system and the security features in the hardware that is commonly used in Android smartphones. We will focus on one very specific use case, cryptographic key storage (and operations using these keys), that is very applicable today and one more elaborate use case, secure computation.

1.2 Research questions

The first question is: *What methods do Android and the underling hardware use to provide secure key storage and what security properties does it provide and how should it be used?* This question will be answered in chapter 3 by the following questions:

1. What methods does Android use to provide secure key storage?
2. What security properties does the secure key storage provide?
3. What best practices should be applied when applying secure key storage in an application?

The second question is; *What methods are available to implement secure computation on Android?* This question will be answered in chapter 4 by the following questions:

1. How does the TEE work and how can it be used?
2. What security properties does the TEE provide?
3. How does it compare to other technologies?

By answering these two main questions we provide an overview of the current possibilities and vulnerabilities of secure key storage and secure computation on Android and mobile platforms in general. we also provide some practical best practices to aid developers with securing their applications based on the possibilities and vulnerabilities. The use cases in chapter 4 provide a look into the future by providing a number of use cases on how secure computation can be used using currently available technologies and software.

2 Background

This chapter is composed of several sections. Section 2.1 describes shortly what cryptographic keys are and what they do. This is needed to understand the concepts discussed in the next sections and chapters. In section 2.2 a description of a Trusted Execution Environment and the requirements associated with it are given. To provide an insight on how a TEE is composed, the ARM TrustZone technology (in section 2.2.1) and the role of the operating system (in section 2.2.2) are discussed. TrustZone technology is not the only method to provide a TEE on a mobile platform, in section 2.2.3 the alternatives are discussed. Because the research will focus on mobile devices running Android OS a technical description of Android OS and its security features is given in section 2.3.

2.1 Cryptography

This section gives a short introduction into cryptography. Since I do not intent to make this an introduction into cryptography course, We will only focus on whát cryptography does and what is needed to accomplish it, not on hów it works. Would you like to know more on how the cryptographic algorithms work, I can recommend the book Fundamentals of Cryptology by Henk C.A. van Tilborg [56]. In this section we will describe two different kinds of algorithms: asymmetric and symmetric cryptographic algorithms. Furthermore we will describe two use-cases of cryptographic keys: encryption and signature generation.

2.1.1 Asymmetric cryptography

Asymmetric cryptography builds on the notion of a *key pair*. This key pair consists of a *private key* and a *public key*. It is created by *generating* it. The private key of the key pair should, as you would expect by looking at the name, be kept private. It should be impossible to derive the private key from the public key, however this depends on the size of the key pair. We call the person who has control over the private-key the "owner" of the private key. The key pair can be used for encryption and signature generation. For this porpuse the public key, binded together with the identity of the key owner, is distributed publicly to everybody. A common way to do is, is by using trusted third party, a *Certificate Authority* (CA). This CA checks the identity of the owner of a key pair, by for example meeting face-to-face. The CA then signs (how we will see later) a certificate that confirms that a certain person or other entity has a certain private key. An example of a structure that uses a CA is the SSL connections and certificates we use for securing internet communication. The use of SSL to secure HTTP traffic is well known as HTTPS and is commonly used for internet banking. Whether the identity of the public key is checked and validated can be seen by the green padlock in the browser for HTTPS connections. However, there are more uses of SSL connections, such as for sending and receiving e-mail and for more application specific connections.

When a third party (Bob) needs to send an encrypted message to the key owner (Alice) he encrypts the message by using the public key as input to the encryption operation of the asymmetric cryptographic algorithm. The only way to decrypt this message is by using the private key in the cryptographic algorithm, that was generated together with the public key as a key pair. As a result the sender, Bob, is sure that only the owner of the private key, Alice, can decrypt the message. This provides secrecy of the message. Here we can already see the importance of the CA. Assume that Bob did not meet with the key owner Alice before but he wants to send her a message. To do this he needs to have the public key of the person he wants to send a message to. However, if he has to rely on a third party, Eve, to provide him this public key, he needs to be sure that the public key that is provided is actually the public key of Alice and not from Eve or another malicious user who has interest in reading the message. Also note that if the private key is not kept private, everybody can read messages that are sent or have been sent.

As analogy assume that a person has a unlimited supply of padlocks with his name on it to which only he has the key. He distributes the padlocks unlocked to everybody who wants to communicate securely with him. When a third party wants to send the owner of the key a message, he first takes a box and puts the message in. Then he takes an unlocked padlock that has the name of the addressee on it. He locks the box by closing the padlock. Note that since he does not have to key he can only close others padlocks not open them. He then sends the box with the locked padlock to the owner. He opens the padlock and the box using his private key and reads the message.

Signature generation works to other way around. Here the key owner wants to prove that a certain message (with a certain content) was written by him. To do this he first creates a cryptographic "summary" of the contents of the message. This is called a *cryptographic hash* and is created by a *hash function*. This hash function is publicly known and can be executed by anybody. If even the smallest bit of the content changes the output of the hash function changes completely. The size of the output of the hash function (the hash) is fixed. As a result there a multiple messages that have the same hash. Still there are countless possible outputs (hashes) for the hash function (regularly 2^{128}-2^{512}). It is easy to compute the hash of a certain message using the hash function. However, it is infeasible to find a message that has a certain hash or find two messages that have the same hash.

The hash of the message is signed by the key owner, for example Alice, using her private-key, this creates the signature by using the cryptographic algorithm. She then sends the message together with the signature to a certain party. Assume that this party, Bob, has the public key of Alice and knowns that it is Alice's public key. Bob then first computes the hash of the message himself. He then applies the asymmetric cryptographic algorithm on the signature from Alice to verify the signature. If the signature is valid he knows that the message is not changed and that it was signed by Alice. This provides integrity and authenticity of the message. If we make sure that Alice is the only one who has the private key, she cannot deny that she has signed the message. This is called *non-repudiation.*

Note that normally it is best practice to use three key pairs, one for encryption, one for authentication and one for signature generation. As we've seen, the encryption and decryption operations are used for both encryption and signature generation. In the case of encryption, the public key is used to encrypt. When decrypting the private key is used. In the case of signature generation the message is signed using the private key. For verification the public key is used. If you can convince the key owner to provide you with the decryption of a certain crafted message (the hash of a certain message) you can use this as a signature for the message. Now the key owner has unintentionally signed a message of which he does not know the contents. If a separate encryption and signature generation key pairs are used you know as a verifying party that something is wrong when you received a message that is signed using an encryption key.

There are multiple cryptographic primitives that are used to create asymmetric cryptographic algorithms for signature generation and encryption. The first well known example is *RSA* [46] which was patented in 1977 [2]. At the moment, however, it freely available in for example the OpenSSL library[1] that is commonly used for cryptography. It is secure under the assumption that factoring a multiplication of large two prime numbers is hard. To be secure RSA key pairs with a length of 2048-bits are recommended for signature generation by the BSI (Bundesamt für Sicherheit in der Informationstechnik, the German Federal Office for Information Security) [41].

Another, newer, technology is *Elliptic Curve Cryptography* (ECC). The signature algorithm based on ECC, called *Elliptic Curve Digital Signature Algorithm* (ECDSA), was first published in 1987 [34]. It is gaining attention because the algorithm is more efficient, both in key-size as in required computational effort, than RSA. For example the recommended ECDSA key-size by BSI is 224-bits or 250-bits depending on some cryptographic details we will not explain here [41]. This is a factor 8 smaller then the recommended RSA key-size which is 2048-bits [41]. However, this is only an estimation, others provide other estimations (see the keylength.org website[2] for an overview of estimations). The OpenSSL library also has support for ECDSA. There are also algorithms to do encryption using ECC, such as the *Elliptic Curve Integrated Encryption Scheme* proposed in [52]. However, it does not appear on the OpenSSL cipher list[3].

[1]https://www.openssl.org/
[2]http://www.keylength.com/
[3]https://www.openssl.org/docs/apps/ciphers.html

2.1.2 Symmetric cryptography

Symmetric encryption uses the same key for encryption and decryption. To enable two parties to send symmetrically encrypted messages they have to establish a key that they both use. When Alice wants to send a message to Bob she encrypts the message using the established key. Bob then uses the same key to decrypt the message. Since Alice and Bob both have the same key there are two parties that can leak the key, and it is hard to establish who did it. On the other hand, if Alice and Bob do not leak the key, there is nobody else who can send encrypted messages using the key. This is in contrast with asymmetric cryptography where anybody who knows the public key of a user can send an encrypted message. This is can be an advantage since you establish a connection between two parties. On the other hand it means that in world where many people communicate with each other you have to establish a key for every pair of users that want to communicate. Symmetric encryption is also magnitudes faster then asymmetric encryption. This is one reason why protocols are developed that establish a symmetric key for exchange of data using asymmetric cryptography. These protocols have the advantages of both asymmetric and symmetric cryptography.

It is also possible to do (a kind of) signature generation using symmetric cryptography. Since two parties have the same key material it does not have the same properties as a asymmetric signature generation. It is therefore called an *Message Authentication Code* (MAC). It also works by creating a cryptographic hash of the contents of the message. However, the key is also included in the contents that are hashed. This means that only the party who has the key can create or verify the hash. To verify the MAC, the other party also computes the hash using the contents of the message and the key. This algorithm provides integrity and authenticity of the message. Note that here authenticity is only provided between the two parties who posses the key. So non-repudiation can not be provided in both users have full control over the key. By using some kind of hardware that makes the key non-exportable and only available for certain specific tasks, however, non-repudiation can be provided. A regularly used MAC function is HMAC [7].

Two regularly used symmetric encryption algorithms are the *Data Encryption Standard* (DES) and the *Advanced Encryption Standard* (AES). DES was published as a standard in 1977 [54]. It originally used a 56-bit key. In 2006 a DES key could be broken in 9 days for under 9000 euro [35]. A variant that uses a chained encryption, decryption, encryption sequence was created, called 3DES [26] to increase the security while using the same hardware. The Rijndael [13] cipher was chosen as AES in 2001 [44]. It uses 128, 192 or 256-bit key-sizes.

2.2 Trusted Execution Environment

An *Trusted Execution Environment* (TEE) is an environment that allows for secure execution of applications. According to ARM documentation applications running in the TEE are called *trustlets*. Vasudevan et al. provide a number of requirements that are needed to ensure a Trusted Execution Environment [57]:

- **Isolated execution** TEE should allow applications to be run in isolation from other applications. This ensures that malicious applications can not access or modify the code or data of an application while it is running.
- **Secure storage** The secure storage of data should be provided to protect the secrecy and integrity of the binaries that represent the applications while they are not running. The same security properties should also be guaranteed for the application data. Note that application data can be even more sensitive than the binaries as passwords and secret keys may be stored in the application data.
- **Remote attestation** For a service to verify that it is actually talking to the software on the device it intends to talk to the principle of remote attestation is invented. It allows parties communicating with the secure execution environment to check the authenticity of the software and/or hardware that implements the TEE.
- **Secure provisioning** It should be possible to send data to a specific software module operating in the execution environment of a specific device while guaranteeing the integrity and secrecy of the data being communicated.
- **Trusted path** It should be possible for the execution environment to communicate with the outside world and to receive communication from the outside world while guaranteeing the authenticity of the communicated data and optionally also the secrecy and availability. This should allow on one hand a party, either human or non-human, to verify that the communicated data originates from the execution environment. On the other hand the technology should ensure that data from peripherals received by the environment is authentic.

Another requirement that is not discussed by [57] is *local attestation*. This is discussed by [45]: Some of the interfaces to communicate to the user can be used by both the secure and non-secure world. For example the secure world and the non-secure world can both control the display in TrustZone-enabled processors. This makes it hard for the user to verify that he is interacting with an application in the TEE. To solve this the TEE has to provide local attestation. Local attestation should enable an user to check that they are in fact interacting with the TEE. An example solution is the Secure Mode Indicator [14] patented by Texas Instruments. This solution provides the user a LED that is hardware controlled and that is only lighted when the secure world is running.

To illustrate the requirements we look at a (fictional) mobile banking application. This application communicates with a back end of a (fictional) bank. Such application may have the following security requirements:

- Nobody should be able to inspect or interfere with the execution of the banking application (or the sensitive parts of the execution) and the data stored by it. To accomplish this isolated execution and secure storage are needed.
- The backend should be able to securely communicate with an instance of the application. Remote attestation provides the ability to check the authenticity of the application. Secure provisioning provides the possibility to securely send data to a specific instance of the application. Secure storage can be used to store credentials needed to setup a secure and authenticated channel with the backend.
- The user of the application should be able to confirm transaction details securely without interference. Local attestation allows the user to verify that he is actually talking to a genuine banking application. It prevents phishing attacks. The trusted path allows the application to securely show the transaction details to the user and it allows the user to securely confirm a transaction (for example by entering a PIN).

2.2.1 ARM TrustZone

Providing a Trusted Execution Environment

There are several methods to implement a TEE as described in section 2.2 on page 8. Recent ARM processors have a feature called ARM TrustZone® Technology that adds security features to the ARM cores. An overview of ARM TrustZone is given in a white paper [9] by ARM. In this section an overview of ARM TrustZone based on this white paper will be given.

The basis concept of TrustZone is based on two environments within one system called *worlds*. The first world is the *normal world* this is the normal environment in which the operating system and all applications running on it operate. The second world is new and is called the *secure world*. In this world applications are using the requirements discussed in Section 2.2 to provide specific security services such as secure key storage. These worlds are achieved by separating both software and hardware resources [9]:

- All memory in the system is separated. This includes the system's RAM but also the registers of the CPUs. The RAM is split into two separate virtual memory spaces. One for the secure world and one for the normal world. This means that the normal world cannot access the memory used by the secure world. The persistent memory (such as flash memory) can be separated by using encryption.
- A dedicated cryptoprocessor and memory for storing keys can only be made accessible by the secure world. This prevents the normal world from accessing sensitive key material.

- The display controller can use both a section of the memory of the normal world and a section of the secure world as display buffer. This dual buffer setup allows the secure world to communicate information to the user without interference from the normal world. As the display buffer for the secure world is located in the secure world memory, the normal world applications cannot access it.

To ensure the requirements discussed to build a Trusted Execution Environment a concept called *secure boot* is normally used. This process verifies the integrity of the contents of the storage that contains the operating system and checks that the operating system is issued by the device manufacturer by checking its signature. This prevents attackers from modifying or changing the operating system.

How TrustZone works

One of the main hardware features of TrustZone technology is the Security bit on the communication bus. The ARM processor has a communication bus called the *AXI-bus* that is used by the main processor to communicate with peripherals (see figure 2.1 on page 11). These peripherals can be located in the same package or chip or outside the package. When multiple systems are located on one chip or in one package this is called a *System on Chip* (SoC). The security bit is added to this bus to communicate to the peripherals whether the transaction they are receiving is either from the secure world or the normal world. All peripherals should check the security status of the transaction and ensure that they do not leak any sensitive information.

Another aspect of the TrustZone hardware is the separation of the two worlds in the processor itself. This is indicated by the *NS*-bit (Non-Secure-bit) in the *Secure Configuration Register* (SCR) of the processor. This bit can only be set in the Secure mode. When the NS-bit is 0 the processor is operating in secure world and when it is 1 the processor is operating in normal world. Two operating systems can run alongside each other using this architecture: One in the secure world and one in the normal world. As a result a special form of virtualization is created: There are two virtual environments that include virtual processors and virtual resources. Access to those virtual resources can be limited depending on the security status of the processor. The value of the NS-bit is used to signal the security status of communications on the AXI bus. This is in turn used by the peripheral to decide if it should act on a certain transaction.

A special state is created in the secure world to facilitate switching between the worlds. This state is called the *monitor mode*. The normal world can start this monitor mode by calling the *Secure Monitor Call* (SMC) instruction. Hardware exceptions such as interrupts can also cause a switch to the secure world. When such switch due to the SMC instruction or exceptions happens the monitor mode of the secure world is enabled. The monitor mode ensures that the state of the world it is leaving is saved and the state of the world it is entering is restored [9]. State data includes all processor registers and optionally additional information depending on the peripherals in the SoC [9]. ARM

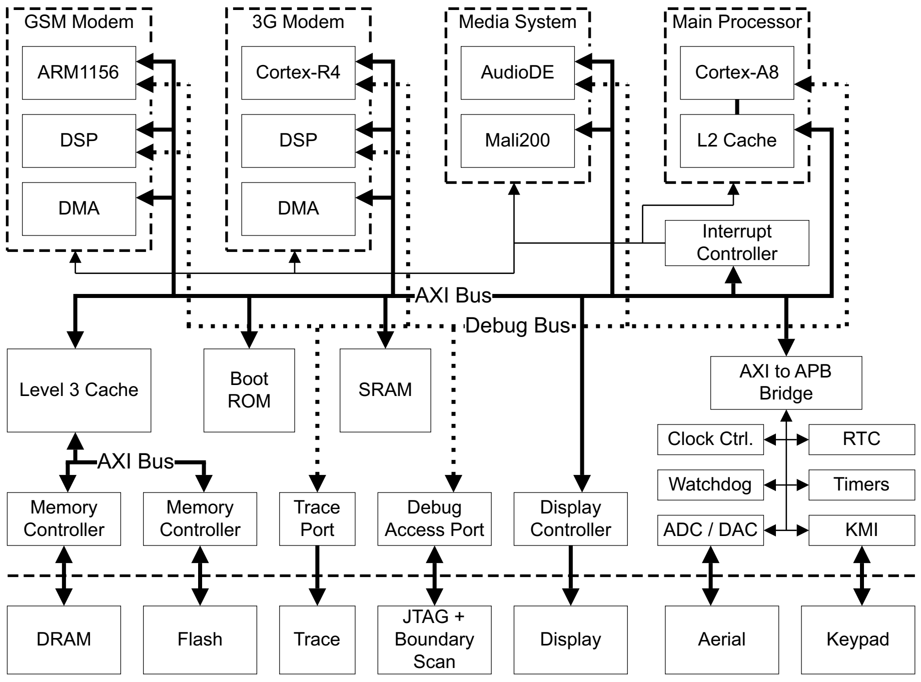

Figure 2.1: The ARM architecture and its AXI bus. Source: *Building a Secure System using TrustZone Technology.* ARM Limited, 2009. URL: `http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf`

recommends to enable the monitor mode to also handle the switch from the secure world to the normal world [9].

Processors can have multiple processor cores to allow multiprocessing. This may present additional difficulties to ensure that all data is kept safe between context switches. To simplify the switching, the secure world can be fixed to a single core or a certain number of cores. Note that, in contrast to dedicated security processors, the TrustZone hardware does not by default include tamper protection [9]. Since ARM only sells the designs to create the processors and does not make the chip itself manufacturers may include additional hardware features to offer better tamper resistance. Also note that while TrustZone provides hardware-based security features, the security of the whole system is also depending on some software features such as the sanity of the switching between the secure and normal world implemented in the monitor mode. A bad implementation could leak the register values that may contain sensitive information.

When an ARM processor with TrustZone support boots it starts by executing an application that is programmed in the on-chip-ROM in the secure world. This application is called the *bootrom*. The bootrom can be fixed by the processor manufacturer at design-time using a *Masked ROM* or by the customer of the manufacturer (the manufacturer of the system that incorporates the processor) by using write-locked flash. A public key is programmed in to the SoC using One-Time-Programmable (OTP) memory. This memory can only be written once. This is often achieved by burning fuses on the chip. The processes of burning the fuses is very hard to reverse but not impossible [60]. If the size of the OTP memory is limited one can resort to writing a hash of the public key in the OTP memory.

The public key or the hash of the public key is used for verifying the integrity and authenticity of the next step in the boot-process which is the *bootloader*. A bootloader is an application that loads the operating system during boot. It is normally stored in flash on mobile platforms and can be updated. If a hash is programmed in the OTP memory the public key itself is provided alongside the bootloader. The bootrom then verifies the public key first by comparing the hash of the public key to the hash stored in OTP [9].

A TrustZone compatible bootloader first starts the operating system that is running in the secure world. When the secure operating system has booted it starts the normal world operating system. Another commonly used setup is to have the secure world operating system to start a normal bootloader in the normal world that is not secured and can be changed by the user (see figure 2.2). This normal bootloader can then start a operating system in the normal world.

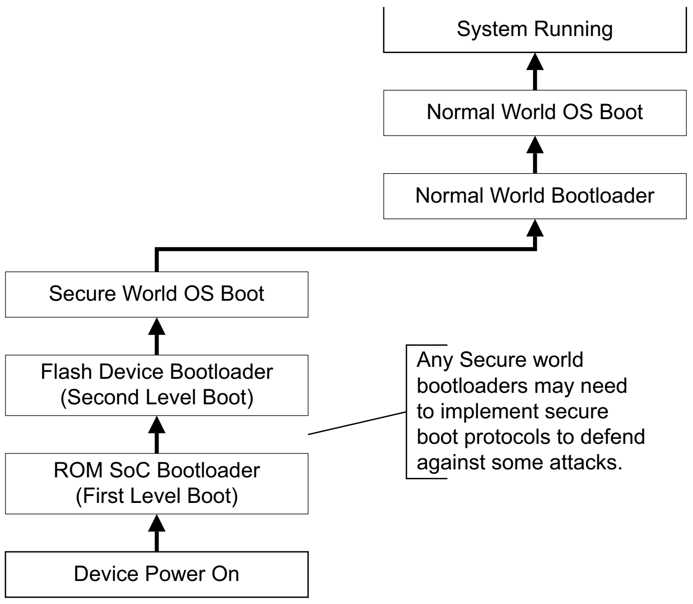
System Running
Normal World OS Boot
Normal World Bootloader
Secure World OS Boot
Flash Device Bootloader (Second Level Boot)
ROM SoC Bootloader (First Level Boot)
Device Power On
Any Secure world bootloaders may need to implement secure boot protocols to defend against some attacks.

Figure 2.2: Typical TrustZone boot process. Source: *Building a Secure System using TrustZone Technology*. ARM Limited, 2009. URL: `http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf`

2.2.2 Secure world operating systems and the TEE

As discussed in section 2.2.1 on page 9 TrustZone provides hardware-features to create a secure environment separate from the normal execution environment. However, the hardware-features that are provided do not implement or ensure all the requirements discussed in section 2.2 on page 8 that are needed for a TEE. As previously noted some functionalities (such as context switching between the two worlds) are left to the software running in the secure world to implement. The communication of data between the two worlds is left to the software running in both worlds to implement. However, hardware features such as the impossibility of the normal world to access the memory of the secure world allow this functionality to be implemented.

To allow multiple applications to be run in the secure world a secure world operating system (*secure world OS*) has to be implemented. We consider the secure world OS as the software that provides the TEE: It provides an execution environment for applications to run in. As discussed previously, applications running in the TEE are called *trustlets*. The secure world OS schedules resources between both the trustlets running in the secure world and the operating system running in the normal world. The secure world OS should handle both context switches (between trustlets in the secure world and between the secure and normal world). It should also ensure that no data is leaked during the context switches. Note that if an untrusted user is allowed to run trustlets in the TEE, also the security of context switches between trustlets in the TEE should not leak any information. Even if all trustlets are created by the same issuer this is still a good property to ensure. The separation between the two worlds each with its operating system is pictured in figure 2.3 on page 13.

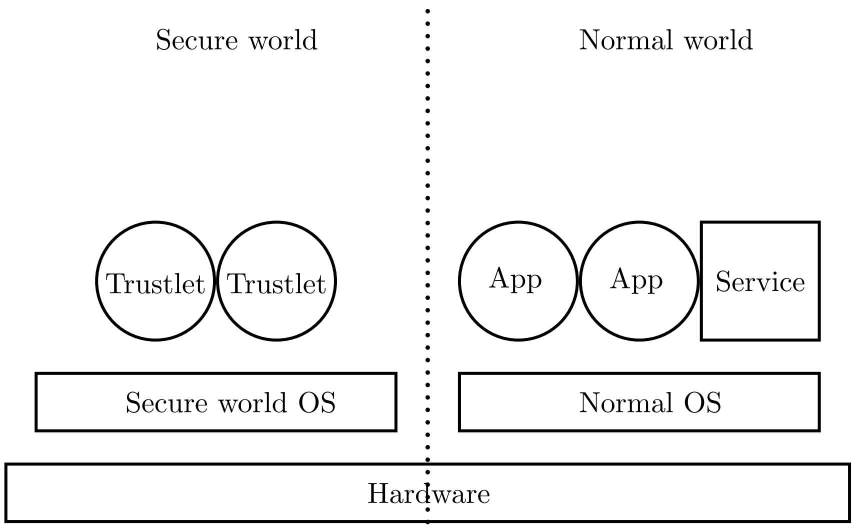

Figure 2.3: The separation of the hardware by TrustZone in two worlds.

A secure world OS is sometimes seen as an *hypervisor*. A term that is generally associated with running virtual machines; running multiple operating systems in their own environment on the same hardware. While the separation by TrustZone also provides two operating systems running in their own environment on the same hardware, the actual architecture is different. In the case of virtualization the hypervisor provides virtual hardware on which the (unlimited number of) operating systems run. The hypervisor sits in between the virtual hardware and the real hardware. In the case of TrustZone technology both the secure world OS and the OS running in the normal world can directly communicate with the hardware. However, some parts of the hardware are only accessible by the secure world.

In principle there is no limit to the complexity and functionality of the secure world OS running in the secure world. However, to reduce the attack-surface, the functionality is usually limited. There is only a small number of open source implementations of TrustZone compatible secure world OSes available [59]. In section 4.1 on page 51 a number of secure world OSes are discussed.

2.2.3 Alternatives providing secure computation

The ARM TrustZone is not the only solution to provide a secure execution environment. There are other solutions that have been used in mobile platforms before the adaption of the TEE. One notable solution is the *UUIC* (SIM-card) as *Secure Element* (SE). This is allows applications to execute in the secure environment of the UUIC.A nother frequently used solution is the Secure Element that is embedded (together with a NFC controller) in a large number of phones [40]. The Secure Element can be accessed by the operating system but also by an external NFC reader. This is a feature of some Secure Elements. The SE can provide an interface to allow an external NFC reader to directly communicate with the Secure Element [29]. The Google Wallet payment solution used the SE to store its sensitive information until recently [5]. The latest version of Google Wallet based on Android 4.4 uses the *Host Card Emulation* (HCE) API [29] that allow developers to emulate RFID/NFC cards in normal Android applications without using a Secure Element. A recent trial by a number of Dutch banks used the UUIC as Secure Element [18]. GlobalPlatform also defined a standard for communication with the Secure Element that can be found on GlobalPlatforms web-page[4]

The Near Field Communication research institution partly funded by NXP says that the Secure Element in the NFC controller uses a Java Card operating system [40]. As expected, this means that it is also possible to load applications on the Secure Element either embedded in the mobile platform or embedded in the UUIC. There are however a number of differences between the Secure Element solutions and a TrustZone-based solution:

[4]`http://www.globalplatform.org/specificationscard.asp`

- The Secure Element does not provide a trusted path. The secure world using the TrustZone technology has access to the display controller and peripherals as discussed earlier. This allows for local attestation and provides a trusted path with the user as discussed earlier. A Secure Element can only communicate with the main processor over a serial bus and cannot control the display controller.
- The Secure Element is based on a chip design that was designed for use in smart cards. As a result the Secure Element is physically tamper resistant [42]. In contrast, a TrustZone-enabled processor does not necessarily have any physical tamper protection [9].
- Smart cards are usually platforms with power constraints. As a result the computing power of a Secure Element is limited compared to the full computing power of all cores on a ARM chip available to the TrustZone-based TEE.
- Getting a certain application in Secure Element that is embedded in a UUIC is a challenging task. The developing party has to negotiate with every telecom provider their users use as each provider controls their own UUICs. Note that this is not an easy task. For example a small part of the user base can be using a foreign provider. A Secure Element that is embedded in the mobile platform itself or a TrustZone-based solution on the other hand is expected to be controlled by the hardware manufacturer that created the mobile platform. So the developing party has to talk to a party anyway, regardless of choosing a UUIC-based, embedded SE or TrustZone-based solution. However, for the embedded SE or a TrustZone solution the developing party only has to talk to the device manufacturer. On the other hand it may be harder for developing party to contact a global hardware manufacturer than some local telecom providers.

So while both the Secure Element and a TEE based on TrustZone Technology provide secure computation there are some major differences. The Secure Element has more physical security features that prevent attacks where the physical circuit of the chip is attacked. TrustZone Technology on the other hand provides a solution that is more powerful both in computing power and interaction with the user. Companies that used smart cards in the past may be more inclined to use the Secure Element because it is the same chip in another package. It provides them with a well-known secure environment that they are experienced with. Also since the ARM TrustZone technology is significantly more complex then a Secure Element it may be harder to verify the security of such a secure computation environment.

2.3 The Android operating system

Android is a operating system developed by the Open Handset Alliance led by Google. It was first released in 2007 [3]. The operating system is based on a Linux kernel that is modified to better fit a mobile operating system. While the Android operating system and its packages are open source, only at the release of a new final version the source code is released. Ongoing development is not opensourced. Most applications are written in Java but C++ is also supported. Most OS services (as opposed to apps) on Android are written in C++. Everybody can use the Android operating system as operating system on their devices. This means that Android phones can be bought from a large number of manufacturers. Some manufacturers supply a Android experience that looks and feels like the versions released by the Open Handset Alliance, for example the Nexus phones that are released by LG and Samsung in collaboration with Google. Others only use the Android OS as basis and modify the experience and features.

The directory layout of the file system for Android is somewhat different from a usual Linux operating system:

- `/data` is used to store the data of all applications and services running on the operating system in.
- `/data/data` is the location for applications to store their data. Each application gets its own directory that is named using the application identifier (`com.company.example`).
- `/sdcard` is the location where the SD-card (if present in the system) is mounted. The internal storage is limited but faster on older Android devices so developers had to chose whether the stored the data internally or on the SD-card. Most recent Android devices have larger internal flash storage so they do not need a SD-card anymore. On systems that have internal flash storage and no SD-card slot the `/sdcard` path is symlinked to `/storage/emulated/legacy`.

The (emulated) SD-card directory can be accessed over USB to write and read all files from it. A number of directories in `/sdcard` are accessible by all applications such as `/sdcard/DCIM` where pictures are stored and `/sdcard/Music` for storing Music. The `/sdcard/Android/data` directory is where applications can store application specific data (like the `/data/data` directory). As with the internal application-data directory each application gets its own directory that only can be read by that application. However, all contents of the (emulated) SD-card can be read and modified over USB. This also includes the application data that is stored in `/sdcard/Android/data`, so caution should be used when storing data on `/sdcard`.

Applications have a `AndroidManifest.xml` file that describes the contents of the application and the permissions that the applications requests [55]. A user can not (at the moment) deny a single permission in the set of permissions requested by the application. The requested permissions are showed before the application is installed by the user or when a updated version of an application requests additional permissions. After installation there is no way for a user the deny a certain app certain permissions other than un-installing the application. A list of all permissions available can be found in the Android API documentation[5].

[5]`http://developer.android.com/reference/android/Manifest.permission.html`

To control file-access and to separate applications the Android operating system assigns each application its own user-id. Internal services also run under separate user-ids. This is different from a normal Linux system where each user is allocated a user-id and all applications that the user runs run under that user-id. The allocation of user-ids to applications can be found in `/data/system/packages.list` and could for example include an entry:

```
com.simplendi.KeyStorageTest 10102
```

Here we can see that the application with package name `com.simplendi.KeyStorageTest` (which is the test application we will use in the next chapter) has user ID 10102.

The Android operating system uses a specially developed virtual machine called Dalvik[6] as basis for the applications and services on their platform. This virtual machine is based on the Java language. As a result a number of Java API's are ported to Android and a number of new APIs are added. The ported APIs can be found under the `java.*` packages. The Java APIs provide support for cryptographic operations in the `java.security` package. Additional functionality, that is not available in the default Java APIs, is implemented in the `android.security` package. The Android API is not fixed and is evolving over time. The API version is indicated by the *API Level*. The first Android release has API Level 1 and the latest Android release (Android 4.4.3) has API level 19. An overview of the API levels and the associated Android versions can be found on the Android Developer web-page[7].

[6]`https://source.android.com/devices/tech/dalvik/index.html`
[7]`http://developer.android.com/guide/topics/manifest/uses-sdk-element.html#ApiLevels`

3 Secure key storage

In this chapter the secure key storage solutions for Android are analyzed. The chapter is divided into a number of sections. Section 3.1 describes the APIs that are available on Android OS to provide cryptographic operations and key storage. Section 3.2 describes how and using what criteria the key storage solutions are evaluated. In the next sections we will evaluate each solution described in 3.1 using the criteria discussed in section 3.2. Each section is divided in two parts: first the evaluation results and how they are achieved are discussed, then in the background section a more elaborate explanation is provided with more details on how the evaluated solution works.

3.1 APIs providing key storage

The first version of Android, API level 1, already provided cryptographic operations and key storage. It has the `java.security.KeyStore` class that provides a standardized interface to store key material. The `KeyStore` class only provides an interface to store keys and facilitates getting an instance of a `KeyStore`. The actual storage of keys is provided by different *key store types*. To accomplish this each key store type is defined in a class that provides an `java.security.KeystoreSpi` interface. This interface is defined as an abstract class. The `KeyStore` class uses the methods provided by the `KeyStoreSpi` interface and using that methods provides a uniform method to store keys to the programmer.

A key store of a certain type can be initialized by using the `getInstance(String type)` method of the `KeyStore` class. There are multiple key store types provided by the Android API:

- `BKS`: This stands for *Bouncy Castle key store*. Bouncy Castle is a cryptographic library for Java[1]. This key store type is provided on all Android versions encountered. However, the Android API provides a limited version of this library [21]. Many functions and classes are removed as the Android developers considered them unnecessary for early versions of Android. For example the APIs needed to create certificates are removed. The full version of the Bouncy Castle library, however, is also available for Android. To avoid naming conflicts this library for Android is called Spongy Castle[2] and can be included in any application. The `KeyStore`

[1]`https://www.bouncycastle.org/`
[2]`http://rtyley.github.io/spongycastle/`

class has a static `getDefaultType()` method to return the default key store type on the machine. The default key store type on all the machines analyzed is `BKS`. Keys can be stored using file-based key stores. The interface for doing this is as discussed in section 3.2.3 and section 3.1 almost the same as the newer hardware based interface. The difference lies in the implementation. The Bouncy Castle library uses a single file for storing the keys.

- `AndroidKeyStore`: This type is added in API Level 18. It communicates with a service called `KeyStore`[3] service using *Inter Process Communication* (IPC). The `KeyStore` service is started when the device boots. Manufacturers can develop drivers for their hardware that communicate with this service to provide hardware-based secure storage. If no drivers are available Android defaults to a software implementation. The following variants of `AndroidKeyStore` will be evaluated:
  - Software-based: Not all hardware that run Android 4.3 or higher support hardware-based key storage. Some ARM chips simply do not have TrustZone support and some device manufacturers do not have the ability to create applications for the TEE. To be able to offer the same `AndroidKeyStore` with the same interface on all platforms the developers created a software-based equivalent of the `KeyMaster` trustlet that runs in the TEE. Note that this software-based `AndroidKeyStore` is not available on devices that have a hardware-based `AndroidKeyStore`.
  - Hardware-based on Qualcomm hardware: Qualcomm created a closed source keymanager trustlet in the TEE and driver for Android.
  - Hardware-based on Texas Instruments (TI) hardware: Just like Qualcomm also Texas Instruments also provides a closed source keymanager trustlet and driver for Android.

Note that while the `AndroidKeyStore` variants are called *software-based* and *hardware-based* all solutions are of course based on both software and hardware. With *software-based* and *hardware-based* we mean that hardware-based or only software-based security features are used to protect the key store. Next to the `KeyStore` types, key storage on the Secure Element will also be discussed. While there is no key store interface to use the standard APIs to store keys on the Secure Element, it is possible to use the Secure Element to store keys. We will see more about this in section 3.7.

Besides the key store type a *provider* can be optionally defined for some classes. This defines what library is used for cryptographic operations. The Spongy Castle library uses this to interact with the Bouncy Castle key store type. When creating an instance of the Bouncy Castle key store type Spongy Castle is defined as provider. Subsequently keys compatible with the Spongy Castle library are provided. Note that not all combinations of providers and key store types are supported, some key store types do not

[3] `https://android.googlesource.com/platform/system/security/+/master/keystore/keystore.cpp`

support providers at all. This can be the case for key store types that only provide a interface to manage the keys but do not provide the actual key material. An example of this is the `AndroidKeyStore` key store type that can run its cryptographic operations outside of application process on other hardware as we will see in section 3.4 on page 30.

API level 14 added a new `KeyChain` class. This is an high-level API to provide asymmetric key storage. In contrast to the `KeyStore` class the `KeyChain` class interacts with the user using an graphical interface. For example the user can be asked confirm key generation. When a private key is required for operation the user is also asked to confirm. This class, however, also provides two interesting static methods. The `isKeyAlgorithmSupported(String algorithm)` can be used to test if the device supports a certain algorithm. In the latest API level 19 release a new static method was added to the class, `isBoundKeyAlgorithm(String algorithm)`. This method “Returns true if the current device's `KeyChain` binds any `PrivateKey` of the given `algorithm` to the device once imported or generated.” according to documentation[4].

In addition to private keys it is also possible to store symmetric keys in the `KeyStore`. To support this the key store has a `SecretKeyEntry`. Secret keys can be created using the `javax.crypto.spec.SecretKeySpec` class. Instances of this class are initialized by a byte-array for the secret key and a string indicating the algorithm that is associated with the secret key.

3.2 Method

3.2.1 Introduction

To evaluate how the different key storage types work and what security properties they provide a number of sources are used: The datasheets of the ARM processors and the secure elements are used to evaluate hardware features. Another good source of information is the open source code that is provided by the *Android Open Source Project* (AOSP). The git repositories[5] of AOSP provide a large part of the source code that is used by phones running Android OS. All resources that are used to build the binary images that run on the Google Nexus phones are also available in the AOSP repositories. This, however, not only includes source code but also binaries for some manufacturer specific code that is not open source. Since the licensing of the AOSP code requires that changes to the code are also made public, the source code for phones that are not developed under the supervision of Google is also available. For example Samsung provides their open source code at Samsungs Open Source Release Center[6].

[4]`http://developer.android.com/reference/android/security/KeyChain.html`
[5]`https://android.googlesource.com/`
[6]`http://opensource.samsung.com/`

By using root privileges on the platforms that will be used for the evaluation, the inner workings of the storage methods can be analyzed. However, root access is by default disabled on Android phones. To gain root privileges first the bootloaders of the phones are unlocked. This allows the user to flash images that are not signed by the manufacturer. Next a special partition, the recovery partition that stores a recovery application is flashed with ClockWorkMod recovery application that can be downloaded from the ClockWorkMod website[7]. The phone is then booted into the recovery application to install SuperSU[8]. This installs the `su` executable on the phone that runs applications under root privileges.

The smartphones that will be used for the evaluation are listed in table 3.1 on page 21. Motorola Mobility was bought by Google in August 2011 and subsequently sold to Lenovo in January 2014. The MSM8226 SoC in the Moto G actually has TrustZone support. Drivers are even provided by the AOSP git repositories[9] that is used to create Android images. This is not enabled in the Android 4.3 image supplied on the phone. The update to Android 4.4.2 enabled hardware backed key storage using TrustZone on the Moto G. The Galaxy Nexus hardware also has TrustZone support but it is not enabled in the images supplied by Google. According to a commit message[10] the TrustZone key storage was disabled due to a power-usage bug. It can be enabled as shown in section 3.5.2.

Nickname	**Manufacturer**	**Model name**	**SoC**	**Android version**	**TrustZone support**
Nexus 5	LG Electronics	LG-D821 16GB	Qualcomm Snapdragon 800 MSM8974	4.4.2	Yes
Nexus 4	LG Electronics	LG-E960 16GB	Qualcomm Snapdragon 600 APQ8064	4.4.2	Yes
Galaxy Nexus	Samsung Electronics	GT-I9250	Texas Instruments OMAP 4460	4.3	Yes*
Nexus S	Samsung Electronics	GT-I9020T	Samsung Exynos 3 Single S5PC110	2.3.6	No
Nexus S	Samsung Electronics	GT-I9020T	Samsung Exynos 3 Single S5PC110	4.1.2	No
Moto G	Motorola Mobility	SM3719AE7F1	Qualcomm Snapdragon 400 MSM8226	4.3	No
Moto G	Motorola Mobility	SM3719AE7F1	Qualcomm Snapdragon 400 MSM8226	4.4.2	Yes

Table 3.1: Phones used in the evaluation

These phones are chosen because they provide an overview of both high-level and low-level phones running recent versions of Android and to evaluate the older 2.3 Android version that is still commonly used today.

[7] `https://www.clockworkmod.com/`

[8] `http://download.chainfire.eu/396/SuperSU/UPDATE-SuperSU-v1.94.zip`

[9] `https://android.googlesource.com/platform/hardware/qcom/keymaster/+/27e66b721ed6472fb319b0421ca460713a1b2df5`

[10] `https://android.googlesource.com/device/samsung/tuna/+/b74801dc22bb4945ddf79b2e12e6328a862d68c3`

3.2.2 Criteria

For the evaluation and comparison of the different cryptographic key storage implementations I defined three requirements for evaluation:

$R_{device-bound}$ The private key material stored in the key store can not be extracted by an attacker to be used outside of the device that generated it. This defines that keys are bound to the device.

$R_{app-bound}$ The private key material stored in the key store can only be used by the application and on the device that generated it. This defines that keys are bound to the device and the application.

$R_{user-consent}$ Using the private key material requires explicit permission from the user and can not be used without explicit consent. This defines user consent.

The requirements $R_{device-bound}$ and $R_{app-bound}$ are evaluated using three assumed attacker models:

$A_{outside-no-root}$ The first attacker model is an attacker that is able to run and install his own application on the phone with any of the permissions that an application can request. This models a rogue application (or update of an application) that a user can install from an app store such as the Google Play Store. See section 2.3 on page 15 for more information about permissions in Android.

$A_{outside-root-no-memory}$ The second is an attacker that has root permissions and can use all data stored on the phone. We assume that these permissions are gained by either an exploit or are given to the attacker by using an application that asks for root permissions on a rooted phone. This models that either the attacker uses an exploit to gain root permissions on the device or that the attacker has the ability to run an application with root permissions. Note that this may seem as a situation that does not happen a lot, however, many users today enable the root permissions on their phone to work around the permission model. There are even applications in the Google Play Store that require root permissions. For example the Titanium Backup in the Play Store[11] application that is used to backup a phone requires root permissions.

$A_{outside-root-memory}$ The third attacker model assumes that an attacker has root permissions and can use all data stored on the phone but also has access to data that is only temporarily stored on the phone such as data stored in memory. For example a PIN that is used to unlock the phone but that is not stored phone.

For the requirement $R_{user-consent}$ there is only one attacker model:

[11] `https://play.google.com/store/apps/details?id=com.keramidas.TitaniumBackup`

$A_{inside-no-root}$ This attacker model assumes that the actual developer of an app is or has become malicious. This is particularly interesting in the case where the developer of an application has interest in using the key pair that is stored for the app. An example of this can be an e-mail application that has a key pair to sign messages. The signature may have legal-effect that may be abused.s

For all criteria the described attacker models are evaluated. For each combination of a criteria and an attacker model the result is either:

✓ An attacker using the attacker model can not violate the requirement.

× An attacker using the attacker model can violate the requirement.

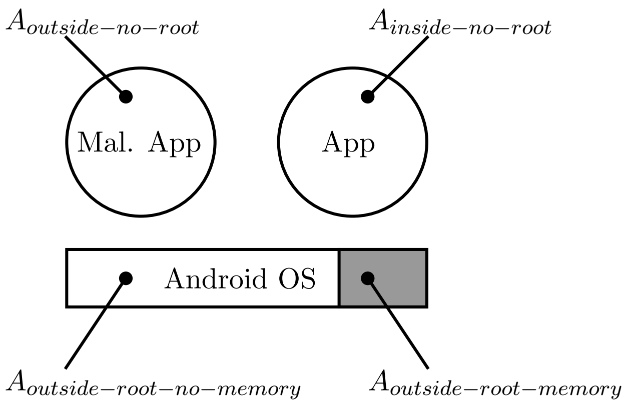

Figure 3.1: Schematic overview of the attacker models

All attacker models are shown in a schematic overview in figure 3.1. In this figure we see two applications, *App* is the genuine owner of the keys that need to be protected, *Mal. App* is an malicious application controlled by an attacker. The attackers that have root permissions can access the operating system controls and are therefore pictures to be inside the operating system. Although in fact attackers could also gain root credentials by using an application.

The solution that scores ✓ on all requirements for all attacker models is clearly the best solution. Note that requirement $R_{app-bound}$ is a more specific variant of requirement $R_{device-bound}$. So if $R_{app-bound}$ is satisfied then $R_{device-bound}$ is also satisfied. Also note that the attacker models for $R_{app-bound}$ and $R_{device-bound}$ are increasing in the privileges the attacker has, $A_{outside-no-root}$ has the smallest number of permissions and $A_{outside-root-memory}$ the most. If an requirement can be violated by an attacker having attacker model $A_{outside-no-root}$ than it can also be violated by an attacker having attacker model $A_{outside-root-no-memory}$ or $A_{outside-root-memory}$. The case where a user is tricked to give for example his password is not considered an attack in this use case. However, this may be a realistic threat.

3.2.3 Evaluation

To test the key stores on Android I created the *KeyStorageTest* application. The code used for the `MainActivity` of this application can be found in Appendix 1 on page 70. On startup the *KeyStorageTest* application first checks if the cryptographic algorithms RSA, ECDSA and DSA are bound to the device (if the function is available). This check is done using the `isBoundKeyAlgorithm(String algorithm)` function of the `KeyChain` class. This function returns `true` for all algorithms that are bound to the device. This function is implemented by the device manufacturer who should guarantee that the keys are bound to the device if the function returns `true`. Next the application gets all keys stored in the key store by alias. This alias is a string defined by the programmer that is used to identify a key (pair). By using the buttons at the bottom of the application window a new key pair can be generated and deleted. The key pair can also be used for signing.

The KeyStorageTest application is used to generate a RSA key pair using the code in listing 3.1 on page 25. This code generates a keypair using the RSA algorithm with a key-size of 2048-bits. The `KeyPairGenerator`, by default, also generates a (self-signed) certificate. To allow this the subject, the serial number and the start and end of the validity period have to be declared. Note that the `setKeySize`-function is only supported from API level 19, API level 18 only supports RSA keys of 2048-bits in size.

The cryptographic functionalities on older Android versions are implemented by the Bouncy Castle library[12] as noted in section 3.1. This Bouncy Castle library can not be used with the same interface as the `AndroidKeyStore`. A version of the `KeyStorageTest` application for Android versions that do not support `AndroidKeyStore` is therefore created with the following modifications:

- The `AndroidKeyStore` key store type is not available so the Bouncy Castle type has to be used. A key store of this type can be initialized using the `initialize`-method by specifying the `BKS` (Bouncy castle Key Store) type instead of `AndroidKeyStore`. To indicate that we want to use the Spongy Castle library for the cryptographic operations we have to add `SC` as second parameter to the initialization function to indicate the cryptographic provider.
- The key store data is stored on a per-application basis and the application is responsible for loading and storing the data. The application should load the `KeyStore` by calling the `load` function of the key store with an `InputStream` as argument. This `InputStream` can for example be a `FileInputStream` that reads the contents of a file. As a second argument a byte array can be provided as password. A random password stored in a file is used here. Note that since loading and storing of the key store is left to the programmer to implement he can modify some parts of the behavior. For example the key store data could be

[12]`https://www.bouncycastle.org/`

```
KeyPairGenerator rsaKeyGen;

try {
    rsaKeyGen = KeyPairGenerator.getInstance("RSA", "AndroidKeyStore");
} catch (Exception exception) {
    writeToLog(exception.toString());
    return;
}

KeyPairGeneratorSpec rsaKeyGenSpec = new KeyPairGeneratorSpec.Builder(
    this)
        .setAlias("TestKeyPair")
        .setSubject(new X500Principal("CN=test"))
        .setSerialNumber(new BigInteger("1"))
        .setStartDate(new Date())
        .setEndDate(new GregorianCalendar(2015, 0, 0).getTime())
        .setKeySize(2048)
        .build();

try {
    rsaKeyGen.initialize(rsaKeyGenSpec);
} catch (InvalidAlgorithmParameterException exception) {
    writeToLog(exception.toString());
    return;
}

rsaKeyGen.generateKeyPair();
```

Listing 3.1: Generating a KeyPair

encrypted additionally and multiple key stores can be created for multiple users of an application.

Two instances of the `KeyStorageTest` application are installed on devices. One instance is used to generate a key pair. The goal then is to give the second instance control over the key pair that is generated by the first instance. This instance should then generate a valid signature over predefined data. If this is possible it violates requirement $R_{app-bound}$. To test if requirement $R_{device-bound}$ is violated we copy the data (if available) from the phone and try to generate a valid signature on another phone. This is repeated for each of the three attacker models with increasing privileges. To verify requirement $R_{user-consent}$ we look at the APIs to see if they have the possibility to require user input or user consent to use a key (pair). If the APIs offer a way to require user consent I try to use the key without user consent. If this is possible or if the APIs do not offer any way to require user consent $R_{user-consent}$ is violated.

3.3 Key storage using Bouncy Castle

3.3.1 Evaluation

Because the test application uses best practices that include storing the keystore-file in the application specific data directory (see section 3.3.2 for more details) the keystore-file can not be accessed without root permissions. Since an attacker without root permissions can not access the keystore-file, the attacker can not read the private keys. So requirement $R_{device-bound}$ and $R_{app-bound}$ are satisfied for attacker-model $A_{outside-no-root}$.

With root-permissions the application-specific data directory where the keystore-file is stored by the test application can be accessed. This application directory also contains the file that stores the password that is provided while storing the keystore-file. By copying both files to another device running the same application I could successfully create a valid signature using the private key stored in the key-store. I did not have to inspect the memory of the device because all data needed was stored in files. So an attacker using attacker model $A_{outside-root-no-memory}$ can successfully violate requirement $R_{device-bound}$ and $R_{app-bound}$ if a stored password is used.

If an user-provided password is used it is still possible to copy the keystore-file to another device for an attacker that gained root permissions. However, the private key entries in it can not be used without the user-provided password. This password is not stored in plain text on the device. A way to learn this password is to intercept it in memory when the user enters it. Only an attacker using attacker model $A_{outside-root-memory}$ is able to do this. So only attacker model $A_{outside-root-memory}$ can violate criteria $R_{device-bound}$ and $R_{app-bound}$ if a user-provided password is used. However, another possibility is to brute-force the password. The entropy of the password used to encrypt the key store may be limited. Even a password that only consists of 4 or 5 digits is regularly used on phones since entering a password that is complex and long on a mobile device is cumbersome. Low entropy passwords are easily brute-force-able. When such password is used an attacker that has access to the keystore-file can brute-force the password. As a result, when a low entropy password is used, an attacker using attacker model $A_{outside-root-no-memory}$ may be able to gain access to the data stored in the key store.

When no user-provided password is used an attacker using attacker model $A_{inside-no-root}$ can use the stored password to do cryptographic operations. So requirement $R_{user-consent}$ is not satisfied for this attacker model. There is no possibility to use the private keys stored in the key store without learning the password. So when a user-provided password is used an attacker using attacker model $A_{inside-no-root}$ can not access the keys without the password. However, if the password has a low entropy an attacker using attacker model $A_{inside-no-root}$ may be able to brute-force the password.

Assuming that best practices are used to store the keystore-file and passwords this brings us to the following conclusion:

- An attacker that learns the password used to encrypt the keystore-file and has access to the keystore-file can use it on another device. If the password is user-provided and has sufficient entropy an attacker using attacker model $A_{outside-root-memory}$ can violate requirement $R_{device-bound}$ and $R_{app-bound}$. If the password is stored in the application data directory or a low entropy password is used attacker model $A_{outside-root-no-memory}$ is needed to violate requirements $R_{device-bound}$ and $R_{app-bound}$.
- If the password is user-provided and has sufficient entropy to prevent brute-forcing an attacker using attacker model $A_{inside-no-root}$ can not violate $R_{user-consent}$. If the password, however, has low entropy an attacker using attacker model $A_{inside-no-root}$ may be able to brute-force the password. If no user-provided password is used the attacker can run cryptographic operations using the stored password. So criteria $R_{user-consent}$ is not satisfied in the case of a stored password for attacker model $A_{inside-no-root}$.

3.3.2 Background

According to the Bouncy Castle documentation the default format only protects against tampering but not against inspection[13]. This is clearly the case for the certificates stored in the key store. We can simply read the certificates using openssl's asn1parse tool as shown in listing 3.2 on page 28. The self-signed certificate of the first key is stored at an offset of 70 bytes. Note that if you add the `-dump` parameter you can see the actual signature and public key bytes.

To ensure the integrity of the key store a password needs to be provided while storing the key store and when the integrity needs to be verified. Without this password the

[13]`http://www.bouncycastle.org/specifications.html`

```
$openssl asn1parse -inform der -in keystore -offset 70
    0:d=0  hl=4 l= 658 cons: SEQUENCE
    4:d=1  hl=4 l= 378 cons: SEQUENCE
    8:d=2  hl=2 l=   1 prim: INTEGER           :01
   11:d=2  hl=2 l=  13 cons: SEQUENCE
   13:d=3  hl=2 l=   9 prim: OBJECT            :sha1WithRSAEncryption
   24:d=3  hl=2 l=   0 prim: NULL
   26:d=2  hl=2 l=  15 cons: SEQUENCE
   28:d=3  hl=2 l=  13 cons: SET
   30:d=4  hl=2 l=  11 cons: SEQUENCE
   32:d=5  hl=2 l=   3 prim: OBJECT            :commonName
   37:d=5  hl=2 l=   4 prim: PRINTABLESTRING   :Test
   43:d=2  hl=2 l=  30 cons: SEQUENCE
   45:d=3  hl=2 l=  13 prim: UTCTIME           :140407130050Z
   60:d=3  hl=2 l=  13 prim: UTCTIME           :141231000000Z
   75:d=2  hl=2 l=  15 cons: SEQUENCE
   77:d=3  hl=2 l=  13 cons: SET
   79:d=4  hl=2 l=  11 cons: SEQUENCE
   81:d=5  hl=2 l=   3 prim: OBJECT            :commonName
   86:d=5  hl=2 l=   4 prim: PRINTABLESTRING   :Test
   92:d=2  hl=4 l= 290 cons: SEQUENCE
   96:d=3  hl=2 l=  13 cons: SEQUENCE
   98:d=4  hl=2 l=   9 prim: OBJECT            :rsaEncryption
  109:d=4  hl=2 l=   0 prim: NULL
  111:d=3  hl=4 l= 271 prim: BIT STRING
  386:d=1  hl=2 l=  13 cons: SEQUENCE
  388:d=2  hl=2 l=   9 prim: OBJECT            :sha1WithRSAEncryption
  399:d=2  hl=2 l=   0 prim: NULL
  401:d=1  hl=4 l= 257 prim: BIT STRING
  662:d=0  hl=2 l=   0 prim: EOC
```

Listing 3.2: Reading a certificate from the Bouncy Castle keystore-file

integrity of the whole key store can not be verified. The private and secret keys stored in the key store are, however, protected against inspection. So while the key store is not protected against inspection the private key material is. They are encrypted using the `pbeWithSHAAnd3-KeyTripleDES-CBC` encryption scheme, as defined in PKCS5 standard [31], using either the password that was also provided to verify the integrity of the key store or using another password. This password does not have to be stored on the device. For example the user can be asked for the password when running cryptographic operations using the private key.

There is no vulnerability known for the encryption algorithm used to encrypt the private keys. So when the password is not stored on the device the user has to provide his password in order to run cryptographic operations. This offers some protection against attackers that have root credentials if a password of sufficient entropy is used: The only way to learn the password is by inspecting the memory while the user enters his password. This may be hard. If the entropy of the password is not sufficient an attacker that has root credentials can brute-force the password.

As discussed before the key store has to be stored by the programmer. Care should be taken where and how to store the keystore. If the keystore-file is stored on the publicly accessible `/sdcard` storage (which is either a real SD-card or a part of the embedded flash) any application can read the encrypted keystore-file. This is an issue if the entropy of the password used to encrypted the keystore-file is low, causing it to be brute-forceable. Additionally if the password is hardcoded in the application source or also stored on the public storage, an attacker can either decompile the application or read the key from public storage and using this password decrypt the private key. So when the password and key store are stored in the wrong location no protection is provided and even an attacker that did not gain root permissions can read the private key and use the private key, even on a other device.

However, when the keystore-file is handled with care the private key can not be used by an attacker that did not gain root permissions. The recommended approach is to store the keystore-file in the private data directory of the application (located under `/data/data/APP_ID/files/`) and to only allow the user-id of the application to access the key store file. Note that both storing the keystore-file in the right location and using a user-provided or securely stored password with sufficient entropy is also recommended. Only storing a password with sufficient entropy that encrypted the keystore-file in the private data directory of the application has almost the same security properties. The only difference is that an attacker that does not have root permissions can read the certificates stored on the key store. These certificates are normally not private data. In both cases an attacker that does not have root permissions can not read the private keys stored in the keystore-file. Yet another possibility is to ask the user for the password that encrypts the private keys in the keystore. This password should then not be stored on the device. An attacker that gained root permissions can still copy the keystore-file to another device. He does, however, need the password used to encrypt the private key to actually use it.

If an attacker learns the password he can decrypt the key store and use it outside of the device. This can be done either by gaining root permissions reading the password from a file stored at a recommended location (attacker model $A_{outside-root-no-memory}$) or by intercepting the password entered by the user from memory (attacker model $A_{outside-root-memory}$) or from a fake application. By default there is nothing that binds the key store to the device or to the application other than the permissions of the keystore-file. It is possible to include a device-specific identifier in the password used to encrypt the keystore-file. This prevents against attacks where the attacker uses the same application on both devices and does not decompile the application. However, applications can be decompiled very easy. Decompiling the application allows an attacker to learn about the device-specific identifier used and to reproduce it.

3.4 Key storage using AndroidKeyStore using the TEE on Qualcomm devices

3.4.1 Evaluation

The `AndroidKeyStore` on Qualcomm devices adds two files in the `/data/misc/keystore/user_0` directory for each key pair when a new key pair is generated using the APIs used by the *KeyStorageTest* application:

- A `USRPKEY` file that stores the key pair parameters including the private key.
- A `USRCERT` file that stores the certificate.

Both files have the following naming format: *(user-id of the app)*`_USRPKEY_`*(key entry alias)* and *(user-id of the application)*`_USRCERT_`*(key entry alias)*. For example `10102_USRPKEY_TestKeyPair`. The user id of the application is the logical user-id of the Android operating system under which the application is running. As discussed in section 2.3 on page 15 each Android application is allocated its own user id. The key entry alias can be chosen by the programmer using the `setAlias(key_alias)` method of the `KeyPairGeneratorSpec.Builder` as shown in 3.1 on page 25.

The `/data/misc/keystore/user_0` directory is not accessible by a non-root user and other apps can not access the private key material of an application. So requirement $R_{device-bound}$ and $R_{app-bound}$ are guaranteed for attacker model $A_{outside-no-root}$. It is interesting to see that the application user-id is used in the filename of the entry-files. Using root permissions an attacker can rename or copy the files to new files in the same directory on the same device with the user-id of a second malicious application. For example here we copy the entry files from the first to the second `KeyStorageTest` application:

```
cp 10102_USRCERT_TestKeyPair 10101_USRCERT_TestKeyPair
cp 10102_USRPKEY_TestKeyPair 10101_USRPKEY_TestKeyPair
```

The private keys will then directly be accessible to the other application. This was verified by renaming the files from the user-id of the first `KeyStorageTest` application to the user-id of the second `KeyStorageTest` application as shown above. Afterwards a valid signature could be generated by the second `KeyStorageTest` application using the key pair that was generated by the first application. So clearly $R_{app-bound}$ is not satisfied for attacker-model $A_{outside-root-no-memory}$. Copying the files to another device does not work, as discussed in section 3.4.2 the private key is probably encrypted using a device specific key that is stored in the TEE. So criteria $R_{device-bound}$ is satisfied for all attacker models.

The `AndroidKeyStore` does not support the encryption of the key store or entries using a password. As a result no user-consent can be required to use a private key. An attacker can access the private key material at any time. So criteria $R_{user-consent}$ is not satisfied.

This brings us to the following conclusion:

- An attacker that has root credentials can easily use the keys of other applications on the same device by renaming the keystore-files. So $R_{app-bound}$ is not satisfied for attacker model $A_{outside-root-no-memory}$ and $A_{outside-root-memory}$.
- Key pairs can not be used outside of the device because, as we will see later in section 3.4.2, the private data is encrypted with a device-specific key stored in the TEE that can not be retrieved. So requirement $R_{device-bound}$ is met for all attacker models.
- There is no way to require any input from the user for operations that use the private keys stored in the key store. So criteria $R_{user-consent}$ is not satisfied.

3.4.2 Background

The interface for the `KeyStore` service is defined in `IKeystoreService`[14]. To communicate with the actual trustlet, that is running in the TEE created by the TrustZone technology, Qualcomm created a driver that is called from the *KeyStore* service. This driver is open source and can be found in the AOSP repositories as `keymaster_qcom`[15]. The architecture of this setup is illustrated in figure 3.2 on page 32. In this figure two applications *App 1* and *App 2* are shown. They communicate with the `AndroidKeyStore` (*KS* in the figure) which in turn communicates via shared memory with the `keymaster` (*KM* in the figure) trustlet running in the TEE. This trustlet runs the actual cryptographic operations such as generating private keys and signatures using the generated private keys.

[14]`https://android.googlesource.com/platform/system/security/+/master/keystore/IKeystoreService.cpp`

[15]`https://android.googlesource.com/platform/hardware/qcom/keymaster/+/master/keymaster_qcom.cpp`

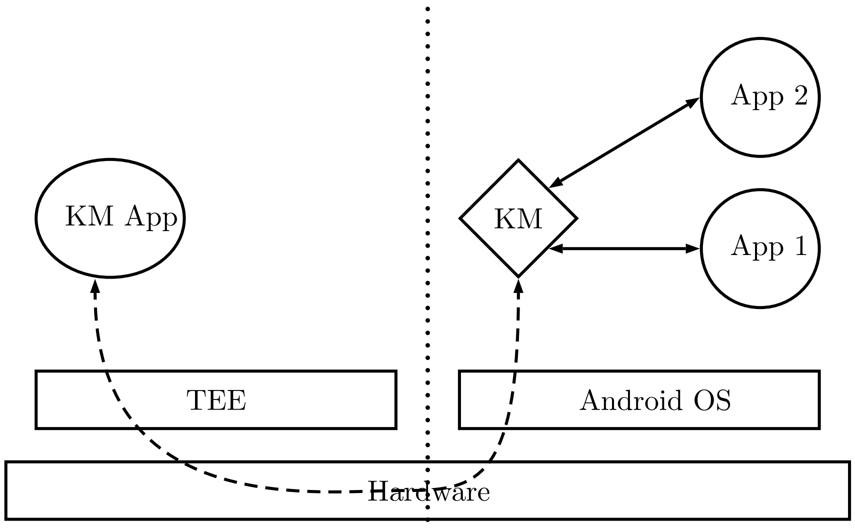

Figure 3.2: Architecture of the AndroidKeyStore on Qualcomm devices

```
dev->context = (void *)km_handle;
ret = (*km_handle->QSEECom_start_app)((struct QSEECom_handle **)&
    km_handle->qseecom,
                     "/vendor/firmware/keymaster", "keymaster", 4096*2);
if (ret) {
    ALOGE("Loading keymaster app failed");
    free(km_handle);
    return -1;
}
```

Listing 3.3: Loading the keymaster binary

During the boot the `keymaster` trustlet is loaded into the TEE. To allow this, the actual keymaster binary (the compiled trustlet) is stored in the `/firmware/image/` or `/system/vendor/firmware/keymaster/` directory separated in a number of files: `keymaster.mdt` and `keymaster.b00-keymaster.b03`. Google also provides the binaries for the Nexus devices on their website[16]. Loading the binaries of the trustlet is implemented in `keymaster_qcom.cpp`[17] as shown in Listing 3.3. This function is called when the `Keystore` service starts.

To gain more information about the keymaster trustlet that is running in the TEE I analyzed the `keymaster` binaries of the Moto G running Android 4.4.2. The first observation was that the `.mdt` files is exactly the same as the concatenation of `.b00` and `.b01` files. The `.b00` file start with the binary value `7F 45 4C 46` which is the magic

[16]`https://developers.google.com/android/nexus/drivers#hammerhead`
[17]`https://android.googlesource.com/platform/hardware/qcom/keymaster/+/master/keymaster_qcom.cpp`

```
$ objdump -p keymaster.b00

keymaster.b00:     file format elf32-littlearm

Program Header:
    NULL off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**0
         filesz 0x000000b4 memsz 0x00000000 flags --- 7000000
    NULL off    0x00001000 vaddr 0x0000a000 paddr 0x0000a000 align 2**12
         filesz 0x000019a8 memsz 0x00002000 flags --- 2200000
    LOAD off    0x00003000 vaddr 0x00000000 paddr 0x00000000 align 2**8
         filesz 0x00003f98 memsz 0x00003f98 flags r-x a0000000
    LOAD off    0x00007fcc vaddr 0x00004000 paddr 0x00004000 align 2**12
         filesz 0x000000b0 memsz 0x00005950 flags rw- 30000000
private flags = 5000002: [Version5 EABI] [has entry point]
```

Listing 3.4: Reading the ELF header

number for the *Executable and Linking Format* (ELF) [11]. This is a format for binary files and libraries. The header can be decoded with the `objdump` tool from the `binutils` package[18].

As we can see in in Listing 3.4, the `objdump` tool recognizes the ELF header and identifies four sections that are defined in the header. One section can be read and written to and one section is executable.

At an offset of 424 bytes I identified another interesting section. Byte 424 starts with `0x30`, This could indicate an ASN.1 encoded section. This is also suggested by strings further on in the section: For example the byte string `0C 4D 6F 74 6F 72 6F 6C 61 20 50 4B 49` that can be found in the section is exactly a string of 12 bytes (`Motorola PKI`) with the length before it (`0x0C`=12). Using an ASN.1 decoder I analyzed the sequence and found a certificate chain:

1. The first certificate is for `Motorola Attestation 280-1-110` and is signed by `Motorola Attestation CA 280-1`.
2. The second certificate is for `Motorola Attestation 280-1` and is signed by `Root CA 280`.
3. The third certificate is a Root CA certificate for `Root CA 280`. It is self-signed by `Root CA 280`.

On Android 4.4 in the directory `/data/misc/keystore/` a directory is created for each user (if not present). Android 4.4 has multi-user support on tablets to allow multiple users to share the same tablet. Phones do not have this functionality so only a directory called `user_0` is created. In this directory two files are created for each generated key pair by the `KeyMaster` running in Android OS that communicates with the trustlet in the TEE. To prevent directory traversal attacks special characters in the the key entry

[18]`http://www.gnu.org/software/binutils/`

```
android::String8 getKeyNameForUid(const android::String8& keyName, uid_t
    uid) {
     char encoded[encode_key_length(keyName) + 1];   // add 1 for null
     encode_key(encoded, keyName);
     return android::String8::format("%u_%s", uid, encoded);
}
```

Listing 3.5: Code used for naming key entry files

```
struct  qcom_km_key_blob {
  uint32_t magic_num;
  uint32_t version_num;
  uint8_t  modulus[KM_KEY_SIZE_MAX];
  uint32_t modulus_size;
  uint8_t  public_exponent[KM_KEY_SIZE_MAX];
  uint32_t public_exponent_size;
  uint8_t  iv[KM_IV_LENGTH];
  uint8_t  encrypted_private_exponent[KM_KEY_SIZE_MAX];
  uint32_t encrypted_private_exponent_size;
  uint8_t  hmac[KM_HMAC_LENGTH];
};
typedef struct  qcom_km_key_blob qcom_km_key_blob_t;
```

Listing 3.6: Storage format used for data from the Qualcomm trustlet

aliases are encoded. The naming of the files can be found in the Android source code in the `AndroidKeyStore` implementation [19] as shown in Listing 3.5.

The actual storage format of the `PKEY` is defined by the Qualcomm API [20] as shown in Listing 3.6.

As we can see the file contains a number of fields according to the struct:

- `magic_num`: A fixed magic number is used to identify the files as key storage. The magic number is `0x4B4D4B42` or `"KMKB"` (KeyMaster Key Blob).
- `version_num`: The version number indicates the version of `key`. Currently the version is 2 in Android 4.4.3. In Android 4.2.2 the version was 1.
- `modulus`: Stores the modulus of the RSA key pair. The size of this field is fixed to `KM_KEY_SIZE_MAX` which is 512 bytes (= 4096 bits) for Android 4.4 and 256 bytes for Android 4.2.2.
- `modulus_size`: The actual size of the modulus. Since the modulus can be smaller than 4096 bits this field indicates the number of bytes that is actual used.

[19]https://android.googlesource.com/platform/system/security/+/master/keystore/keystore.cpp
[20]https://android.googlesource.com/platform/hardware/qcom/keymaster/+/master/keymaster_qcom.h

- `public_exponent`: Stores the public exponent of the RSA key pair. The size of this field is also fixed to `KM_KEY_SIZE_MAX`.
- `public_exponent_size`: The actual size of the public exponent.
- `iv`: The IV (initialization vector) that is used for the encryption of the `encrypted_private_exponent` using AES-CBC-128.
- `encrypted_private_exponent`: Stores the private exponent of the RSA key pair in a encrypted form using AES-CBC-128 encryption. For the encryption the `iv` and an unknown key is used. This key is probably fixed in hardware to bind the private key to the device.
- `encrypted_private_exponent_size`: Stores the actual size of the private exponent in unencrypted form.
- `hmac`: A HMAC is computed over the whole file using a SHA-2 and a key which is probably also fixed in hardware. This should ensure the integrity of the file.

The `KeyStore` service adds some data around the struct to identify the contents. The key used for encryption of the private exponent and for computing the HMAC can not be found on the device. Also of the trustlet is not open source. According to the source code that is available the encryption is done in the TEE by the `keymaster` trustlet. We suspect that the keys are also stored in the TEE and are device specific. No method to access these keys from outside the TEE is documented, nor could we find one.

3.5 Key storage using AndroidKeyStore using the TEE on TI devices

3.5.1 Evaluation

The naming of the key entry files stored on devices that have a TI (Texas Instruments) processor and a `AndroidKeyStore` is exactly the same as the Qualcomm-based version discussed in section 3.4. The `AndroidKeyStore` on TI devices also adds two files in the `/data/misc/keystore/user_0` directory for each key pair:

- A `USRPKEY` file that stores the key pair parameters including the private key.
- A `USRCERT` file that stores the certificate.

However, the actual content of the private key file is different. The private key can not be stored in the private key file since the file size is smaller than the actual private key size used.

The `/data/misc/keystore/user_0` directory is not accessible by non-root user and other apps can not access the private key material of an application. So requirements $R_{app-bound}$

and $R_{device-bound}$ are guaranteed for attacker model $A_{outside-no-root}$. In section 3.4 an attacker having root permissions could rename or copy the files to new files in the same directory on the same device with the user-id of a second malicious application. The private keys would then directly be accessible by the other application. This attack also works on devices using a chip by Texas Instruments. The application that was assigned the key-files of another application could successfully generate a valid signature using the private key that was generated by the other application. Again this was exploited by renaming the files from user id of the first `KeyStorageTest` application to the user id of the second `KeyStorageTest` application.

So, like in section 3.4, requirement $R_{app-bound}$ is not guaranteed for attacker model $A_{outside-root-no-memory}$. Copying the files to another device does not work, as discussed in section 3.5.2. Like the Qualcomm solution the private key material is probably encrypted using a device specific key that is stored in the TEE. So requirement $R_{app-bound}$ is guaranteed for all attacker-models.

The `AndroidKeyStore` does not support the encryption of the key store (or entries within it) using a provided password. Nor can the user be asked to confirm that a certain private key can be used. As a result no user-consent can be required to use a private key. An attacker can access the private key material at any time. So requirement is not met for attacker-model $A_{outside-no-root}$.

This brings us to the conclusion that while the actual location for key pair parameters is different in Texas Instruments devices compared to Qualcomm devices (more about this in section 3.5.2) the results are the same:

- An attacker that has root credentials can use the keys in other applications on the same device by renaming the key-entry files. So requirement $R_{app-bound}$ is not satisfied for attacker models $A_{outside-root-no-memory}$ and $A_{outside-root-memory}$.
- Key pairs can not be used outside of the device because the private data is encrypted with a device-specific key stored in the TEE that can not be retrieved. So requirement $R_{device-bound}$ is satisfied for all attacker models.
- There is no way to require any input from the user for operations using the private keys. So requirement $R_{user-consent}$ is not satisfied.

3.5.2 Background

The Texas Instruments implementation of the `AndroidKeyStore` uses the same interfaces as the Qualcomm implementation that is described in section 3.4.2 on page 31. For completeness the complete implementation is also described here. To communicate with the actual trustlet that is running in TEE a driver is created that is called from the *KeyStore* service, this is defined in `keymaster_tuna.cpp`[21]. The low-level driver that is

[21] `https://android.googlesource.com/device/samsung/tuna/+/master/keymaster/keymaster_tuna.cpp`

used to actually communicate messages with the TEE on TI chips is also open-source and available on the AOSP git repositories [22]. The whole architecture is illustrated in figure 3.3 on page 37. In this figure two applications *App 1* and *App 2* are shown. They communicate with the `AndroidKeyStore` (*KS* in the figure) which in turn communicates via the shared memory with the `keymaster` (*KM* in the figure) application running in the secure world. This is exactly the same as the implementation on Qualcomm-devices.

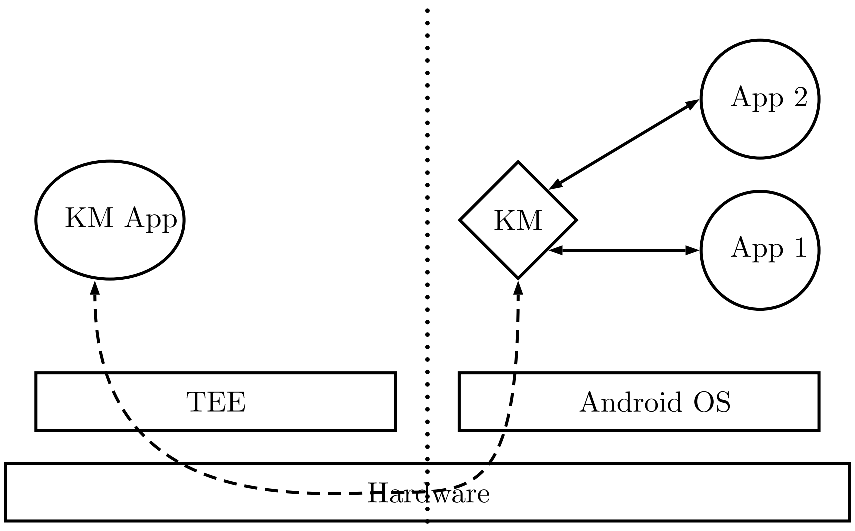

Figure 3.3: Architecture of AndroidKeyStore on TI devices

An analysis of the key store by Nikolay Elenkov shows that the actual private key data is stored encrypted in `/data/smc/user.bin` [20]. However, the actual format is unknown [20]. When I generated a new key pair using the `KeyStorageTest` application indeed the data stored in the `/data/smc/user.bin` file changed. The `/data/smc/user.bin` file appears to have the following format:

- 2048 bytes that contain `0x00` except for two 4 byte words that change after key generation. For the first word at offset 0 only values `0xEF00` and `0x0EF03` are observed. For the second word at offset 1024 only values `0x3F00` and `0xBF01` are observed. These words may indicate some (persistent) state.
- 2048 bytes of (presumably) encrypted data related to the `AndroidKeyStore`: These bytes change completely after every key pair generation and appear to have no structure. This indicates that some form of encryption is probably used.
- $n \times 2048$ bytes. Where n is the number of key pairs that are stored in the `AndroidKeyStore`. These bytes appear to store the key-entry data in a inde-

[22]`https://android.googlesource.com/platform/hardware/ti/omap4xxx/+/master/security/`

```
$ openssl asn1parse -inform der -in 10093_USRPKEY_TestKeyPair2 -offset
   326
    0:d=0  hl=4 l=1189 cons: SEQUENCE
    4:d=1  hl=2 l=   1 prim: INTEGER           :00
    7:d=1  hl=4 l= 257 prim: INTEGER           :E408...5ED7
  268:d=1  hl=2 l=   3 prim: INTEGER           :010001
  273:d=1  hl=4 l= 257 prim: INTEGER           :BA4B...27F1
  534:d=1  hl=3 l= 129 prim: INTEGER           :FD3E...F73F
  666:d=1  hl=3 l= 129 prim: INTEGER           :E683...8A69
  798:d=1  hl=3 l= 128 prim: INTEGER           :078C...273D
  929:d=1  hl=3 l= 129 prim: INTEGER           :C891...FE01
 1061:d=1  hl=3 l= 129 prim: INTEGER           :CA12...C6FA
 1193:d=0  hl=2 l=   0 prim: EOC
```

Listing 3.7: Parsing private data from the key entry file

pendently encrypted form: If a new key pair is generated in the `AndroidKeyStore` the 2048 bytes for key entries already present stay the same.

- 17 kilobytes (17408 bytes) that are always `0xA5`. The purpose of these bytes is unknown. They may be some kind of padding. However, the total file-size of `/data/smc/user.bin` increases with 2048 bytes for every key pair.

3.6 Key storage using AndroidKeystore using a software-based keymaster

3.6.1 Evaluation

The naming of the key entry files when a software-based keymaster is used is the same as we have seen in the Qualcomm-based key storage in section 3.4: The key entry files include the user id of the application. Again, if an attacker gains root permissions he can rename or copy the key entry files to include the user id of another application and use the keys in that application. This issue appears to be specific to the `AndroidKeyStore` and not to the actual implementation of key storage that the `AndroidKeyStore` uses. So criteria $R_{app-bound}$ is not met for attacker model $A_{outside-root-no-memory}$.

When a device does not require a PIN to unlock the device no encryption of the private key file is used: By parsing the `USRPKEY`-file an attacker can learn all information that should be kept secret such as the private exponent and the two primes of the RSA key pair as shown in Listing 3.7. Here we show how an attacker can learn the private exponent from a file. The fourth integer at offset 273 is the private exponent of the RSA key pair that was generated.

In this overview we see the following data (encoded as hexadecimal arrays):

1. The version number (=0)
2. The (public) modulus n (starts with `E408`)
3. The public exponent e (=`010001`)
4. The private exponent d (starts with `BA4B`)
5. The first prime p (starts with `FD3E`). This is one of the prime factors of the modulus.
6. The second prime q (starts with `E683`). This is the other prime factor of the modulus.
7. $d \mod p-1$ (starts with `078C`)
8. $d \mod q-1$ (starts with `C891`)
9. $q^{-1} \mod p$ (starts with `CA12`)

Items 2-4 are represent all the data that is stored for an RSA key pair (including the private key). The additional data stored in parameters 5-9 are used for the Chinese Remainder Theorem [16]. This allows for faster RSA operations compared to a native implementation. Even when this data to use the Chinese Remainder Theorem is not present the private key can be used as described in the OpenSSL documentation for the RSA algorithm[23].

An attacker that does not have root permissions (attacker model $A_{outside-no-root}$) can not access the key store directory and therefore can not rename or copy key entry files. So therefore requirements $R_{device-bound}$ and $R_{app-bound}$ are satisfied for attacker model $A_{outside-no-root}$. An attacker that has root permission (attacker models $A_{outside-root-no-memory}$ and $A_{outside-root-memory}$) can simply read the `PKEY` file to learn all private key information as shown above. Subsequently there is nothing limiting the attacker in copying this data of the device. Therefore requirement $R_{device-bound}$ and $R_{app-bound}$ are not met for attacker-model $A_{outside-no-root}$ if no PIN is used to lock the device.

However, when a PIN is required to unlock the device a random 128-bit AES master key is used for encryption. This master key is randomly generated and stored in the `.masterkey` file in the `/data/misc/keystore/user_0` directory. This file is encrypted using a key that is derived from the PIN using 8192 rounds of `PKCS5_PBKDF2_HMAC_SHA1`. The master key is used to encrypt all key entries without any form of per-entry key-derivation.

Even when requiring a PIN to unlock the device, an attacker that has root permissions (attacker model $A_{outside-root-no-memory}$) can assign private keys to other applications on the same device by renaming the (possibly encrypted) files as shown in section 3.4 and discussed above. However, an attacker using attacker-model $A_{outside-root-no-memory}$ has to decrypt the private key entries to be able to use them outside of the device. To do

[23]`http://www.openssl.org/docs/crypto/rsa.html`

this he has to learn the PIN (or password) used to unlock the device. While research shows that this may not be impossible it does require brute-forcing [6]. Therefore we conclude that this is only possible by learning the password from memory which only an attacker according to attacker-model $A_{outside-root-memory}$ can do.

Again like we saw for the Qualcomm and TI based solutions in section 3.4 and section 3.5, the `AndroidKeyStore` itself, regardless of the underlying storage solution, offers no way to require user consent. So even an attacker without root permissions can use the private key to do cryptographic operations without user permission.

This brings us to the following conclusion that if no password is used:

- An attacker that has root permissions (attacker-model $A_{outside-root-no-memory}$) can learn the private key data and use it on another device. Without root an attacker can not access the entry files. So requirement $R_{app-bound}$ and $R_{device-bound}$ are only satisfied for attacker model $A_{outside-no-root}$.
- The `AndroidKeyStore` offers no method to require user-consent so requirement $R_{user-consent}$ is not satisfied.

However, if a password or PIN is required to unlock the device:

- An attacker that has root permissions but can not inspect the memory of the device (attacker-model $A_{outside-root-no-memory}$) can rename or copy the key entry files such that another application on the same device can use them. So the requirement $R_{app-bound}$ is not satisfied for attacker model $A_{outside-root-no-memory}$ and up. To use the keys outside of the device the attacker has to decrypt the files for which the password or PIN is required. Research shows that this may be possible by brute-forcing based on some stored files which attacker model $A_{outside-root-no-memory}$ could do. This may be hard for good passwords with sufficient entropy. However, the password can only be learned from memory by an attacker using attacker model $A_{outside-root-memory}$. Therefore we consider the requirement $R_{device-bound}$ as violated by attacker using attacker model $A_{outside-root-memory}$.
- There is no difference for requirement $R_{user-consent}$ between requiring and not requiring a password to unlock the device. The requirement $R_{user-consent}$ is violated.

3.6.2 Background

The software-based key store is called `SoftKeyMaster` or `OpenSSLKeyMaster` and can be found in the Android repositories[24]. The communication between the application and the software-based is exactly the same as for the TEE-based implementation, on for example Qualcomm hardware, as described in section 3.4 on page 30. The `SoftKeyMaster` acts as a kind of driver between the `KeyMaster` daemon and the filesystem. This is shown

[24]`https://android.googlesource.com/platform/system/security/+/master/softkeymaster/keymaster_openssl.cpp`

in figure 3.4 on page 41. In this figure we see two applications (App 1 and App 2) that communicate with the `KeyMaster` (KM) that in turns communicates with the Android OS to store the key entries. There is no involvement from the TEE. A TEE may and the TrustZone technology needed to create a TEE may very likely not even be available on the device.

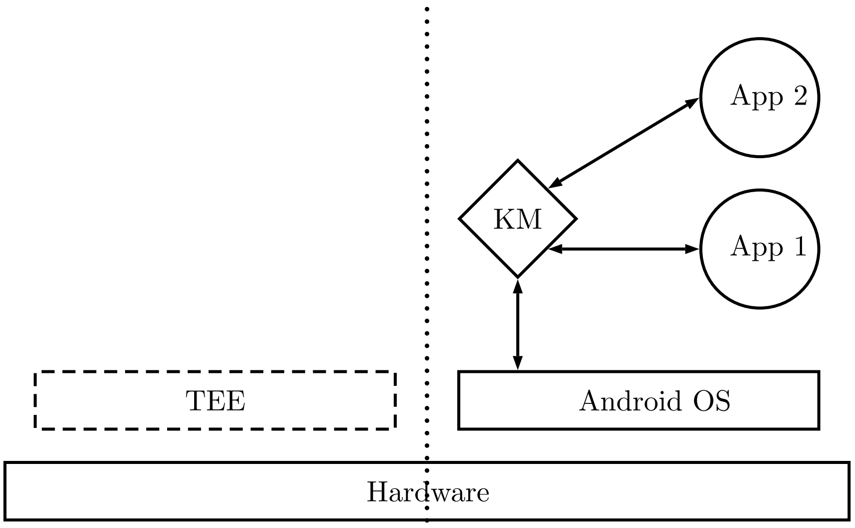

Figure 3.4: Architecture of the software based AndroidKeyStore

As discussed in the previous section, the master-key that is stored in the `.masterkey` file is used to decrypt files that store key-data. These files are defined by the `blob` struct that can be found in the key store code[25] as show in Listing 3.8 on page 42. The last field of the struct, the `value` field is not stored in the file and only used in memory to store unencrypted values.

The blobs are encrypted using AES-128-CTR and MD5 in the cipher-text is used to check the integrity. As shown in the `KeyStore.cpp`[26] file the developers themselves note that this should be replaced with a better format as shown in Listing 3.9 on page 42. Even in the newer format the developers make no note of including the application that is associated with the key in the stored file. It appears that it is still only present in the filename.

[25] `https://android.googlesource.com/platform/system/security/+/master/keystore/keystore.cpp`

[26] `https://android.googlesource.com/platform/system/security/+/master/keystore/keystore.cpp`

```
struct __attribute__((packed)) blob {
    uint8_t version;
    uint8_t type;
    uint8_t flags;
    uint8_t info;
    uint8_t vector[AES_BLOCK_SIZE];
    uint8_t encrypted[0]; // Marks offset to encrypted data.
    uint8_t digest[MD5_DIGEST_LENGTH];
    uint8_t digested[0]; // Marks offset to digested data.
    int32_t length; // in network byte order when encrypted
    uint8_t value[VALUE_SIZE + AES_BLOCK_SIZE];
};
```

Listing 3.8: The blob struct used by the AndroidKeyStore

```
 * Currently this is the construction:
 *   metadata || Enc(MD5(data) || data)
 *
 * This should be the construction used for encrypting if re-implementing
   :
 *
 *   Derive independent keys for encryption and MAC:
 *     Kenc = AES_encrypt(masterKey, "Encrypt")
 *     Kmac = AES_encrypt(masterKey, "MAC")
 *
 *   Store this:
 *     metadata || AES_CTR_encrypt(Kenc, rand_IV, data)  ||
 *             HMAC(Kmac, metadata || Enc(data))
```

Listing 3.9: Proposal to improve the storage format for the AndroidKeyStore

3.7 Key storage on the secure element

3.7.1 Evaluation

Android 4.1 and up also supports communicating with a Secure Element. However, the APIs allowing this are not available for normal applications. The Secure Element can be either a UUIC (SIM-card) or be embedded into the NFC controller as discussed in 2.2.3 on page 14. No open-source application for the Secure Element is available to provide a `KeyMaster`-like interface. An application that works like the `KeyMaster` trustlet in a TEE has exactly the same properties as a Secure Element-based variant: The only notable difference at the moment is the communication channel: The TEE-based trustlet communicates with Android OS over shared memory. A Secure Element-based solution uses a serial interface to communicate. Of course the hardware supporting the solution is different but this is transparent to the system. It should even be possible to develop a driver that implements the `IKeystoreService` interface and uses the a Secure Element for the actual cryptographic operations. This is depicted in figure 3.5 on page 44. In this picture *SE* represents the Secure Element where an application *SE App* is running. The `AndroidKeyStore` communicates with this application.

One thing to note is that it may be actually hard to implement an application that binds the Secure Element to the actual device if the Secure Element is not embedded in the phone. For example a SIM can be removed and installed in another phone, so the keys are bound to the SIM instead of the device. This may not be a problem for most use cases but it should be considered when evaluating risks.

To make the comparison fair we evaluate a Secure Element solution the same as the TEE-based solution. For both platforms better solutions could be developed that guarantee additional requirements such as, for example, user consent. The differences between the two solutions are discussed in section 2.2.3 on page 14. However, these differences do not reflect in any differences in the security criteria as described in section 3.2.2 in comparison to the TEE-based solution. Since no implementation is available to evaluate, no concrete evaluation is provided.

3.7.2 Background

For an application to communicate with a Secure Element the application has to have a special permission that is controlled by the `/etc/nfcee_access.xml` file in Android OS. By default only the Google Wallet is allowed:

```
<?xml version="1.0" encoding="utf-8"?>
<resources xmlns:xliff="urn:oasis:names:tc:xliff:document:1.2">
  <!-- Applications granted NFCEE access on user builds
  See packages/apps/Nfc/etc/sample_nfcee_access.xml for full documentation.
```

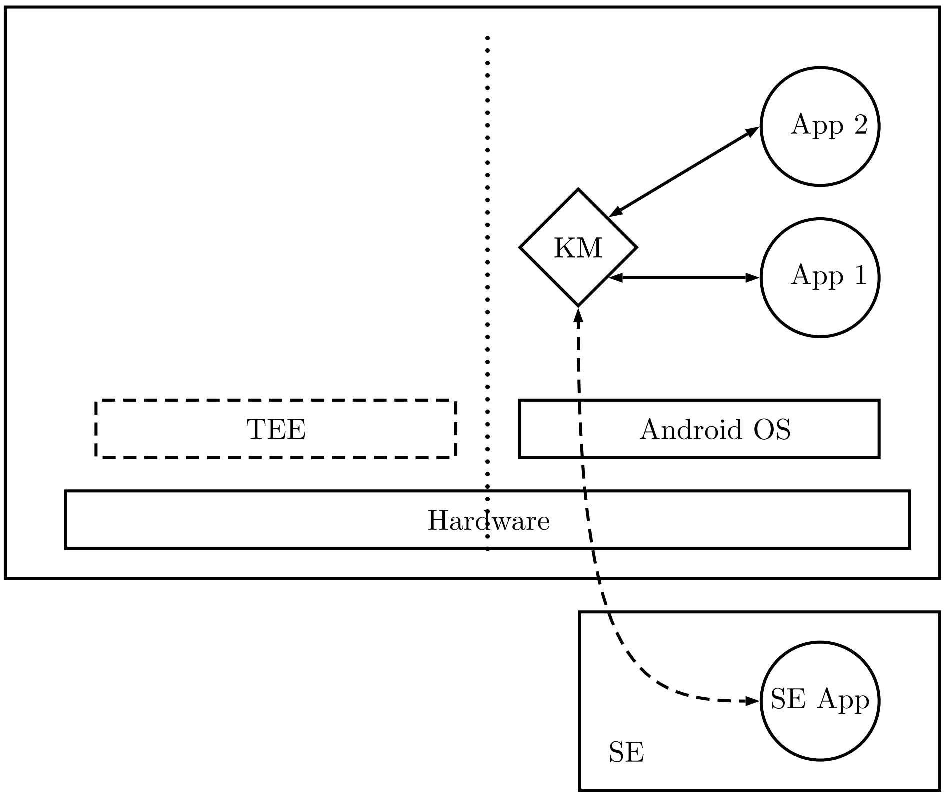

Figure 3.5: Possible architecture of AndroidKeyStore using a Secure Element

```
   -->
  <!--  Google wallet release signature -->
  <signer android:signature="3082...852e" />
</resources>
```

Only a user having root permissions can write to this file. The permissions to access the Secure Element can not be requested by an application like normal permissions. They have to be added by the device-manufacturer in the device image.

An application that does have the right permissions can communicate with the Secure Element using a interface that resembles smart-card communication as shown in Listing 3.10. The transcieve parameter should be a byte-array resembling a command APDU according to ISO/IEC 7816-4 and the response is a response APDU. These APDU's are also used for normal smart-cards. The actual content of the commands is defined by actual software running on the smart-card. The SEEK (Secure Element Evaluation Kit) project[27] provides a more programmer friendly API for the Secure Element.

[27]https://code.google.com/p/seek-for-android/

```
NfcAdapterExtras adapterExtras = NfcAdapterExtras.get(NfcAdapter.
    getDefaultAdapter(context));
NfcExecutionEnvironment nfceEe = adapterExtras.
    getEmbeddedExecutionEnvironment();
nfcEe.open();
byte[] response = nfcEe.transceive(command);
nfcEe.close();
```

Listing 3.10: Communicating with the Secure Element

An application for the Secure Element can be developed that is similar to the `KeyMaster` application. The chips that house such Secure Elements are generally more tamper-resistant then ARM chips that house TrustZone technology. They are based on smart-cards that are used for years in payment-systems.

3.8 Conclusion

Solution	$A_{outside-no-root}$	$A_{outside-root-no-memory}$	$A_{outside-root-memory}$
Bouncy Castle with stored password (section 3.3)	✓	×	×
Bouncy Castle with user-provided password (section 3.3)	✓	✓ (if password has sufficient entropy)	×
AndroidKeyStore using the TEE on Qualcomm devices (section 3.4)	✓	✓	✓
AndroidKeyStore using the TEE on TI devices (section 3.5)	✓	✓	✓
AndroidKeyStore using a software-based keymaster without a PIN to unlock the device (section 3.6)	✓	×	×
AndroidKeyStore using a software-based keymaster with a PIN to unlock the device (section 3.6)	✓	✓	×

Table 3.2: Overview of the evaluation results for requirement $R_{device-bound}$

An overview of the results is provided in table 3.2, table 3.3 and table 3.4. Note that the Secure Element (as discussed in Section 3.7) is not included in these tables as no key store implementation is available. However, we can imagine an implementation that has the same security properties as the `AndroidKeyStore` on devices with TrustZone-support. As shown the `AndroidKeyStore` using the TEE is the only solution that provides a key store where keys can not extracted from the device ($R_{device-bound}$). However, as

Solution	$A_{outside-no-root}$	$A_{outside-root-no-memory}$	$A_{outside-root-memory}$
Bouncy Castle with stored password (section 3.3)	✓	×	×
Bouncy Castle with user-provided password (section 3.3)	✓	✓ (if password has sufficient entropy)	×
AndroidKeyStore using the TEE on Qualcomm devices (section 3.4)	✓	×	×
AndroidKeyStore using the TEE on TI devices (section 3.5)	✓	×	×
AndroidKeyStore using a software-based keymaster without a PIN to unlock the device (section 3.6)	✓	×	×
AndroidKeyStore using a software-based keymaster with a PIN to unlock the device (section 3.6)	✓	×	×

Table 3.3: Overview of the evaluation results for requirement $R_{app-bound}$

table 3.3 shows, the integrity of the allocation of keys to a certain application within the device is not guaranteed. The result is that any implementation that uses the `AndroidKeyStore` is vulnerable to attacks where an attacker that gains root permissions (attacker-model $A_{outside-root-no-memory}$) can assign any key stored in the key store to his app by simply making copies of the entry files that include another user id in the filename. So while attackers that gain root credentials can not copy the key pair of the device to use it elsewhere they can make an application that provides an *oracle*: A service that the attacker can query for the encrypted/signed data of attacker-defined plain text.

The software-based `AndroidKeyStore` on devices that do not require a PIN to unlock the device does not apply any form of encryption. The key store entries are protected by the fact that a normal application can not read the key store due to the file permissions. This of course is a very weak solution that requires minimal effort by an attacker to learn the key pair data. Any exploit that allows an attacker to gain root permissions directly exposes the private keys. While encrypting the data using a password that is also stored on the device provides no increase in security guarantees it does require more effort from the attacker to learn the private data.

The Bouncy Castle key store that uses a user provided password provides a better solution security-wise for all requirements except $R_{device-bound}$. This is the result of requiring something to use the key store that is not stored on the device (the user provided password). Most banking application already require a PIN to use the application that could serve as the user-provided password. So this may not be a problem for those

Solution	$A_{inside-no-root}$
Bouncy Castle with stored password (section 3.3)	×
Bouncy Castle with user-provided password (section 3.3)	✓ (if password has sufficient entropy)
AndroidKeyStore using the TEE on Qualcomm devices (section 3.4)	×
AndroidKeyStore using the TEE on TI devices (section 3.5)	×
AndroidKeyStore using a software-based keymaster without a PIN to unlock the device (section 3.6)	×
AndroidKeyStore using a software-based keymaster with a PIN to unlock the device (section 3.6)	×

Table 3.4: Overview of the evaluation results for requirement $R_{user-consent}$

applications. However, not all applications may be able to require a (application specific) password due to usability constraints. Also, remembering a password for each application on the device may be hard for the common user. Furthermore a user-provided password may have a low entropy which allows it to brute-forced, effectively making the user-provided password solutions worse. Another problem with user-provided passwords is that they can be observed when the user is entering them (shoulder-surfing). While a user-provided password in theory provides a more secure solutions on almost all requirements the security in practice may be less.

There is no effective difference between the TEE-based `AndroidKeyStore` on Qualcomm devices and the TEE-based `AndroidKeyStore` on TI devices. Both have a difference in inner-workings (especially where the private key data is stored). However, both solutions still allow allocating a key pair to another application by renaming the files in the `keymaster` directory. For the Qualcomm solution you actually copy the (encrypted) private key data for TI you copy a reference to the private key data.

Maybe for a more fair comparison the key stores should be divided in two classes: those that require a user provided password and those that don not. If we consider a use case where user-consent is not a problem because we assume that the application is always honest (criteria $R_{user-consent}$ does not matter) and we do not want to require a user-provided password the TEE-based `AndroidKeyStore` is clearly the best solution. The best solution for the user provided password class is Bouncy Castle.

There may be little to none difference between the solution that the `AndroidKeyStore` provides where the keys can not be extracted from the device but can be used by any application on the device and the solution that Bouncy Castle provides where keys can be extracted. Most devices have a nearly constant connection to the internet so extracting a key from a device to be used elsewhere or querying the device to do the operations using the key is almost the same.

3.9 Recommendations

3.9.1 For developers intending to use key storage

Developers of applications that require secure key storage should, to make the right decision about what key store to use, first decide whether they can (and want) to:

A Ask the user for a password (that has sufficient entropy).

B Use the hardware-backed key storage.

Note that not all devices support a hardware-backed key store. Android versions before 4.3 do not have the `AndroidKeyStore`. On later Android versions the availability of a hardware-backed `AndroidKeystore` can be checked by using the static `isBoundKeyAlgorithm(String algorithm)` function of the `KeyChain` class. If this function returns `true` the `AndroidKeyStore` is hardware-backed.

If the user can not be asked for a password and hardware-backed key storage can be used the `AndroidKeyStore` provides the most secure solution. It seems impossible for an attacker to extract the keys stored in the key store on these devices. However, developers should be aware that the keys can be used by other application on the device if an application is able to run under root permissions. There are generally two methods for applications to run under root permissions:

1. As described in section 3.2.2 on page 22, many users root their phone to allow applications to work around the permissions they can get or to modify behavior of the Android OS that is not possible using the normal APIs. An attacker can create a malicious application that asks for root permissions for some valid reason but then also use the permissions to attack the key store.
2. Attackers may be able to gain root permissions by exploits in applications or libraries that run under root permissions in Android. A recent example of this is the futex privilege escalation [12] that is used by *Geohot* in the towelroot[28] proof-of-concept on Android.

To minimize the risk, users should be educated about the risks that occur when users *root their phone* (allow applications to run under root permissions). In extreme cases where a vulnerability is used on a large scale by attackers to gain root credentials developers should have methods to revoke the keys stored in the devices that are vulnerable.

If the user can not be asked for a password and no hardware-backed key storage can be used the `AndroidKeyStore` provides the most secure solution on devices that need to be unlocked using a PIN or password. On devices that use no PIN both the `AndroidKeyStore` does not encrypt the sensitive data at all. The file permissions prevent users that do

[28]`http://towelroot.com/`

not have root permissions from accessing these files. While the Bouncy Castle key store can use encryption the key still needs to be stored on the device. So a user having root permissions can also retrieve that.

If the user can be asked for a password an no hardware-backed key storage can be used the Bouncy Castle key store provides the most secure solution. However, the password should be of sufficient entropy to be able to withstand brute-force attacks from attackers. Also care should be taken to store the keystore-file in the application specific data directory. If the key stored in the device is only intended for secure communication, a password-authenticated key agreement scheme such as PACE [33] may provide a more secure solution as it is resistant to brute-force attacks.

If the user can be asked for a password, hardware-backed key storage is available and the protocol allows it, the best solution can be created by using a key pair that is stored in the `AndroidKeyStore` that authenticates the device and a key pair that is stored in a Bouncy Castle key store using a user provided password that authenticates the user. Again, a password-authenticated key agreement may provide a more secure solution.

3.9.2 For developers of the key stores

The security of the `AndroidKeyStore` can be improved by using a number of measures while retaining the same architecture:

- Encrypt the key store files for the software-based `AndroidKeyStore` when the device requires no PIN to unlock. A key could for example be derived from the device-id. While this provides no additional security properties it makes it harder for attackers to read a key: The attacker first has to derive the encryption key.
- Include the user-id of the application that generated the key pair actually in the integrity checked section of the keyblob. This again does not solve the whole problem: A attacker can change the user-id of the application that used the files. However, this is again harder than just renaming the files.
- Optionally allow encryption of the key entry file using a user-provided password. This provides the possibility to require user-consent to use a key pair.

Solving the problem that multiple applications on the same device can access the key pair is not trivial: As discussed, integrity-checking the user-id of the application in the key entry file does not solve the problem, the user-id of an application can be changed. Like we've seen in Section 2.2.1 on page 9, the TEE can read the memory of the Android OS. So a possible solution could be to check the signature or application-id of the application requesting key operations from the TEE. This may be very hard for an attacker to work around. However, no research has been done in this area, so research should be done to validate whether this is actually a viable and secure solution.

Another solution could be to ask the user to confirm the usage of a key pair by showing a dialog from the TEE on the screen of the device. This dialog should show which application requests which key and optionally what data it wants to sign (to provide a *what you see is what you sign solution*). If the touch input from the touchscreen can not be faked by an attacker, this solution is not susceptible to a remote attack over the internet or by an application running on Android OS. The usability, however, may be low if a large number of key operations is requested and each one has to be confirmed by the user. A security trade-off can be made to improve the usability by allowing an application to use a certain key pair for a predefined period (for example 10 minutes).

4 Secure computation

As discussed in section 2.2.1 on page 9 and as observed in section 3.4.2 on page 31 the ARM TrustZone environment uses signatures to check whether certain code is allowed to run in the secure world. This also means that, in order to run certain code on a device for testing, the code has to be signed by the hardware manufacturer. I contacted one of those parties (Motorola) to see if there were any possibilities to get an application signed for scientific purposes. Motorola did not respond to this request. The only other solution is to use a development platform that does not have its certificate fixed yet. However, these platforms are expensive. As a result only an analysis of the documentation and information available about secure computation will be discussed in this chapter. First a number of secure world OSes that provide a Trusted Execution Environment (TEE) to run in the secure world provided by ARM TrustZone technology are discussed in section 4.1. Next in section 4.2 we will discuss a number of use cases where secure computation is useful and how it, in these use cases, can be used.

4.1 Secure world operating systems

At the moment there are a number of solutions that provide a TEE for trustlets to run in. Most solutions provide both an operating system for the secure world and an API that applications running in the TEE should use. Note that applications running in the TEE are called *trustlets*. We call the software that runs in the secure world and provides a TEE *secure world OSes*. As we will see, some solutions are developed with a specific operating system running in the normal world in mind others support all operating systems. In this section we discuss a number of secure world OSes that are described either in literature or are open source:

- **SierraTEE**[1] A open source solution provided by SierraWare.
- **Genode**[2] A new open source operating system that is not purposely developed for the TrustZone. However, its aim for a small TCB (Trusted Computing Base) makes it suitable for use in the secure world of the TrustZone.
- **TLR** [50] A scientific solution based on .NET by Microsoft Research and the University of Lisbon.

[1]http://www.sierraware.com/open source-ARM-TrustZone.html
[2]http://genode.org/documentation/articles/trustzone

All three solutions use the same architecture that is pictured in figure 4.1 on page 52. The secure world operating system is running in the secure world and a normal operating system such as Android OS is running in the normal world. On both operating systems applications are running. The applications running on the secure world OS are called *trustlets* and the applications running on the normal OS are called *apps*.

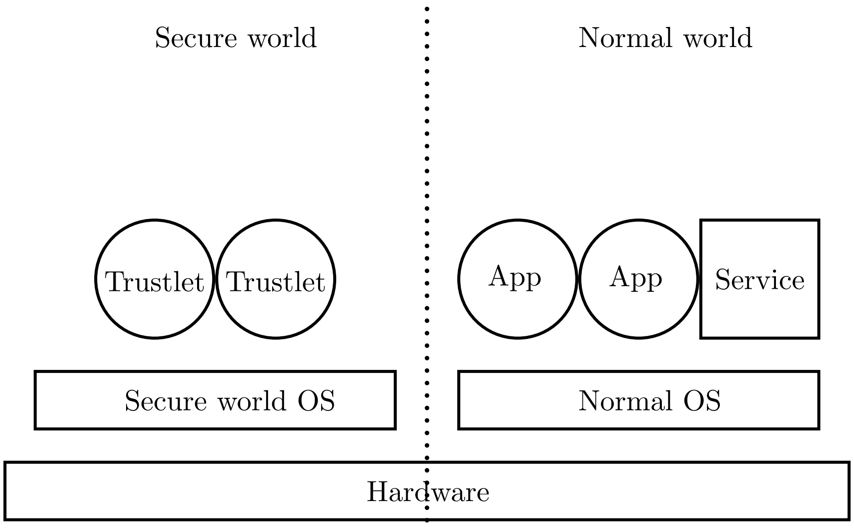

Figure 4.1: The separation of the hardware by TrustZone in two worlds.

4.1.1 SierraTEE

SierraTEE is a open source secure world operating system specifically designed by Sierraware to run in the secure world of a TrustZone enabled processor. It has support for different operating systems such as Linux, Android and BSD [53]. Recently, Sierraware announced a version for 64-bit ARM processors[3]. Particularly interesting in this paper is the Android integration that Sierraware shows. One of the example applications that is provided, is a banking application that communicates with a trustlet running in the TEE that shows a PIN dialog to confirm transactions.

According to two presentations[4] from Sieraware the SieraaTEE provides:

- Integrity management, a scanner that checks the integrity of:
  - The file-system of Android OS.
  - The processes running under Android OS.

[3]`http://www.embedded.com/electronics-products/electronic-product-reviews/operating-systems/4428035/Sierraware-hypervisor-targets-64-bit-ARM-Architectures-`

[4]`http://www.slideshare.net/sgopu/se-android-securityintegritymanagement` and `http://www.sierraware.com/sierraware_tee_hypervisor_overview.pdf`

  - The interrupt-table.
  - The Android OS kernel.
- Key management, secure storage of keys.
- MDM (Mobile Device Management) and MAM (Mobile Application Management), controls the settings of the device and what applications can run on the device.
- DRM (Digital Rights Management), allows for copyright protection of multimedia.
- Secure display and input, allows developers to display content, ask for user-input, and do cryptographic operations based on that input in the TEE. The result of such operation can then be transferred to Android OS. The user communicates directly with the TEE without any interference from Android OS.

4.1.2 Genode

The TU Dresden OS group introduced Genode in 2012 [25]. It is an operating system created with a low complexity of only 10.000 lines of code [1]. This low complexity makes it easier to verify the security of the operating system. This in turn makes it a good basis for a secure world OS because of its small TCB (*Trusted Computing Base*). In 2014 an implementation of Genode running in the secure world of a TrustZone-enabled ARM processor was presented [32]. The whole TCB was only 20.000 lines of code. The implementation supports running along side Android. In a video[5] the author of the Genode implementation on TrustZone shows both Android and Genode running on a Freescale development platform. Both operating systems are able to show content on the display.

In an article[6] on the Genode website more details are given about the implementation. The architecture of the implementation is shown in figure 4.2 on page 54. Genode is running in the secure world, the box in the upper-right corner of the figure. What is particularly interesting to see is that both the touchscreen driver interfacing with the touchscreen hardware and the framebuffer driver interfacing with the display are run in the secure world. This provides a trusted path which prevents attackers that exploited the normal world operating system to learn what a trustlet running in the secure world displays on the screen and what the user enters.

As shown in figure 4.2 the Android operating system running in the normal world provides the secure world with (a pointer to) a display-buffer. This display-buffer is then provided to the display by Genode running in the secure world. An application running in the secure world can also display content on the screen. However, a small part of the display is always controlled by Genode. In this small part the source of the content shown on the rest of the display is presented to the user. This part of the screen can not be controlled

[5] `https://www.youtube.com/watch?v=voFV1W4yyY8`
[6] `http://genode.org/documentation/articles/trustzone`

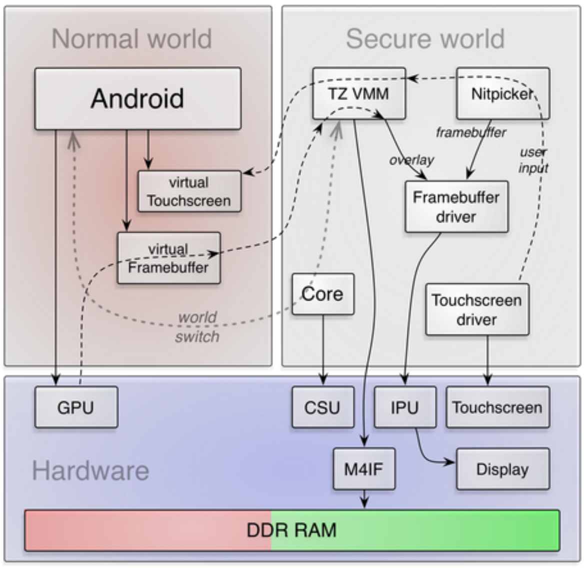

Figure 4.2: Architecture of the TrustZone implementation of Genode. Source: `http://genode.org/documentation/articles/trustzone`

from the normal world. This prevents applications running in Android OS from tricking the user into believing that they are in fact looking at a secure world application (and possibly entering sensitive information such as a PIN). So this feature provides local attestation as discussed in section 2.2 on page 8.

4.1.3 TLR

Microsoft also created an architecture for running applications in a secure environment using TrustZone. To do this they created the *Trusted Language Runtime* (TLR) which is based on the .NET MicroFramework runtime [50]. The Trusted Language Runtime supports multiple *trustboxes* in the secure world OS. This are isolated containers that house a trustlet as shown in figure 4.3 on page 55.

TLR does not provide remote attestation itself. The authors note that this could be achieved, however, it is omitted because the authors note that devices are usually initialized by the device manufacturer in an environment that could be secured. The device manufacturer can generate a (non-exportable) key pair in a trustlet in the TEE during initialization in the factory and sign the public key of this key pair to certify its authenticity. The devices are then shipped from the factory with a trustlet that contains a key pair and a certificate. This trustlet can be used to provide remote attestation. The result is that TLR does not have to provide remote attestation, it is provided by a trustlet which is initialized by the device manufacturer in the secure environment of the factory [49].

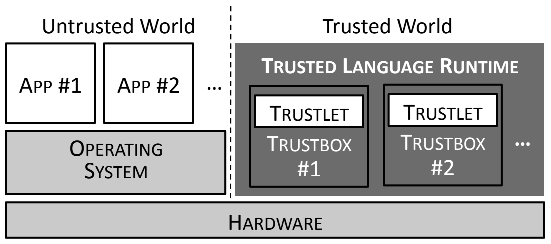

Figure 4.3: Architecture of TLR. Source: [49]

In contrast to SieraTEE and Genode, TLR provides no possibilities to communicate directly with the the local user via local devices such as the display and the touchscreen. Communication with local devices is handled by the OS running in the normal world. This OS can be exploited and communication from TLR via the normal world to the devices can be intercepted and modified. This means that local attestation is not possible and getting user-input in a secure way is impossible. Both showing output to the user and receiving input from the user are handled by the operating system running in the normal world which could be compromised.

4.1.4 Other platforms

Another alternative solution to provide secure computation is use of the Secure Element (SE). A comparison between the Secure Element (SE) and a TrustZone-based TEE is already made in chapter 3 in section 14 on page 14.

4.2 Use cases

This section provides a number examples how the security of certain use cases can be enhanced by using secure computation on mobile devices. While we focus here on Android devices, the underlying hardware on other mobile platforms is very comparable: Both Apple iPhones (and iPads), Windows Phone devices and Android devices use ARM processors. So the examples can possibly very easily be ported to other mobile operating systems. We assume that a secure world OS provides a TEE which is comparable to Genode in section 4.1.2 and that this secure world OS is running in the secure world to provide secure computation. What is particularly interesting about this solution, as discussed earlier, is that it provides local attestation and that can communicate directly with the user by interfacing with touchscreen driver from the secure world. We also assume that trustlets can be installed in the TEE by uploading a file to the environment that is signed by a entity of which the public key is stored securely in the environment. A key pair is stored in the TEE to allow authentication of the device. We call this key pair the *device key pair*. This key pair can not be extracted. A certificate confirming the authenticity of the public key is signed by the hardware manufacturer during initialization. In this section we call the application running in the normal world under Android OS the *app* or *mobile application* and the application running in the TEE in a container that protects the confidentiality and integrity of the data within the *trustlet*.

Additionally we discuss two existing uses of secure computation on mobile platforms using TrustZone technology. The first is the Samsung Knox 2 system designed by Samsung. This system adds security features for corporate use to Android OS. The second is DRM (Digital Rights Management) which is available on most Android platforms.

4.2.1 Mobile banking

The TEE can be used to secure mobile banking on mobile devices. In a mobile banking app the users of a bank can view their transaction history and submit new transactions. After installation the user has to register his device at the bank. This is usually done on the banks website or from the application using some form of an one-time-password or signature that is generated on a secure device. After registration the banking application is protected by a PIN of 4 to 6 digits that the user has to enter to unlock the application and to confirm transactions. Some banks require additional confirmation for transactions depending on the risk profile of the transaction. A mobile banking application generally has a number of requirements that need to be achieved in order to secure the app:

- Only the device that was registered and only the user of the device should be able to submit valid transactions.
- The application should provide local attestation such that the user knows he is entering his PIN in a authentic application.

These requirements can be achieved in a secure way by using the TEE. Beside the mobile banking app in Android OS a trustlet is installed in the TEE that handles sensitive tasks. The user, as usual, registers the application after installation with the bank. During registration a key pair is generated by the trustlet. The public key of this key pair should be associated with the user or the back account of the user by the bank in a secure way. The actual implementation of this may differ depending on the token that is used by the bank.

Communication with the bank is handled by the banking application running under Android OS. When the user wants to transfer money he enters the transaction details in a dialog in the application. This transaction details are then send to the trustlet to be confirmed by the user. The trustlet shows the transaction details and asks the user to confirm. The most simple way to confirm is by simply showing a *confirm* and *cancel* button. However, this setup may be prone to attacks where an attacker gains physical control over the phone and can create transactions. To make this physical attack harder the user could be asked for a PIN that he could choose himself during device registration. Of course an attacker may still be able to change, for example, the amount of the transaction in the data that is send to the trustlet from the application. A wrong amount or destination, however, will then be displayed by the trustlet. The user may not notice this and confirm the transaction anyway. To reduce this risk the user can be asked to enter for example the amount of the transaction for confirmation.

After confirmation the trustlet signs the transaction using the private key and provides it to the application to be send to the bank. The signed transaction is checked by the bank by using the registered public key and, if valid, executed. An attacker can change the signed transaction data. However, this will render the signature invalid. Note that care should be taken to prevent replay attacks. A possible approach can be to add a timestamp or a counter in the transaction data.

To provide local attestation the user should be able to check whether the transaction confirmation dialog is actually from the trustlet in the TEE or whether it is from some phishing app. This can be achieved by using a TEE that can indicate when the display is controlled by a certain trustlet and when the Android is controlling the display.

4.2.2 Mobile payments and ticketing

In the use case of mobile banking the genuine user is generally not considered as an attacker, he has generally no incentive to attack the application (although this may not be completely true if the bank has to refun the damages due to fraud). However, in mobile payment and ticketing use cases the user himself may profit from attacks on his mobile device. For example he could try to attack the system to get free access to transportation or to pay at a point-of-sale (PoS) without actually paying. Also in these use cases where the user of the phone itself is the attacker the TEE can provide a secure solution.

Using the device key pair stored in the TEE a mobile payment provider or ticketing provider can set up a secure channel with the TEE to send data and code to the TEE. This data can only be decrypted in the TEE using the device key pair that can not be extracted. For example a mobile payment provider could have an application that first asks the TEE for its certificate for the device key pair. The payment provider verifies that this is a valid signature signed by the device manufacturer. It then encrypts and sends the trustlet code and data needed to perform a payment using the public key of the TEE. Because only the TEE has the private key, only the TEE can decrypt this. The trustlet can now for example communicate over NFC with a payment terminal. This can be done either directly from the TEE if there is support for this or by using a secure tunnel that runs through Android OS.

The architecture between such solution is very comparable to a setup where the UUIC (SIM) is used as Secure Element. However, in contrast to a Secure Element-based solution the trustlet can securely ask for user-input such as a confirmation that a transaction should be executed or a PIN to ensure that the owner is present.

Note that, to ensure confidentiality, the integrity of the TEE should be ensured. Furthermore to provide a secure platform the confidentiality and integrity of all data and code stored within the TEE should be ensured, even between trustlets within the TEE. The payment provider has to put his trust in the TEE as he stores information such as keys in the TEE that should be kept secret. However, using secure boot as discussed in section 2.2.1 on page 10 the integrity of the TEE can be ensured. In such a secure setup the integrity and authenticity of the TEE code is protected by signatures that are checked upon booting. By validating the code of the secure world OS the payment provider can be ensured that the TEE has no security issues. This can be simplified by using a small TCB, so that less code has to be checked.

To simplify the adaption of the TEE for payment and ticketing providers a JavaCard-compatible runtime could be designed for the TEE. This allows the providers to push the code they already have for JavaCard based smart cards into the TEE without any modifications. By defining some additional interfaces for the runtime they can slowly adapt to an interactive solution by asking the user for input on the device. This user input can increase the security of such solution, for example entering a PIN on a mobile device running a TEE may be more secure than entering a PIN on the point-of-sale terminal.

4.2.3 Secure communication

A third use case is secure communication between users. For example a setup can be created where PGP [62] is used to secure communications between users. A trustlet is used to generate two key pairs for signing messages and receiving encrypted messages. Both key pairs are stored in a trustlet in the TEE. The most secure solution is a setup where, when the users receives a message, the trustlet decrypts the message using the private key stored in the TEE and shows the contents of the messages directly from the trustlet. When the user wants to send a message he enters it in the trustlet and the trustlet signs the message and encrypts it using the addressees public key. A PKI (Public Key Infrastructure) can be used to bind the public keys of parties to their identity. Another solution is to use a web-of-trust as used in PGP. In this web-of-trust parties meet (often face-to-face) to confirm their public keys. After confirming they sign a kind of certificate that confirms that they confirmed the public key of a party.

A comparable setup can also be used for voice and/or video communication. When calling a certain party, both parties provide a certificate that binds their public key to their identity. Using these certificates a secure connection is setup between the two parties that prevents eavesdropping. The certificates also ensure that the authenticity of each party is ensured. Next the trustlet directly accesses the speaker and microphone (and display and camera for video communication) of the device. This also prevents the Android OS from accessing these devices while the call is active.

4.2.4 DRM

As discussed in the introduction, recent Android versions have a trustlet to allows Android devices to play DRM-protected media. The trustlet runs on a proprietary secure world operating system created by Trustonic. According to a presentation from Trustonic [36] Android devices provide a complete solution which allows the whole video-path to be handled by a TrustZone application. This means that the encoded video packets are received by the Android OS and directly provided to the trustlet. Keys for decryption are either hard-coded using white-box cryptography or established. The trustlet then sequentially decrypts, decodes and renders the video in the secure world. Using this architecture the decrypted video can not be intercepted to be stored, it never leaves the secure world.

4.2.5 Samsung Knox

Samsung Knox is a set of features developed by Samsung to improve the security of mobile devices for corporate use. It allows users of corporate phones to use the phone both for personal and corporate use without endangering the security of data stored in the corporate applications. Knox does this by offering multiple environments to the user, which look like separate Android systems. Typically an environment is provided for personal use and an environment is provided for business use. Knox is available on the latest (non-budget) phones produced by Samsung, such as the Samsung Galaxy S4 and Samsung Galaxy S5. It can not be installed on other Android phones. There are two versions of Knox: Knox 1.0 and Knox 2.0. In this section we will focus on Samsung Knox 2.0. Samsung does not give much details about how it implemented its features. However, it provides one page[7] on their website with some technical details. We will analyze these here and relate them with secure computation on the TEE.

The first feature of Samsung Knox is trusted and secure boot. This is actually largely based on secure boot as discussed in section 2.2.1. All firmware on the device is signed and these signatures are verified during boot to ensure that the device is booting from a valid, genuine image. Trusted boot ensures that data stored in the TEE can not be extracted when the validity of the firmware is not verified or not valid.

SELinux is used in all Android versions since 4.3 to limit the damage that a malicious application running in Android can do [51]. However, according to the Samsung webpage, SELinux assumes that the integrity of the kernel is not compromised. To ensure this Samsung added TIMA (TrustZone-based Integrity Measurement Architecture). TIMA regularly checks the integrity of the Android kernel by inspecting the memory from the

[7]`https://www.samsungknox.com/en/solutions/knox/technical`

TEE and it verifies the integrity of kernel modules when they are loaded. According to Samsungs documentation TIMA also provides a key store protected by hardware-based security comparable to the `AndroidKeyStore` discussed in chapter 3 to store key pairs and certificates. Which secure world operating system is used by Samsung is unclear.

According to Samsung's documentation the data stored in the different Android environments offered by Knox (called containers) are individually encrypted using an AES-256 key. The encryption key is derived from a user-provided password using PBKDF2 [31]. Samsung does not provide any information that could indicate that the TrustZone-technology such as their TIMA is used for this purpose.

There is not much (public) research available about the security of Samsung Knox. Researchers from Ben-Gurion University Cyber Security Labs reported on their website that they found an vulnerability in Knox that allowed for a man-in-the-middle attack [8]. However, they provide no exact details and the Samsung denied the vulnerability in a official response [48].

4.3 Conclusion

Using currently available open source software such as SierraTEE in section 4.1.1 or Genode in section 4.1.2 and the TrustZone features available on many mobile platforms a TEE can be build to provide secure computation. Microsoft Research describes a framework and architecture for secure computation using TLR as discussed in section 4.1.3. An (open source) implementation of this framework is not provided.

Mobile payments and ticketing, mobile banking and secure communication can all benefit security-wise from secure computation as shown in the use cases. The possibilities of the TEE provide developers with better ways to secure applications. For mobile payments and ticketing the TEE can replace the smart cards that are currently used. This process can be simplified by providing a JavaCard implementation for the TEE. This would allow providers to run their implementations they designed for JavaCard smart cards in the TEE without any modifications. By adding the possibilities of the TEE to communicate with the user via secure input and output the security of existing solutions can be improved.

The current features implemented by Samsung as part of their Knox solution are also interesting. Especially the TIMA application running in the TEE that verifies the integrity of the Android OS kernel and kernel modules could provide some additionally security to Android systems to prevent exploits from causing any damage. This is also provided by Sierraware in their SierraTEE as discussed in section 4.1.1.

5 Conclusion

In Chapter 3 different key store solutions available on different versions of Android OS and different hardware are analyzed. The Bouncy Castle is key store is protected by software-based security. It is available on all Android devices and all versions of Android OS. The key store does not depend on any hardware features. The `AndroidKeyStore` is only available on devices running Android 4.3 or later. There are two versions of the `AndroidKeyStore`, a version that is secured by hardware-based security features by using a trustlet in the TEE, and a version that is only protected by using software-backed security. The software-backed version is only available as fallback if no hardware to support a hardware-backed version is available. While there is no actual implementation of a Secure Element as key storage platform this is also analyzed as it could be used as key store platform. The following results are found:

- For the Bouncy Castle key store the security guarantees are as expected from a key store that is only protected by software. No vulnerabilities are found. However, care should be taken by the programmer how and where to store the actual key store. This is left to the programmer to decide, and mis-usage could case security issues. The Bouncy Castle library allows the key store to be encrypted using an user-provided password. This may, however, not always be possible do to for example usability constrains. Therefore two use cases are analyzed, with and without an user-provided password. When used as recommended (see Section 3.9.1 for the recommendations) the Bouncy Castle key store ensures the confidentiality and integrity of keys stored within the store in both cases for attackers that do not have root credentials.

  For the user-provided password use case the attacker has to learn the password to violate the integrity and confidentiality of the key store. This password is not stored in plain text on the device. So it can only be learned by inspecting either the memory of the device when the user enters his password or by learning the password in another way (such as shoulder surfing). Therefore the conclusion is that if an user-provided password is used for the Bouncy Castle key store an attacker needs root credentials and the ability to learn the password by inspecting for example memory to violate the integrity and confidentiality of the key store.

  If no user-provided password is used the password is assumed to be stored somewhere on the system. Since an attacker with root credentials can inspect the whole filesystem he can learn this password and use it. Therefore if no user-provided password is used, an attacker with root credentials can violate the integrity and confidentiality of the the key store. The use case with an user-provided password also provides the possibility the require user-consent for using a key from the key store. Without an user-provided password this is not possible.

- The results for the `AndroidKeyStore`, however, are more surprising. This key store offers an interface to communicate with a trustlet in the TEE on the mobile device. This allows for key storage that is protected by hardware-based security features. The `AndroidKeyStore` uses a separate service in Android OS for handling key storage operations. Applications communicate with this service using inter-process communication. The `AndroidKeyStore` stores all (possibly encrypted) key data in a single directory. The allocation of keys to application is controlled by embedding the user-id of the application in the filename of the key-file. However, the analysis showed that regardless of the fact that the keys are hardware-backed, keys can be assigned to other applications within the same phone by an attacker that has root privileges. This is simply a matter of renaming the files stored in the `AndroidKeyStore` directory.

  Both the hardware-backed solution on Qualcomm devices and on Texas Instruments devices use an application in the TEE created using TrustZone technology to run cryptographic operations using the keys. The key data is encrypted using a device-specific symmetric key that can only be accessed from TEE in the secure world. The result is that the keys can not be extracted from the device to be used on another device. Since an attacker can use the key on the device by allocating it to an application controlled by the attacker the effect of the hardware-backed key storage is effectively nullified. An attacker can make an *oracle* that he can query to do cryptographic operations using the keys stored on the device. The conclusion is that the hardware-backed `AndroidKeyStore` solutions do bind keys to the device. However, keys are not bond to an application on the device.

  The software-backed `AndroidKeyStore` provides no binding to the device nor to the application. What is even more surprising is that, if no PIN is used to unlock the device, the private key data is not encrypted by the software-backed `AndroidKeyStore`. This means that an attacker that gains root credentials can read the private keys stored in the `AndroidKeyStore` without any effort. When a PIN is required to unlock the device, it is used to encrypt the keys stored in the `AndroidKeyStore`. To decrypt the keys an attacker has to learn the PIN. Since PINs may be of low entropy they may be brute-forced by the attacker. Another option is to learn the PIN from memory while it is entered or learn it by shoulder-surfing.

The conclusion is that secure key storage on Android is protected against attackers that do not have root credentials. When a attacker gains root credentials the `AndroidKeyStore` solutions are not app-bound, since keys can then be assigned to other applications by the attacker. While the TEE-based solution on Qualcomm and TI devices ensure that keys are device-bound, a oracle can be created by an attacker to run key-operations with the private keys stored on the device at will. So the effective protection of key data is limited. A more detailled overview of the results can be found in Section 3.8 on page 45.

To see what is possible in the future and how the TrustZone technology can be used (apart from key storage) we looked at a number of operating systems that could run in the secure world in chapter 4. In this chapter we also discuss a number of use cases for the TEE, both existing solutions and new implementations. We see that there are a number of operating systems either specially build to be run in the secure world (such as SierraTEE and TLR). Others (such as Genode) are suited to be run in the secure world. Genode is particularly interesting because it offers a working solution with Android and it solves the problem of local attestation by disallowing Android to directly write to the display controller.

By using the features and security properties of the TEE provided by the Genode secure world operating system anumber of use cases can be secured. In section 4.2 we discuss how mobile banking, mobile payments and secure communication can be secured by using the TEE. On recent Android versions the TEE is used to securely decrypt and play DRM protected media. Samsung Knox platform also makes use of the TEE. While the implementation of Samsung Knox is closed source, Samsung does provide some details in their documentation. According to this documentation Samsung uses the TEE for key storage and for checking the integrity of the kernel.

The conclusion is that a TEE can be build by using open source secure world operating systems and that there are a number of use cases that can be secured by using the TEE. So secure computation on Android is possible and it offers possibilities to increase the security of mobile banking, mobile payments and secure communication. However, the problem is that device manufacturers control the code that runs in the TEE and decide which trustlets to run in the TEE. So to run your own trustlets in the TEE you should talk to every hardware manufacturer of every device you want to support.

5.1 Future work

TrustZone technology allows a trustlet to inspect the memory of the operating system running in the normal world (in this case Android OS). As discussed in section 3.9.2 on page 49 an interesting solution could be to control and audit cryptographic operations using private keys stored in the TEE by a trustlet. This means that the trustlet checks if an application is allowed to use a certain private key and if the application is not compromised. A number of questions need to be answered:

- How can a trustlet reliably identify the requesting application in Android OS?
- Can an application running in the TEE reliably audit and control cryptographic operations in Android OS?
- Is this a secure solution? Can the protection be circumvented by an attacker?

Another area that could be interesting is the use of the trusted path of the TEE to interact with the user by means of the display and the touchscreen. As discussed in section 3.9.2 on page 49 a trustlet can be created that asks the user to confirm requests of applications to use a private key. A PIN can also be required to confirm a request. Possible questions to answer in future work are:

- How to prevent applications running in Android OS from mimicking a trustlet? Or in other words, how to provide local attestation as discussed in Section 2.2 on page 8? A possible solution could be the solution that the Genode secure world operating system provides. In the case of Genode, Genode controls the display and Android OS can only request the TEE to draw certain content on the display. The question then is whether this is a safe solution? Are there other solutions?
- The TLR paper [50] claims that remote attestation does not have to be provided by the secure world operating system. The authors describe a situation where a trustlet for remote attestation is initialized during manufacturing of the device. Is this a good solution? Can remote attestation indeed be created by using the other properties of a TEE?
- The security of all solutions that are described in this thesis is depending on the security of the TEE. How secure is the TrustZone technology and the secure world operating systems using which a TEE is build? Are the chips resistant to attacks on the hardware? Can the TEE be attacked from software?
- The analysis of Samsung Knox in this thesis is only based on documentation from Samsung and one claim that is dismissed by Samsung. It should be possible to reverse engineer Samsung Knox to learn more about it's inner workings. This could give an insigned in how Samsung Knox is implemented and whether it is it secure.

Bibliography

[1] *About Genode.* URL: http://genode.org/about/index.

[2] L.M. Adleman, R.L. Rivest, and A. Shamir. *Cryptographic communications system and method.* US Patent 4,405,829. Sept. 1983. URL: http://www.google.com/patents/US4405829.

[3] Open Handset Alliance. "Industry leaders announce open platform for mobile devices". *Press release* (2007).

[4] Tiago Alves and Don Felton. "TrustZone: Integrated hardware and software security". *ARM white paper* 3.4 (2004).

[5] *Android 4.4 Feature: NFC Payments No Longer Require Secure Element.* URL: http://www.androidpolice.com/2013/10/31/tap-that-android-4-4-feature-nfc-payments-no-longer-require-secure-element-use-emulation-workaround/.

[6] *Android Pin/Password Cracking: Halloween isn't the Only Scary Thing in October.* URL: http://linuxsleuthing.blogspot.nl/2012/10/android-pinpassword-cracking-halloween.html.

[7] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. "Keying hash functions for message authentication". *Advances in Cryptology-CRYPTO96.* Springer. 1996, pp. 1–15.

[8] *BGU Security Researchers discover Vulnerability in Samsung's Secure Software on the Company's Flagship Device Galaxy S4.* Dec. 24, 2013. URL: http://in.bgu.ac.il/en/Pages/news/samsung_breach.aspx.

[9] *Building a Secure System using TrustZone Technology.* ARM Limited, 2009. URL: http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

[10] *Cisco 2014 Annual Security Report.* URL: https://www.cisco.com/web/offer/gist_ty2_asset/Cisco_2014_ASR.pdf.

[11] TIS Committee et al. "Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2". *TIS Committee* (1995).

[12] *Linux kernel futex local privilege escalation.* National Vulnerability Database. June 7, 2014. URL: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-3153.

[13] Joan Daemen and Vincent Rijmen. "AES proposal: Rijndael" (1998).

[14] Franck B Dahan and Bertrand Cornillault. *Secure mode indicator for smart phone or PDA*. US Patent 20,040,123,118. June 2004.

[15] Lucas Davi et al. "Privilege escalation attacks on android". *Information Security*. Springer, 2011, pp. 346–360.

[16] C Ding, D Pei, and A Salomaa. "Chinese remainder theorem: applications in computing, coding, cryptography" (1996).

[17] John E Dunn. *Eurograbber SMS Trojan steals 36 million euro from online banks*. Dec. 2012. URL: `http://news.techworld.com/security/3415014/eurograbber-sms-trojan-steals-36-million-from-online-banks/`.

[18] *Dutch NFC payments trial starts in Leiden*. URL: `http://www.telecompaper.com/news/dutch-nfc-payments-trial-starts-in-leiden--963975`.

[19] Nikolay Elenkov. *Accessing the embedded secure element in Android 4.x*. Aug. 2012. URL: `http://nelenkov.blogspot.nl/2012/08/accessing-embedded-secure-element-in.html`.

[20] Nikolay Elenkov. *Jelly Bean hardware-backed credential storage*. URL: `http://nelenkov.blogspot.nl/2012/07/jelly-bean-hardware-backed-credential.html`.

[21] Nikolay Elenkov. *Using ECDH on Android*. Dec. 2011. URL: `http://nelenkov.blogspot.nl/2011/12/using-ecdh-on-android.html`.

[22] William Enck, Machigar Ongtang, and Patrick McDaniel. "Understanding Android security". *Security & Privacy, IEEE* 7.1 (2009), pp. 50–57.

[23] William Enck et al. "A Study of Android Application Security." *USENIX security symposium*. 2011.

[24] Adrienne Porter Felt et al. "A survey of mobile malware in the wild". *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM. 2011, pp. 3–14.

[25] Norman Feske. "Introducing Genode". *FOSDEM 2012*. 2012.

[26] "FIPS46-3: Data Encryption Standard (DES)". *National Institute of Standards and Technology* 25.10 (1999).

[27] Bert Flier et al. "The changing landscape of the European financial services sector". *Long Range Planning* 34.2 (2001), pp. 179–207.

[28] *GlobalPlatform made simple guide: Secure Element*. GlobalPlatform. URL: `http://www.globalplatform.org/mediaguideSE.asp`.

[29] *Host-based Card Emulation*. URL: `http://developer.android.com/guide/topics/connectivity/nfc/hce.html`.

[30] Don Johnson, Alfred Menezes, and Scott Vanstone. "The elliptic curve digital signature algorithm (ECDSA)". *International Journal of Information Security* 1.1 (2001), pp. 36–63.

[31] Burt Kaliski. *RFC 2898; PKCS# 5: Password-Based Cryptography Specification Version 2.0.* 2000.

[32] Stefan Kalkowski. "Virtualization Dungeon on ARM - Hands on experience talk about virtualization experiments". *FOSDEM 2014.* 2014.

[33] D Klugler. "Advance security mechanisms for machine readable travel documents, Technical report". *Federal Office for Information Security (BSI), Germany* (2005).

[34] Neal Koblitz. "Elliptic curve cryptosystems". *Mathematics of computation* 48.177 (1987), pp. 203–209.

[35] Sandeep Kumar et al. "How to Break DES for 8,980". *SHARCS06-Special-purpose Hardware for Attacking Cryptographic Systems* (2006), pp. 17–35.

[36] Michael Lu. "TrustZone, TEE and Trusted Video Path Implementation Considerations" (). URL: `http://www.arm.com/files/event/Developer_Track_6_TrustZone_TEEs_and_Trusted_Video_Path_implementation_considerations.pdf`.

[37] Mitsuru Matsui. "Linear cryptanalysis method for DES cipher". *Advances in Cryptology-EUROCRYPT93.* Springer. 1994, pp. 386–397.

[38] Jason Mick. *ARM to Bake On-Die Security Into Next Gen Smartphone, Tablet, PC Cores.* DailyTech. URL: `http://www.dailytech.com/ARM+to+Bake+OnDie+Security+Into+Next+Gen+Smartphone+Tablet+PC+Cores/article24372.htm`.

[39] Charlie Miller, Jake Honoroff, and Joshua Mason. "Security evaluation of Apple's iPhone". *Independent Security Evaluators* 19 (2007).

[40] *NFC Research Lab: Devices.* URL: `http://www.nfc-research.at/?id=45`.

[41] *Notification with regard to electronic signatures in accordance with the Electronic Signatures Act and the Electronic Signatures Ordinance.* URL: `http://www.bundesnetzagentur.de/SharedDocs/Downloads/EN/BNetzA/Areas/ElectronicSignature/PublicationsNotifications/SuitableAlgorithms/2013algokatpdf.pdf?__blob=publicationFile&v=4`.

[42] *NXP's New Generation SWP-SIM Secure Elements Bests Conventional SIMs with Increased Security and Performance.* URL: `http://www.nxp.com/news/press-releases/2012/02/nxp-s-next-generation-swp-sim-secure-element-bests-conventional-sims-with-increased-security-and-performance-.html`.

[43] Global Platform. "The Trusted Execution Environment: Delivering Enhanced Security at a Lower Cost to the Mobile Market". *Whitepaper, February* (2011).

[44] NIST FIPS Pub. "197". *Announcing the Advanced Encryption Standard (AES)* (2001).

[45] Roland van Rijswijk-Deij and Erik Poll. "Using Trusted Execution Environments in Two-factor Authentication: comparing approaches". *Open Identity Summit* 223 (2013), pp. 20–31.

[46] Ronald L Rivest, Adi Shamir, and Len Adleman. “A method for obtaining digital signatures and public-key cryptosystems”. *Communications of the ACM* 21.2 (1978), pp. 120–126.

[47] Michael Roland. “Software card emulation in NFC-enabled mobile phones: Great advantage or security nightmare”. *Fourth International Workshop on Security and Privacy in Spontaneous Interaction and Mobile Phone Use (IWSSI/SPMU 2012)*. 2012, p. 6.

[48] *Samsung’s official response to the recent article on KNOX vulnerability.* Jan. 9, 2014. URL: `https://www.samsungknox.com/en/blog/samsungs-official-response-recent-article-knox-vulnerability`.

[49] Nuno Santos et al. “Trusted language runtime (TLR): enabling trusted applications on smartphones”. *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications.* ACM. 2011, pp. 21–26.

[50] Nuno Santos et al. “Using ARM Trustzone to Build a Trusted Language Runtime for Mobile Applications”. *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems.* ASPLOS ’14. Salt Lake City, Utah, USA: ACM, 2014, pp. 67–80. ISBN: 978-1-4503-2305-5. DOI: `10.1145/2541940.2541949`. URL: `http://doi.acm.org/10.1145/2541940.2541949`.

[51] *SE for Android.* URL: `http://selinuxproject.org/page/SEAndroid`.

[52] Victor Shoup. “A proposal for an ISO standard for public key encryption (version 2.1)”. *IACR E-Print Archive* 112 (2001).

[53] *SierraTEE for ARM TrustZone.* URL: `http://www.sierraware.com/open-source-ARM-TrustZone.html`.

[54] Data Encryption Standard. “Federal Information Processing Standards Publication 46”. *National Bureau of Standards, US Department of Commerce* (1977).

[55] *System Permissions.* URL: `http://developer.android.com/guide/topics/security/permissions.html`.

[56] Henk CA Van Tilborg. *Fundamentals of cryptology.* Kluwer Academic Publishers, 2000.

[57] Amit Vasudevan et al. “Trustworthy Execution on Mobile Devices: What security properties can my mobile platform give me?” *Trust and Trustworthy Computing.* Springer, 2012, pp. 159–178.

[58] *White Paper: An Overview of Samsung KNOX.* URL: `http://www.samsung.com/se/business-images/resource/2013/samsung-knox-an-overview/%7B3%7D/Samsung_KNOX_whitepaper-0-0-0.pdf`.

[59] Johannes Winter. “Trusted computing building blocks for embedded linux-based ARM trustzone platforms”. *Proceedings of the 3rd ACM workshop on Scalable trusted computing.* ACM. 2008, pp. 21–30.

[60] Marc Witteman. “Advances in smartcard security”. *Information Security Bulletin* 7.2002 (2002), pp. 11–22.

[61] Yajin Zhou and Xuxian Jiang. “Dissecting android malware: Characterization and evolution”. *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE. 2012, pp. 95–109.

[62] Philip R Zimmermann. *The official PGP user’s guide*. MIT press, 1995.

Appendix 1: KeyStorageTest MainActivity

Listing 1: MainActivity.java

```java
package com.simplendi.KeyStorageTest;

import android.app.Activity;
import android.os.Build;
import android.os.Bundle;
import android.security.KeyChain;
import android.security.KeyPairGeneratorSpec;
import android.util.Log;
import android.view.View;
import android.widget.TextView;

import javax.security.auth.x500.X500Principal;
import java.math.BigInteger;
import java.security.*;
import java.security.cert.*;
import java.util.*;

public class MainActivity extends Activity {
  // Tag used for identifying this Activity in the logs
  private final static String TAG = "KeyStorageTest";
  // Asymmetric algorithm to use for testing
  private final static String KEY_ALGORITHM = "RSA";
  // Signature algorithm to use
  private final static String SIGNATURE_ALGORITHM = "SHA512WithRSA";
  // Alias of the key to be used
  private final static String KEY_ALIAS = "TestKeyPair";
  // Define the algorithms to check if they are supported by the app
  private final static ArrayList<String> algorithmsToCheck = \
    new ArrayList<String>() {{
    add("RSA");
    add("EC");
    add("DSA");
  }};
   // The reference to the view in the activity that shows the log
  private TextView logView;
  // Reference to the AndroidKeyStore
  private KeyStore androidKeyStore;

  /**
   * Called when the activity class is constructed.
   */
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    logView = (TextView) findViewById(R.id.consoleTextVIew);

    try {
      androidKeyStore = KeyStore.getInstance("AndroidKeyStore");
      androidKeyStore.load(null);
    } catch (Exception exception) {
      writeToLog(exception.toString());
    }

    doStartUpChecks();
  }

  /**
   * Do a number of checks when the Activity is launched.
   */
  private void doStartUpChecks() {
    writeToLog("*** Checking supported algorithms ***");
    for(String algorithm : algorithmsToCheck) {
      if (KeyChain.isKeyAlgorithmSupported(algorithm)) {
        if (KeyChain.isBoundKeyAlgorithm(algorithm)) {
          writeToLog("Algorithm " +algorithm+ " is bound to device");
        } else {
          writeToLog("Algorithm " +algorithm+ " is not bound to device");
        }
      } else {
        writeToLog("Algorithm " +algorithm+ " is not supported");
      }
    }

    writeToLog("*** Checking stored keys ***");

    try {
      Enumeration<String> keyStoreAliases = androidKeyStore.aliases();
      while(keyStoreAliases.hasMoreElements()) {
        String keyStoreAlias = keyStoreAliases.nextElement();
        writeToLog("Found alias: " + keyStoreAlias);
        try {
          X509Certificate certificate = (X509Certificate) \
            androidKeyStore.getCertificate(keyStoreAlias);
          PrivateKey privateKey = (PrivateKey) \
            androidKeyStore.getKey(keyStoreAlias, null);
          writeToLog("\tAlgorithm: " \
            +privateKey.getAlgorithm());
          writeToLog("\tSubject: " \
            +certificate.getSubjectDN().toString());
          writeToLog("\tNot Before: " \
            +certificate.getNotBefore().toString());
          writeToLog("\tNot After:  " \
            +certificate.getNotAfter().toString());

        } catch (Exception exception) {
          writeToLog(exception.toString());
        }
      }
    } catch (Exception exception) {
      writeToLog(exception.toString());
    }
  }

  /**
   * Write a message to the log. Writes to both the the view and
   * the logcat log
   * @param message
   */
  private void writeToLog(String message) {
    logView.append(message + "\n");
    Log.d(TAG, message);
  }

  /**
   * Called when the "Generate Key" button is clicked.
   * @param view
   */
  public void onGenerateKeyButtonClick(View view) {
    KeyPairGenerator rsaKeyGen;
    writeToLog("*** Generating key ***");

    try {
      rsaKeyGen = \
        KeyPairGenerator.getInstance(KEY_ALGORITHM, "AndroidKeyStore");
    } catch (Exception exception) {
      writeToLog(exception.toString());
      return;
    }

    KeyPairGeneratorSpec rsaKeyGenSpec;
    if(Build.VERSION.RELEASE.startsWith("4.4")) {
      rsaKeyGenSpec = new KeyPairGeneratorSpec.Builder(this)
```

```
          .setAlias(KEY_ALIAS)
          .setSubject(new X500Principal("CN=test"))
          .setSerialNumber(new BigInteger("1"))
          .setStartDate(new Date())
          .setEndDate(new GregorianCalendar(2015, 0, 0).getTime())
          .setKeySize(2048)
          .build();
  } else {
    rsaKeyGenSpec = new KeyPairGeneratorSpec.Builder(this)
          .setAlias(KEY_ALIAS)
          .setSubject(new X500Principal("CN=test"))
          .setSerialNumber(new BigInteger("1"))
          .setStartDate(new Date())
          .setEndDate(new GregorianCalendar(2015, 0, 0).getTime())
          .build();
  }

  try {
    rsaKeyGen.initialize(rsaKeyGenSpec);
  } catch (InvalidAlgorithmParameterException exception) {
    writeToLog(exception.toString());
    return;
  }

  rsaKeyGen.generateKeyPair();
  writeToLog("Done");

}

/**
 * Called when the "Delete" button is clicked.
 */
public void onDeleteKeyButtonClick(View view) {
  try {
    androidKeyStore.deleteEntry(KEY_ALIAS);
  } catch (KeyStoreException exception) {
    writeToLog(exception.toString());
  }
}

/**
 * Called when the "Sign Data" button is clicked.
 * @param view
 */
public void onSignDataButtonClick(View view) {
  writeToLog("*** Signing data ***");
  try {
    Signature testKeySignature = \
      Signature.getInstance(SIGNATURE_ALGORITHM);
    PrivateKey testPrivateKey = (PrivateKey) \
      androidKeyStore.getKey(KEY_ALIAS, null);
    testKeySignature.initSign(testPrivateKey);
    testKeySignature.update("TestTestTest".getBytes());
    byte[] signature = testKeySignature.sign();

    Signature testKeySignatureVerf = \
      Signature.getInstance(SIGNATURE_ALGORITHM);
    testKeySignatureVerf.initVerify( \
      androidKeyStore.getCertificate(KEY_ALIAS));
    testKeySignatureVerf.update("TestTestTest".getBytes());
    if (testKeySignatureVerf.verify(signature)) {
      writeToLog("Signature is valid");
    } else {
      writeToLog("Invalid signature");
    }

  } catch (Exception exception) {
    writeToLog(exception.toString());
  }
}}
```