Analysis of Secure Key Storage Solutions on Android

Tim Cooijmans
SNS REAAL
tim.cooijmans@sns.nl

Joeri de Ruiter
Institute for Computing and Information Sciences
Radboud University Nijmegen
joeri@cs.ru.nl

Erik Poll
Institute for Computing and Information Sciences
Radboud University Nijmegen
erikpoll@cs.ru.nl

ABSTRACT

Mobile phones are increasingly used for security sensitive activities such as online banking or mobile payments. This usually involves some cryptographic operations, and therefore introduces the problem of securely storing the corresponding keys on the phone. In this paper we evaluate the security provided by various options for secure storage of key material on Android, using either Android's service for key storage or the key storage solution in the Bouncy Castle library. The security provided by the key storage service of the Android OS depends on the actual phone, as it may or may not make use of ARM TrustZone features. Therefore we investigate this for different models of phones.

We find that the hardware-backed version of the Android OS service does offer device binding – i.e. keys cannot be exported from the device – though they could be used by any attacker with root access. This last limitation is not surprising, as it is a fundamental limitation of any secure storage service offered from the TrustZone's secure world to the insecure world. Still, some of Android's documentation is a bit misleading here.

Somewhat to our surprise, we find that in some respects the software-only solution of Bouncy Castle is stronger than the Android OS service using TrustZone's capabilities, in that it can incorporate a user-supplied password to secure access to keys and thus guarantee user consent.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Cryptographic controls

Keywords

Android; key storage; trusted computing; TrustZone; TEE

1. INTRODUCTION

The use of mobile platforms such as smartphones has grown enormously in the last years and with it also the number of mobile applications (also called *apps*) on these platforms. Naturally this also gained the interest of criminals. 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. Since very early on security has already been an important topic for both Android and Apple iOS, the two main operating systems for mobile phones [7, 12].

One of the basis of securing mobile platforms is the principle of *secure key storage*. Secure key storage offers a secure environment that protects the integrity and confidentiality of the cryptographic keys stored within the environment. To ensure the confidentiality of the keys, basic cryptographic operations using the keys are executed within this secure environment.

Most mobile phones and tablets are based on an ARM processor. ARM does not produce processor itself but rather licences processor designs to chip manufacturers. The manufacturers use this design and add their own features, after which they produce the actual chips. Examples of such chip manufacturers are Qualcomm and Texas Instruments (TI).

To improve the security of solutions such as secure key storage on mobile devices, hardware manufacturers introduced hardware-based security features. One of these features is ARM TrustZone Technology. ARM TrustZone Technology is a hardware-based solution embedded in the ARM processor cores that allows the cores to run two execution environments. These execution environments are 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. In 2012 ARM announced that it would include ARM TrustZone Technology in every processor design they license to manufacturers [11]. As a result, many smartphones today are equipped with a processor with ARM TrustZone Technology.

This article analyses the different secure key storage solutions available on Android smartphones. Our research focused on Android because it is currently the most used operating system on mobile phones and tablets [2]. Also, a large part of the operating system is open source and therefore can easily be inspected. There is a large number of Android smartphones available today produced by various manufacturers. On top of this, each manufacturer creates its own "flavor" of Android OS by modifying the open source software. Because it is infeasible to analyse all different hardware with different versions and variants of Android we focus on the Nexus phones with the original software as provided by Google.

SPSM'14, November 7, 2014, Scottsdale, Arizona, USA.

http://dx.doi.org/10.1145/2666620.2666627.

A more detailed description of this research can be found in [3], the MSc thesis of the first author, which besides options for secure storage also considers possibilities for secure computation on Android devices.

2. RELATED WORK

Though quite some research has been done on Android, not much of this research has focused on key storage. Elenkov discussed the architecture and some details of secure key storage on Android in a number of blog posts, such as ECDH support in Bouncy Castle [4] and an analysis of hardware-backed storage in Android Jelly Bean [5].

Hay and Dayan discovered a vulnerability in the AndroidKeyStore code on Android 4.3 [9]. The vulnerability allows for a stack buffer overflow which could allow for code execution under the rights of the KeyStore process. However the practical exploitability of this vulnerability is limited as noted in the article: the buffer overflow is situated in a variable that limits the data in the buffer. Also note that the vulnerability is in the code running in Android OS, not in the code running in the secure world.

In 2013, Rosenberg showed that the software running in the secure world on Motorola devices based on a Qualcomm chip could be attacked [16]. The vulnerability allows the non-secure world to write to arbitrary locations in the secure world memory. Using this exploit the secure boot could be disabled (also known as "unlocking" the device), which allows the device to run code that is not signed by Motorola. Possibly this attack could allow the master key used for encryption to be extracted.

Teufl et al. discuss the encryption systems used within Android OS in [19]. This includes both disk encryption and credential storage. For the latter, the default functionality offered by new Android versions is discussed. Before, Teulf et al. performed a similar analysis for Apple's iOS [20]. In [18], Shabtai et al. give a more general security of Android OS. Othman et al. propose a high level API for Android in [15] to let developers make easy use of Trusted Computing to perform security sensitive operations.

3. BACKGROUND

3.1 Android OS

Android OS is an operating system developed by the Open Handset Alliance led by Google. It was first released in 2007 [13]. 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 open-sourced. Most apps are written in Java but C++ is also supported. As opposed to apps, OS services on Android are mostly written in C++. Everybody can use the Android operating system on their devices. This means that Android phones are available from a large number of manufacturers. Some manufacturers supply an Android experience that looks and feels like the versions released by the Open Handset Alliance, such as, 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.

A few features of the Android OS are relevant in the light of secure key storage. The first is file storage on Android OS. 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 apps and services running on the operating system.
- `/data/data` is the location for apps to store their data Each app gets its own directory, which is only accessible by the intended app.
- `/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 choose whether to stored 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, also called external storage, can be accessed via USB to write and read all files from it. Developers are recommended to only use internal storage for data that they want to restrict access to[1].

The second feature is the assignment of separate logical user IDs to every individual application and internal services. This is different from a normal Linux system where each user is allocated a user ID and all applications he runs use the user ID assigned to the user.

3.2 TrustZone Technology and the TEE

As discussed in Section 1, ARM TrustZone Technology provides a separation of the hardware in two worlds as depicted in Figure 1. In the normal world runs Android OS or any other operating system and in the secure world security sensitive operations can be handled.

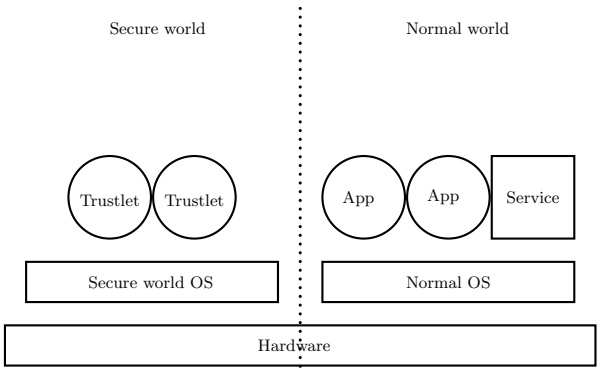

Figure 1: The separation of the hardware by TrustZone into two worlds

One of the main features of ARM TrustZone Technology is the *security bit* on the communication bus [1]. 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). These peripherals can either be located

[1]See `http://developer.android.com/guide/topics/data/data-storage.html#filesExternal`

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 or the normal world. All peripherals should check the security status of the transaction and ensure that they do not leak any sensitive information from the secure world to the normal one.

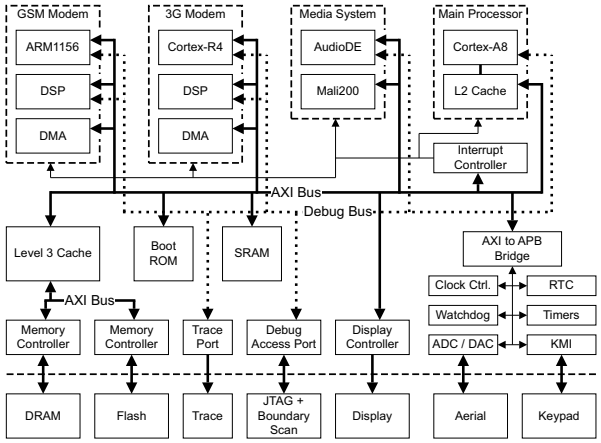

Figure 2: The ARM architecture and its AXI bus (source: [1])

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 [1]. This bit can only be set by the system running in the secure world. When the NS-bit is 0 the processor is operating in the secure world and otherwise it operates in the 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 by means of the security bit. This is in turn used by the peripheral to decide how it should act on a certain transaction.

TrustZone Technology provides hardware features to create a secure environment separated from the normal execution environment. However, the hardware features that are provided do not implement or ensure a secure environment. 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. Hardware features such as the possibility of the secure world to access the memory of the normal world allow these functionalities to be implemented. However, note that the normal world can not inspect the secure world's memory.

To complete the secure environment and to allow multiple apps to be run in the secure world a secure world operating system (*secure world OS*) has to be implemented. This provides an execution environment for apps to run in. This environment is usually called a Trusted Execution Environment (*TEE*). Applications running in the TEE are also 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 context switches both 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.

4. SECURITY MECHANISMS

There are several security mechanisms that can be used to secure access to cryptographic keys on Android:

Android access control As a first line of defence, the Android OS provides conventional access control on files. Because different apps run as different users on Android, this allows access to files to be controlled per app.

TEE Additionally, the TEE provided by the ARM TrustZone Technology, and the separation between the secure and normal world it offers, can be used. There are (at least) two ways of doing this:

1. The app that accesses the key material runs in the secure world. For most apps, however, this will not be an option, as it requires the cooperation of whoever is in control of the TEE to grant permission. We will therefore not consider this option, but we will come back to it when drawing our conclusions in Section 9.
2. The app runs in the insecure world and requests operations that use the keys from a service running in the secure world. This service is responsible for storing and using these keys. This is an option we will consider, as on some Android devices such a service is available, as discussed in sections 8.3 and 8.4.

Password-protected storage Independently of the security mechanisms above, access to key material could be further protected by an additional encryption key or password. Of course, this introduces the problem of where to store this encryption key or password. We will refer to this encryption key as a password to avoid confusion. Here we have two options:

Stored password An app could simply store the password in the file system or use white-box crypto techniques to try and hide it in its code. Of course, an attacker with access to the code and data of an app, and with enough time and patience, can then in principle always recover the password.

User-provided password An app could rely on the user to provide the password which is used as the encryption key, or used to derive the encryption

key. An attacker with access to the code and files of an app can then no longer retrieve its key material. This approach might leave a possibility open for a brute-force attack. Such an attack could work if the password is checked locally on the phone and the attacker can determine when he guessed the correct password. This could be countered if the user-provided password provides enough entropy to make this attack infeasible. Note that this may not be a realistic assumption for most smartphone users: entering a password with sufficient entropy on a smartphone is not very user-friendly. An attacker could of course still try to obtain the user-supplied password by eavesdropping on the running system with, for instance, a key logger, or by some form of phishing that tricks the user into providing the password to the attacker.

5. ATTACKER MODELS

To evaluate the security guarantees, which we will discuss in Section 6, offered by the different solutions, we define three attacker models. In increasing power, these are:

Malicious app attacker This attacker tries to attack the secure key storage from another app installed on the same device. The app is assumed to be installed from an app-market (such as the Google Play-store or the Amazon appstore). The attacker is assumed to be able to use all permissions that an app that is installed from such store can request.

Root attacker This attacker has root credentials and is able to run apps under root permissions and inspect the file system. This models an attacker that either uses an exploit to gain root permissions on the device or has the ability to run an application with root permissions. Note that this is not an unrealistic scenario as many users today enable root permissions on their phone to work around the permission model. There are even apps in the Google Play Store that require root permissions such as, for example, Titanium Backup[2] which can be used to backup phones.

Intercepting root attacker This attacker has the same abilities as the *root attacker* but is also able to capture user-input in Android OS as the user enters it. This can be done by, for example, inspecting the memory of the device.

6. SECURE KEY STORAGE SOLUTIONS

Android provides a number of secure key storage solutions. The first version of Android, API level 1, already provided cryptographic operations and key storage. It has a standardised interface to store key material. An abstract class defines an interface to store keys and facilitates getting an instance of a particular storage method. The actual storage of keys is provided by different *keystore types*. Two commonly used keystore types are analysed:

Bouncy Castle Bouncy Castle is a cryptographic library for Java[3]. It is provided on all Android versions encountered and provides a keystore type. The Android version of Bouncy Castle [4] is a limited version of the regular Bouncy Castle library. 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.[4] The Android API provides a method that returns the default keystore type on the device. On all the devices we analysed, the default keystore is Bouncy Castle (indicated by `BKS`). Keys are stored using file-based keystores.

AndroidKeyStore This type is added in API Level 18. It communicates with a service called `KeyStore` 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. AndroidKeyStore does not provide an API to use a user-provided password to protect the stored keys.

Different devices provide different realisations for AndroidKeyStore, which may or may not involve a TEE. As can be seen in Table 1, the same model phone may or may not provide a TEE depending on the version of Android running on it. When the Bouncy Castle library is used, the password could be user-supplied or stored on the file system. All this means we end up with several options to compare:

1. **Bouncy Castle using a stored password** This option does not use the TEE or a user-provided password. The keystore is encrypted using a password that is stored in the application-specific data directory as discussed in Section 3.1.

2. **Bouncy Castle using a user-provided password** This option uses a user-provided password with sufficient entropy to store the keystore. However, the TEE is not used.

3. **AndroidKeyStore using the TEE on Qualcomm devices** On devices that have a Qualcomm processor with TrustZone Technology, the TEE is used to secure AndroidKeyStore.

4. **AndroidKeyStore using the TEE on TI devices** Similar to the previous option, except a device with a processor by Texas Instrument is used. The actual implementation differs from the Qualcomm implementation, so both solutions are analysed separately.

5. **AndroidKeyStore using software-fallback** This option concerns the case where no TEE (or no driver for the TEE) is available on the device for AndroidKeyStore to use. As discussed before, no user-provided password can be used.

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

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

[4] 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 (`http://rtyley.github.io/spongycastle/`) and can be included in any application.

When analysing the different key storage solutions, we distinguish three security requirements:

App-binding The key can only be used by an instance of a certain application on a certain device. The key can not be used by another application or on a other device.

Device-binding The key can only be used on a certain device.

User-consent required The key can only be used when the user wants to use the key and has given his explicit consent to do so.

The requirements of App-binding and Device-binding are related: App-binding is a stronger variant of Device-binding. If App-binding is guaranteed then, by definition, Device-binding is also guaranteed.

7. METHOD

To test the keystores on Android we created the `KeyStorageTest` application. On start-up the `KeyStorageTest` application first checks if the cryptographic algorithms RSA, ECDSA and DSA are claimed to be bound to the device (if the necessary method is available). This check is done using the `isBoundKeyAlgorithm(String algorithm)` method of the `KeyChain` class. 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 stores all keys in the keystore under an alias. This alias is a string defined by the programmer that is used to identify a key or key pair. The `KeyStorageTest` application provides functionality to generate a new key pair and delete it again. This key pair can be used to generate signature on specific data.

The `KeyStorageTest` application is used to generate a RSA key pair using the `KeyPairGenerator`. 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.

Two instances of the `KeyStorageTest` application are installed on a device. 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. This is repeated for each of the three attacker models with increasing privileges. We also look at the APIs to see if they have the possibility to require user input or user consent to use a key. If the APIs offer a way to require the user's consent, we try to use the key without this consent. The phones that are used in our tests are listed in Table 1. These phones are chosen because they provide represent both high-level and low-level phones running recent versions of Android and the older Android 2.3 that is still commonly used today.

8. RESULTS

8.1 Bouncy Castle using stored password

According to the Bouncy Castle documentation[5] the default format only protects against tampering but not against inspection. This is clearly the case for the certificates stored in the keystore. The certificates stored in the keystore can be read using OpenSSL's asn1parse tool[6].

To ensure the integrity of the keystore, a password needs to be provided while storing the keystore and when the integrity needs to be verified. Without this password the integrity of the whole keystore cannot be verified. The private and secret keys stored in the keystore are protected against inspection. These keys are encrypted using the *pbeWithSHAAnd3-KeyTripleDES-CBC* encryption scheme, as defined in the PKCS #12 standard [17]. No vulnerabilities are known for the encryption algorithm that is used. The password needs to be provided by the application in order to use the keys stored in the keystore.

Because the test application uses best practices, which include storing the keystore file in the application specific data directory the keystore file cannot be accessed by other apps without root permissions (see Section 3.1 for more details). Since the *malicious app attacker* without root permissions cannot access the keystore file, the attacker cannot retrieve the private keys.

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 used when storing the keystore file. By copying both files to another device running the same application we could successfully create a valid signature using the private key stored in the keystore. The attacker does not have to inspect the memory of the device because all data needed is stored in files. The *root attacker* can therefore violate the confidentiality of the keys and use them in other applications and on other devices.

8.2 Bouncy Castle using user-provided password

If a user-provided password is used to store a Bouncy Castle keystore it is still possible to copy the keystore file to another device by an attacker that gained root permissions. However, the key entries in it can not be used without the user-provided password. This password itself is not stored on the device. A way to learn this password is to intercept it in memory when the user enters it. The *intercepting root attacker* would therefore be able to compromise keys stored using this method.

Another possibility is to brute-force the password. The entropy of the password used to encrypt the keystore may be limited. Passwords that only consists of 4 or 5 digits are regularly used on phones since entering a long and complex password on a mobile device is cumbersome. When such a low entropy password is used, an attacker that has access to the keystore file can easily brute-force the password. As a result, when a low entropy password is used, the *root attacker* may be able to gain access to the data stored in the keystore. Using the integrity check provided by the Bouncy Castle keystore, the attacker can determine whether a password guess was correct.

8.3 AndroidKeyStore on Qualcomm devices

AndroidKeyStore on Qualcomm is implemented using the `KeyMaster` service running in Android OS and a trustlet running in the TEE. An application that uses AndroidKeyStore

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

[6] `https://www.openssl.org/docs/apps/asn1parse.html`

Phone	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

* Disabled by default

Table 1: Phones used in the evaluation

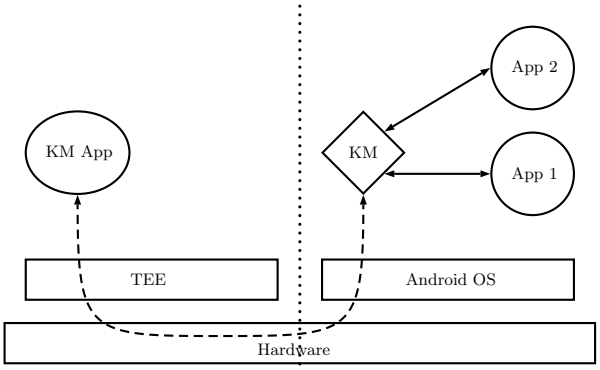

Figure 3: Architecture of AndroidKeyStore on Qualcomm devices

communicates using Inter-Process Communication with the `KeyMaster` service. This `KeyMaster` service in turn communicates with a trustlet in the TEE that is responsible for the key operations. This process is shown in Figure 3. The actual keys are not stored in the TEE. Two files are created in the `/data/misc/keystore/user_0` directory when a new key pair is generated:

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

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 in the Android OS under which the application is running. Each Android application is allocated its own user ID as discussed in Section 3.1. The key entry alias can be chosen by the programmer.

The `/data/misc/keystore/user_0` directory where the key-entry files are stored is not accessible by a non-root user and apps can not access the private key material of another app. The usage of the application user ID in the filename of the key-entry files is suspicious. And indeed this is what controls the assignment of keys to certain applications. 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_TKP 10101_USRCERT_TKP
cp 10102_USRPKEY_TKP 10101_USRPKEY_TKP
```

The private keys will then be directly accessible to the other application. So the *root attacker* can use the keys in another application on the device. The key-files seem to be encrypted using a device-specific key that is stored in the TEE and cannot be retrieved.

8.4 AndroidKeyStore on TI devices

AndroidKeyStore on Texas Instruments devices also stores two files in the `/data/misc/keystore/user_0` directory for each key pair. However, the actual contents of the private key file is different than with the Qualcomm implementation. The private key cannot be stored in the private key file since the file size is smaller than the actual private key used. An analysis of the keystore by Elenkov shows that the actual private key data is stored encrypted in `/data/smc/user.bin` [5]. The actual format is unknown and again appears to be encrypted using a device-specific key stored in a trustlet in the TEE.

The naming of the key entry files stored on devices that have a TI processor and use AndroidKeyStore is exactly the same as the Qualcomm-based version discussed in Section 8.3. Again the *root attacker* can rename or copy the files to assign them to another application. So, again, App-binding is not achieved. However, since the keys are encrypted using a device-specific key, no attacker can use the keys on another device and Device-binding is therefore still achieved.

8.5 AndroidKeyStore using software-fallback

The naming of the key entry files when a software-based keymaster is used is the same as we have seen with the Qualcomm-based key storage: 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 app and use the keys in that application. This issue appears to be specific to AndroidKeyStore and not to the actual implementation of the key storage that AndroidKeyStore uses.

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.

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 needed to reconstruct the private key such as the private exponent and the two primes of the RSA key pair. An attacker that does not have root permissions (*malicious app attacker*) cannot access the keystore directory and therefore cannot rename or copy key entry files. The *root attacker* 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 off the device. Therefore Device-binding is not achieved in presence of the *root attacker*.

Even when requiring a PIN to unlock the device, an attacker that has root permissions (*root attacker*) can assign private keys to other applications on the same device by renaming the (possibly encrypted) files as shown in Section 8.3 and discussed above. However, the *root attacker* has to decrypt the private key entries to be able to use them outside of the device. To do 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 [10]. The *intercepting root attacker* can however learn the necessary information when the user inputs it and therefore breaks the Device-binding requirement.

9. DISCUSSION

The results for the security requirements Device-binding, App-binding and User-consent as defined in Section 6 on the various phones can be found in tables 2, 3 and 4 respectively.

A first thing to note is that all keystores protect against the *malicious app attacker*. The use of hardware-based security features ensures that the AndroidKeyStore solutions that use the TEE guarantee device binding, even against the *intercepting root attacker*. However, when we look at App-binding we see that only the Bouncy Castle keystore provides App-binding when used with a password with sufficient entropy. While AndroidKeyStore provides Device-binding, an attacker can create a signing oracle on the device to query for signatures over arbitrary data. So while the attacker cannot gain access to the private key, he can effectively use it without any limitations. This raises the question what the value is of device binding when app binding is not guaranteed. It does make it harder for an attacker to always have access to a key. However, with devices being connected to the internet most the time nowadays, this is not necessarily a major obstacle for the attacker.

When looking at the results for User-consent and App-binding, the same solutions seem to provide the same security. This is not surprising, as User-consent is a stronger requirement than App-binding. If an attacker cannot break App-binding, he will not be able to break User-consent either. For User-consent there is an additional risk, namely when considering malicious app developers. An example of this is when using an email app that can also sign messages. The developer of the app could modify it to sign other data apart from the user's messages as well. This cannot be prevented in the solutions we considered in this paper. A possible countermeasure for this would be a TEE that has a trusted path to the user such that it can display the message to be signed to the user and get confirmation directly without it having to go through the untrusted operating system. This would provide a what-you-see-is-what-you-sign solution.

Given the results we wonder what is actually achievable. Is it possible to have a secure method to bind keys stored in the secure world to apps running in the normal world? A possible solution could be to inspect the integrity of the normal world from the secure world as we discuss in Section 9.1. However, more research should be done on how to actually ensure and check the integrity of (components of) the normal world from the secure world.

In the end, the additional security that TEE-backed secure key storage provides is Device-binding. It provides security against malicious apps, but not against a root attacker. This is a fundamental limitation when using a secure service for key storage from the secure world to the normal world. Such a service will have to rely on the operating system running in the insecure world to identify the app asking access to some key. Still, the developers of AndroidKeyStore could have made it a bit more difficult for a root attacker. Now the attacker only needs to rename a file to obtain access from a different app.

There may be other ways to do secure key storage that were not discussed in this paper. An option that we have not considered here is the use of a *secure element* (SE), where keys could be stored in the SIM card or an embedded chip in the phone.

With respect to app binding, the Android documentation is a bit misleading. In the list of security enhancements in Android 4.3 the documentation notes that[7]:

> "AndroidKeyStore Provider. Android now has a keystore provider that allows applications to create exclusive use keys. This provides applications with an API to create or store private keys that cannot be used by other applications."

In this paper we showed that this is clearly not true. However, on the same page it is also noted that:

> "KeyChain isBoundKeyAlgorithm. Keychain API now provides a method (isBoundKeyType) that allows applications to confirm that system-wide keys are bound to a hardware root of trust for the device. This provides a place to create or store private keys that cannot be exported off the device, even in the event of a root compromise."

This does correctly indicate that it is possible that key cannot be exported off the device, which is true for AndroidKeyStore using the TEE.

It is a pity that AndroidKeyStore does not provide support for the use of a password to encrypt the key storage files like Bouncy Castle does. When done correctly, this could make attacking AndroidKeyStore more difficult, as the attacker then also needs to learn the password. Also, this provides a way to implement for user consent.

For the average user it is hard to see if their phone provides the security of hardware-backed secure key storage. For example, initially the Motorola Moto G was sold without hardware-backed secure key storage – it did have the

[7] `https://source.android.com/devices/tech/security/enhancements43.html`

Solution	***Malicious app attacker***	***Root attacker***	***Intercepting root attacker***
Bouncy Castle with stored password	✓	×	×
Bouncy Castle with user-provided password	✓	✓*	×
AndroidKeyStore using the TEE on Qualcomm devices	✓	✓	✓
AndroidKeyStore using the TEE on TI devices	✓	✓	✓
AndroidKeyStore using software-fallback without a PIN to unlock the device	✓	×	×
AndroidKeyStore using software-fallback with a PIN to unlock the device	✓	✓	×

* If password has sufficient entropy

Table 2: Overview of the results for Device-binding

Solution	***Malicious app attacker***	***Root attacker***	***Intercepting root attacker***
Bouncy Castle with stored password	✓	×	×
Bouncy Castle with user-provided password	✓	✓*	×
AndroidKeyStore using the TEE on Qualcomm devices	✓	×	×
AndroidKeyStore using the TEE on TI devices	✓	×	×
AndroidKeyStore using software-fallback without a PIN to unlock the device	✓	×	×
AndroidKeyStore using software-fallback with a PIN to unlock the device	✓	×	×

* If password has sufficient entropy

Table 3: Overview of the results for App-binding

Solution	***Malicious app attacker***	***Root attacker***	***Intercepting root attacker***
Bouncy Castle with stored password	✓	×	×
Bouncy Castle with user-provided password	✓	✓*	×
AndroidKeyStore using the TEE on Qualcomm devices	✓	×	×
AndroidKeyStore using the TEE on TI devices	✓	×	×
AndroidKeyStore using software-fallback without a PIN to unlock the device	✓	×	×
AndroidKeyStore using software-fallback with a PIN to unlock the device	✓	×	×

* If password has sufficient entropy

Table 4: Overview of the results for User-consent

required hardware for it but not the required software and drivers – but later it received an update that enabled the hardware-backed key storage. Another example is the Samsung Galaxy Nexus, for which drivers that allow hardware-backed key storage are available but they are not used on production devices. Sometimes a phone model exists with different hardware configurations that may or may not offer TrustZone; for example, the Samsung Galaxy S3 may contain a Samsung Exynos 4 Quad processor or a Qualcomm Snapdragon S4 MSM8960, and only the former supports TrustZone.

9.1 Recommendations

To improve AndroidKeyStore we recommend the three improvements listed below. These do not require any change to the architecture, but we should stress that these improvements do not really solve the fundamental problem, and – especially the first two – only make things a little bit harder for an attacker.

- Encrypt the keystore 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 does makes it harder for attackers to read a keystore file: the attacker first needs to derive the key before begin able to retrieve the keystore file.
- Include the user ID of the application that generated the key pair in the integrity checked section of the keyblob in the keystore file. This again does not solve the whole problem: an attacker can change the user ID of the application that uses the files. However, this is again harder than just renaming the files.
- Allow encryption of the keystore file using a user provided password. This provides the possibility to require user consent to use a key.

Solving the problem that multiple applications on the same device can access a key is not trivial. As discussed, adding an integrity check on the user ID of the application in the keystore file does not solve the problem as the user ID of an application can be changed. A possible solution could be to check the signature or application ID of the application requesting a key operation from the TEE. This can be done as the TEE has access to the memory of the Android OS. This solution may be very hard for an attacker to work around. However, we are not aware of any research that has been done in this area and further research is needed to validate whether this is actually a viable and secure solution.

10. CONCLUSIONS

We reported the findings of our research to Google. After our research, the source code of a new beta version of Android (Android L) was released. This release includes a fix to make the process of renaming the keystore files of AndroidKeyStore harder. The assignment of keys to certain applications is now also checked using SELinux permissions. Though this does make it harder to rename the files, the root attacker would be able to change the SELinux permissions and therefore still be able to rename the files. A future fix could use the capability of the TEE to communicate directly and securely with the user of the device. It could ask for user consent in a secure way. By displaying the data to be signed it could even provide a what-you-see-is-what-you-sign solution. Of course, providing a trusted user interface is one of the obvious uses of a TEE, and already standardised in the Global Platform specifications [8].

In the end, the addition of AndroidKeyStore to Android OS can provide *device binding* against a root attacker on devices where the TrustZone TEE is used. Even though some documentation on AndroidKeyStore suggests that it also provides *app binding*, this is not the case in the presence of a root attacker. When you think about it, this is not so surprising, given the inherent limitations of what be done against a root attacker; still, we were surprised how easy it was, simply by renaming a file.

We were surprised to find that in one respect the Bouncy Castle key storage can provide stronger security guarantees than the hardware-backed AndroidKeyStore using the TEE, even though the former only relies on software: because the AndroidKeyStore does not provide a way to require a password for using a key, Bouncy Castle with a user-provided password is the only solution that can guarantee user consent.

The leading standard for electronic payments using smartcards, EMV[8], has recently been extended to account for different security levels of key storage, amongst other things [6]. One of the motivations for this extension is the use of NFC-enabled mobile phones as replacement for contactless smartcards. It will be interesting to see whether EMV's classification of key storage solutions will in the future be refined to include some of the security levels discussed in this paper, such as device-binding, something that is already suggested in [14].

11. REFERENCES

[1] Building a secure system using Trustzone Technology. Technical report, ARM Limited, 2009. `http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf`.

[2] Apple cedes market share in smartphone operating system market as Android surges and Windows phone gains, according to IDC, August 2013. `http://www.businesswire.com/news/home/20130807005280/en/`.

[3] T. Cooijmans. Secure key storage and secure computation in Android. Master's thesis, Radboud University Nijmegen, 2014.

[4] N. Elenkov. Using ECDH on Android, December 2011. `http://nelenkov.blogspot.nl/2011/12/using-ecdh-on-android.html`.

[5] N. Elenkov. Jelly Bean hardware-backed credential storage, July 2012. `http://nelenkov.blogspot.nl/2012/07/jelly-bean-hardware-backed-credential.html`.

[6] EMVCo. *EMV Payment Tokenization Specification. Technical Framework (version 1.0)*, 2014.

[7] W. Enck, M. Ongtang, and P. McDaniel. Understanding Android security. *IEEE Security & Privacy*, 7(1):50–57, 2009.

[8] Trusted User Interface API Specification v1.0. Technical report, Global Platform, 2013.

[8] EMV stands for Europay-Mastercard-Visa.

[9] R. Hay and A. Dayan. Android KeyStore stack buffer overflow - CVE-2014-3100, 2014.

[10] J. Lerr. Android pin/password cracking: Halloween isn't the only scary thing in October, October 2012. http://linuxsleuthing.blogspot.nl/2012/10/android-pinpassword-cracking-halloween.html.

[11] J. Mick. ARM to bake on-die security into next gen smartphone, tablet, PC cores, april 2012. http://www.dailytech.com/ARM+to+Bake+OnDie+Security+Into+Next+Gen+Smartphone+Tablet+PC+Cores/article24372.htm.

[12] C. Miller, J. Honoroff, and J. Mason. Security evaluation of Apple's iPhone. *Independent Security Evaluators*, 19, 2007.

[13] Industry leaders announce open platform for mobile devices, 2007. Press release.

[14] D. Ortiz-Yepes. A critical review of the EMV Payment Tokenisation Specification. *Computer Fraud and Security*, 2014. To appear.

[15] A. T. Othman, S. Khan, M. Nauman, and S. Musa. Towards a high-level trusted computing API for Android software stack. In *Proceedings of the 7th International Conference on Ubiquitous Information Management and Communication*, ICUIMC '13, pages 17:1–17:9. ACM, 2013.

[16] D. Rosenberg. Unlocking the Motorola bootloader, 2013. http://blog.azimuthsecurity.com/2013/04/unlocking-motorola-bootloader.html.

[17] RSA Laboratories. PKCS #12 v1.0: Personal information exchange syntax, 1999.

[18] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer. Google Android: A comprehensive security assessment. *IEEE Security and Privacy*, 8(2):35–44, March 2010.

[19] P. Teufl, A. G. Fitzek, D. Hein, A. Marsalek, A. Oprisnik, and T. Zefferer. Android encryption systems. In *International Conference on Privacy & Security in Mobile Systems*, 2014. To appear.

[20] P. Teufl, T. Zefferer, C. Stromberger, and C. Hechenblaikner. iOS encryption systems - deploying iOS devices in security-critical environments. In *SECRYPT*, pages 170 – 182, 2013.