# Android Encryption Systems

Peter Teufl, Andreas Fitzek, Daniel Hein, Alexander Marsalek, Alexander Oprisnik, Thomas Zefferer
Institute for Applied Information Processing and Communications
Graz University of Technology
Inffeldgasse 16a
8010 Graz, Austria

***Abstract*—The high usability of smartphones and tablets is embraced by consumers as well as the corporate and public sector. However, especially in the non-consumer area the factor security plays a decisive role for the platform-selection process. All of the current companies within the mobile device sector added a wide range of security features to the initially consumer-oriented devices (Apple, Google, Microsoft), or have dealt with security as a core feature from the beginning (RIM, now Blackerry). One of the key security features for protecting data on the device or in device backups are encryption systems, which are available in the majority of current devices. However, even under the assumption that the systems are implemented correctly, there is a wide range of parameters, specific use cases, and weaknesses that need to be considered when deploying mobile devices in security-critical environments. As the second part in a series of papers (the first part was on iOS), this work analyzes the deployment of the Android platform and the usage of its encryption systems within a security-critical context. For this purpose, Android's different encryption systems are assessed and their susceptibility to different attacks is analyzed in detail. Based on these results a workflow is presented, which supports deployment of the Android platform and usage of its encryption systems within security-critical application scenarios.**

## I. Introduction

Data encryption represents a central concept of security-critical applications. During the past years, data-encryption mechanisms have been included in all major mobile device platforms, such as iOS, Android, Windows Phone or Blackberry. However, the deployed encryption systems differ in various security-related aspects. For instance, different platforms rely on different approaches to encrypt data (file-based encryption vs. file-system based encryption) and implement different methods to derive required encryption keys from user input (e.g. PIN or passcodes[1]). Furthermore, different platforms offer developers, administrators and end users different options to use and configure provided encryption features.

This raises several challenges, when the platform should be deployed in a security-critical context. In this case, a security analysis for a given mobile device platform must consider a wide range of different security-related aspects. Even under the assumption that built-in encryption systems and algorithms are implemented correctly, there are still many other aspects that can render the systems ineffective. Examples for such high-level aspects are parameters related to system configuration, application development, or the utilization of backup systems. In fact, those higher-level aspects play a vital role when deploying a mobile platform within a security-critical context.

[1]Subsequently, we will refer to PIN and passcodes with the term passcode.

The topic of data encryption on mobile platforms has already been approached from different points of view by various authors. The importance of encryption solutions for mobile devices and possible implications on jurisdiction have been discussed in [1]. Proprietary encryption solutions for smartphone platforms have for instance been proposed in [2] and [3]. Interestingly, most related work on encryption systems for mobile devices does not focus on encryption systems that are already included into the platform, but on the development of proprietary solutions.

This also applies to the popular Android platform. While there are several papers on proprietary encryption solutions for Android-based devices, related scientific work on Android's built-in encryption mechanisms is still rare. Despite publicly available official information on the general design of the Android encryption system[2], only few papers have investigated the security and reliability of Android's encryption solution. An interesting contribution to the evaluation of the robustness of Android's built-in encryption system has been provided by Müller et al. [4]. In this paper, the authors show that protected data can be retrieved from encrypted phones using cold boot attacks. Representing a physical attack scenario, which reveals protected data by reading it out directly from the RAM after freezing the device, this approach however requires significant effort.

Compared to the Apple iOS platform, whose encryption features have for instance been investigated and discussed in [5], [6], or [7], Android's encryption system has not been subject to detailed assessments by the scientific community so far. This seems reasonable at a first glance, as Android's encryption system is rather simple and provides less features compared to the encryption system that is for instance integrated into Apple's iOS platform. Still, Android's encryption system needs to be well understood in order to be able to assess its security and its capability to protect security-critical data. This paper contributes to a deeper understanding of Android's encryption system by analyzing its core security features and identifying strengths, potential weaknesses, and possible attack scenarios. This way, this paper provides a solid basis for deploying the Android platform also in security-critical fields of application that require reliable data protection by means of encryption. To assure to be close to reality and to provide support for real-world scenarios, we assume that cryptographic functions are implemented correctly and put focus on high-level issues regarding passcode properties, relations between different crypto systems, and the developer's

[2]https://source.android.com/devices/tech/encryption/android_crypto_implementation.html

and administrator's role regarding the security of the deployed systems.

The remainder of this paper is structured as follows. In Section II, we define threats and assumptions, on which we have based our analysis of Android's encryption system. Subsequently, we define common assessment criteria that are used to systematically assess Android's encryption systems in Section III. The different encryption systems are then discussed and assessed in Section IV. Furthermore, this section identifies potential attack scenarios based on obtained assessment results. Form the identified attack scenarios, a workflow is derived and presented in Section V. This workflow supports a systematic deployment of Android in security-critical scenarios.

## II. Threats and Assumptions

The presented security analysis is based on the general scenario that a security officer of a company or public agency is in charge of deploying the Android platform to allow employees to process and store security-critical data with mobile Android devices. This is a common scenario, as mobile devices are increasingly issued to employees, in order to improve efficiency.

As mobile devices are much more likely to be subject to loss or theft than classical computing devices such as desktop PCs, theft is the main threat to be considered by the security analysis presented in this paper. Another potential threat is malware that is installed by an attacker on a mobile device to spy on security-critical data. However, the focus is not put on this threat for several reasons. First, according to the underlying scenario, we can assume a controlled environment and hence managed mobile device that is under control of a mobile device management (MDM) solution. As for managed devices the set of installed software can be controlled by an administrator (or security officer), the risk of installed malware can be neglected. Of course, also managed devices are prone to highly sophisticated malware that exploits system vulnerabilities to gain root access to the operating system. However, this kind of malware must be assumed to have almost unlimited capabilities, including the capability to circumvent any encryption system and security feature in place. For these reasons, this work mainly focuses on the identified threat *theft*.

Regarding the identified threat *theft*, several additional assumptions apply. **First**, our assessments are based on the assumption that the Android encryption algorithms are implemented correctly. The goal of this assessment is to analyze weaknesses located on a higher level, such as bad configurations, weak passwords, limits of used key derivation functions, or wrong assumptions in relation to the encryption scope (e.g., files vs. file-system). **Second**, we assume that a passcode-locked Android device is stolen by an attacker who is an expert with in-depth knowledge about the deployed encryption systems and their weaknesses. This scenario is similar to the one faced by a forensic expert who needs to analyze the data stored on an Android device. In this context we also assume that the attacker employs jailbreaking/rooting tools. This is the only type of malware that will be considered in the conducted analysis.

## III. Assessment Criteria

The conducted assessment of Android's encryption systems has been basically based on two assessment criteria. These criteria are discussed in this section in detail, in order to motivate their inclusion in the conducted analysis process.

### A. Key Derivation

The appropriate derivation of encryption keys is a crucial aspect of any encryption system. This also applies to smartphone platforms, where key-derivation functions (KDF) are used to derive encryption keys from some kind of securely protected master key or from secret credentials entered by the user. Thus, the used KDF represents the first assessment criterion.

In general, key-derivation functions need to meet the following security requirements [8]: **First**, the KDF must be a deterministic function used to derive cryptographic keying material from a secret value (e.g., a password). The KDF is defined by the used PseudoRandom Function (PRF) (e.g., a SHA-1 based HMAC) that is executed for multiple iterations. **Second**, the key-derivation process should be rather slow, in order to significantly slow down brute-force attacks. This can be achieved by using a large number of iterations. **Third**, a random salt (at least 128 bits) is required to allow the generation of a large set of keys for a given password. Since this value is randomly generated on each device, an attacker cannot generate a table of possible keys prior to gaining access to the device and the stored salt value.

Two wide-spread examples for KDFs are the PBKDF2 [9] and SCRYPT functions [10]. Typically, KDFs are based on a common design. They take as input a random salt value, a passcode defined by the user, and/or a master key provided by a hardware element. While the salt input is obligatory, reliance on the user input and on a hardware-protected key actually depends on the particular implementation. The chosen input parameters for the used KDF directly influence the security of the derived keys and hence the overall security of the encryption system. Usually, the derived key is not directly used to encrypt the respective data, but only to protect the actual data encryption key. This two-stage approach is required to allow changing the passcode without the need to re-encrypt the protected data. If a secret key provided by a secure hardware element is incorporated into the key-derivation process (e.g., as in the iOS encryption system [6]), the key derivation is bound to the mobile device, as the secret key cannot be extracted from the hardware element. This means that brute-force attacks cannot be outsourced to powerful external components, rendering the parallelization of these attacks infeasible. Integration of a user-defined passcode into the key-derivation process is also relevant in terms of security. If the derived key depends on a device-specific key only (e.g., provided by a secure hardware element), attackers can easily circumvent any encryption system by taking over control of the operating system (e.g., by employing known vulnerabilities that enable the attacker to root or jailbreak the device).

### B. Configuration Capabilities

Configuration capabilities of a provided encryption system represent another relevant assessment criterion. Depending on the particular smartphone platform, encryption systems can be configured by the developer of smartphone applications and/or by the user during runtime.

Depending on the smartphone platform, encryption systems might need to be manually activated by the user. In such cases, the security and confidentiality of data being stored on the mobile device must not be taken for granted. Configuration properties defined by the user can also influence the security of the keys that are used to encrypt data. For instance, if a user-defined passcode is incorporated into the key-derivation process, weak passcodes potentially compromise the security of the entire encryption system.

Beside the user, also developers potentially need to configure encryption systems correctly during the application-development process. For instance, developers might be responsible to correctly configure and use encryption features provided by the platform in their applications. This, in turn raises the risk that design or implementation errors made by the developer can negatively influence the security of an application. The developer's influence on the provided encryption systems is hence another relevant assessment criterion for the evaluation of encryption systems. In this context, also the ease of integration needs to be analyzed, as complex and difficult-to-integrate solutions increase the probability of implementation errors in third-party applications that rely on provided encryption systems.

## IV. Android Encryption Systems

Based on the assessment criteria defined above, Android's encryption systems are discussed and assessed in this section to derive potential attack scenarios. Android offers two primary encryption systems: **First** a file-system based encryption system that needs to be activated by the user/administrator, and **second** the Android KeyChain, which can be employed by the developer to store credentials used in an application in a secure way on the file-system. Apart from these encryption systems, the following analysis also discusses the various backup facilities on the Android platform, and cloud-storage components that directly or indirectly influence the security of data protected by the encryption systems. Although the backup system does not fall into the category of encryption systems, the protection mechanisms (or their lack) are crucial when deploying Android in security critical scenarios. An attacker who might not be able to break the device encryption system could still gain access to data by gaining access to backup files on a laptop or by breaking into the user's Google account.

### A. File-System Encryption – Dm-crypt

The primary encryption system on Android is a file-system based encryption system based on dm-crypt[3], which has been available in Linux kernels since Version 2.6.x. The Android file-system encryption has been introduced in Android 3.0, which was solely used on tablets. This version has later been adapted for smartphones and made available for those devices as Version 4.0. In Version 4.4, the employed KDF has been changed, which improves the security level of the used passcodes.

An overview of the encryption system is given in Figure 1. The user is required to enter the passcode as the first step in the Android boot process. The KDF is then used to derive a symmetric key from the user's passcode. This symmetric key protects the actual file encryption master key which is at the top of the key hierarchy required for the file-encryption system. The encryption system must be activated manually by the user, or be enforced by the corresponding MDM rule. The activation of the system requires the user to set an encryption passcode, which is also used for the phone's lock screen. When activating the encryption system, it is mandatory to activate the passcode screen lock functionality. Other lock-screens, such as the pattern lock functionality of the face-unlock system cannot be used when file-system encryption is activated.

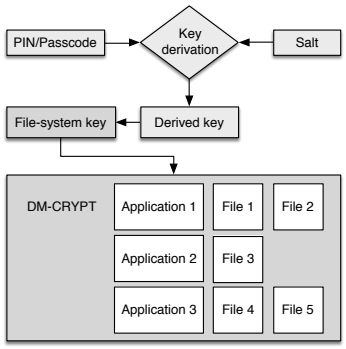


Fig. 1: Overview of the Android dm-crypt based encryption system. The whole file system is encrypted and the required file-system master key is protected by a key derived from the user's passcode.

*1) Assessment:* Regarding the first assessment criterion (*key derivation function*), a distinction between Android 3.0 to 4.3, and 4.4 must be made. While 3.0 to 4.3 use the Password-Based-Key-Derivation-Function 2 (PBKDF2) [9] with a SHA-1 based HMAC to derive a symmetric key from the user's passcode, Android 4.4. employs the SCRYPT function [10], which complicates parallel brute-force attacks by requiring computational and memory intensive resources.

**Android 3.0 to 4.3**: The PBKDF2 derivation process employs 2000 iterations to slow down possible brute-force attacks. The input values for the key derivation process are the user's passcode and a random salt value (16 bytes). The salt value is randomly generated during the activation of the encryption process and eliminates the possibility for an attacker to pre-calculate derived keys in order to speed-up the attack.

**Android 4.4**: The default parameters used for the SCRYPT function for the dm-crypt system are $N = 15$, $r = 3$, $p = 1$, where $N$ influences the required CPU resources, $r$ defines the memory requirements and $p$ influences the parallelization cost [10]. This setting can be overwritten during the Android build process by defining specific parameters in the `build.properties` file. Since these parameters can therefore be adapted by the different device manufacturers, this flexibility also results in different brute-force times and in

[3] http://code.google.com/p/cryptsetup/wiki/DMCrypt

different passcode properties that assure a secure encryption environment.

Regarding the second assessment criterion (*configuration capabilities*), it has to be stated that the Android file-system encryption is not activated per default, and must either be activated manually by the user, or be enforced by the respective MDM rule. Apart from the activation of the encryption system, the appropriate rules[4] for defining passcode properties must also be set by the administrator according to the security requirements of the envisaged deployment scenario and the times required to carry out a possible brute-force attack (discussed in Section IV-A2).

From a developer's point of view, it is not possible to influence the configuration of the encryption system. Due to the requirement to manually activate the system, the developer cannot assume that the Android system used to run the application is protected by the file-system encryption system. This is especially important for BYOD scenarios, where the security of the system depends on the settings chosen by the user and cannot be enforced by MDM.

*2) Attack Scenarios:* From the obtained assessment results, several attack scenarios can be derived for Android's file-based encryption system. The first attack scenario is based on the fact that Android's encryption system must be activated by the user manually or be enforced by the corresponding MDM rule. In addition, the security of the system depends on the strength of the passcode. Thus, the negligence to activate the system or to chose weak passcode properties allow for attacks on the encryption system when the device is stolen.

Another possible attack scenario targets the encryption system's KDF. The security of Android's file-system encryption system primarily depends on the length and the properties of the user's passcode used to derive the symmetric key, which protects the actual file-system encryption keys. Android uses a software-only architecture for encrypting the file-system. Thus, the system is susceptible to external brute-force attacks, where computationally expensive operations are outsourced to powerful computing units. For concrete numbers, we again need to distinguish between different Android versions.

**Android 3.0 to 4.3**: Since the KDF involves standard hash-algorithms, optimimized implementations can be leveraged to speed up the required calculations. In order to attach a price-tag to brute-force attacks carried out on Android, we have used the pricing model of the Amazon EC2 cloud computing[5] instances to calculate the costs for carrying out attacks on passcodes with different length and properties. Thereby, two scenarios have been evaluated: While the first scenario utilizes calculations based on standard CPUs available within Amazon cloud instances, the second scenario considers special Amazon instances that employ GPUs that speed up the hash calculation process. For the CPU scenario, obtained results of brute-force times are presented in Table I. In this table, only the prices for the on-demand instances are considered, which – in contrast to the reserved instances – do not require any upfront payments. The table shows the brute-force times for different passcode lengths and properties. Password complexities were arranged in the categories numerical, alphanumeric with lower case characters only and alphanumeric with lower and uppercase characters, and complex, which also includes symbols. The brute-force times are calculated for 1 instance and 1000 instances. Thereby, Amazon describes the performance of the instances by EC2 Compute Units (ECU)[6]. The price is calculated by using the per-hour price for the on-demand instances listed on the Amazon EC2 Web site (currently 0.06$ per ECU)[7]. The time to derive one key from a passcode is taken from the previously conducted measurements and is approximately 17.5 ms for executing the PBKDF2 function with 2000 iterations.

| Lock-Screen Type | Length | Chars | Number of passcodes | Brute-Force Days 1 instance | Brute-Force Days (1000 instances) | Cost $ On-Demand Instances |
|---|---|---|---|---|---|---|
| Numerical | 4 | 10 | 10000 | 0,1 | 0,0 | 0,1 |
| | 6 | 10 | 1000000 | 8,1 | 0,0 | 11,7 |
| | 8 | 10 | 100000000 | 810,2 | 0,8 | 1.166,7 |
| | 10 | 10 | 10000000000 | 81.018,5 | 81,0 | 116.666,7 |
| Alphanum 10/26 letters | 4 | 36 | 1679616 | 13,6 | 0,0 | 19,6 |
| | 6 | 36 | 2176782336 | 17.636,0 | 17,6 | 25.395,8 |
| | 7 | 36 | 78364164096 | 634.894,8 | 634,9 | 914.248,6 |
| | 8 | 36 | 2821109907456 | 22.856.214,5 | 22.856,2 | 32.912.948,9 |
| | 9 | 36 | 101559956668416 | 822.823.723,0 | 822.823,7 | 1.184.866.161,1 |
| | 10 | 36 | 3656158440062976 | 29.621.654.028,3 | 29.621.654,0 | 42.655.181.800,7 |
| Alphanum 10/52 letters | 4 | 62 | 14776336 | 119,7 | 0,1 | 172,4 |
| | 5 | 62 | 916132832 | 7.422,4 | 7,4 | 10.688,2 |
| | 6 | 62 | 56800235584 | 460.187,1 | 460,2 | 662.669,4 |
| | 7 | 62 | 3521614606208 | 28.531.599,8 | 28.531,6 | 41.085.503,7 |
| | 8 | 62 | 218340105584896 | 1.768.959.188,8 | 1.768.959,2 | 2.547.301.231,8 |
| | 9 | 62 | 13537086546624 | 109.675.469.703, | 109.675.469,7 | 157.932.676.373, |
| Complex | 4 | 107 | 131079601 | 1.062,0 | 1,1 | 1.529,3 |
| | 5 | 107 | 14025517307 | 113.632,7 | 113,6 | 163.631,0 |
| | 6 | 107 | 1500730351840 | 12.158.695,0 | 12.158,7 | 17.508.520,8 |
| | 7 | 107 | 160578147647 | 1.300.980.362,9 | 1.300.980,4 | 1.873.411.722,6 |
| | 8 | 107 | 17181861798336 | 139.204.898.829, | 139.204.898,8 | 200.455.054.313, |
| | | | | Android Amazon CPU | | CPU Price |

TABLE II: Passcode brute-force times for Android 4.4 SCRYPT key derivation function. A huge improvement in security is observed.

For the GPU scenario, the brute-force time calculation is based on a special Amazon instance that employs two NVIDIA Tesla Fermi M2050 GPUs. Although we did not implement a tool that carries out the brute-force attack on these GPUs, a good estimation on the required time can be provided by taking a closer look at the PBKDF2 function and the number of required hash operations and the block size of the input data. From the known number of required hash operations and known performance numbers of Amazon GPU instances, the results presented in the last three columns of TABLE I can be derived. The prices are adapted to the price of one hour instance time, which is currently at 2.1$ for an on-demand instance.

**Android 4.4**: A similar brute-force attack has also been implemented for the SCRYPT function. By using 1 ECU of the Amazon EC2 system, we have been able to calculate the times required to derive a key from a given passcode via the SCRYPT function. The results are presented in Table II. Due to the choice of the parameters, the time to derive one key is approximately 710 ms. This is a 40-fold increase when compared to the KDF used in Android 3.0 to 4.3. The level of security is significantly improved even for shorter passcodes as can be observed in Table II.

[4]http://developer.android.com/guide/topics/admin/device-admin.html
[5]http://aws.amazon.com/ec2/
[6]http://aws.amazon.com/ec2/faqs/
[7]In practice, it is possible to save money by changing to more powerful instances, for which prices do not grow linearly with the provided computing power. For the sake of simplicity and comparability, all prices have however been computed for the same instances.

| Lock-Screen Type | Length | Chars | Number of passcodes | Brute-Force Days | Brute-Force Days 1 instance | Brute-Force Days (1000 instances) | Cost $ On-Demand Instances | Brute-Force Days 1 instance | Brute-Force Days (1000 instances) | Cost $ On-Demand Instances |
|---|---|---|---|---|---|---|---|---|---|---|
| Numerical | 4 | 10 | 10000 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| | 6 | 10 | 1000000 | 0.9 | 0.2 | 0.0 | 0.3 | 0.0 | 0.0 | 0.0 |
| | 8 | 10 | 100000000 | 92.6 | 20.3 | 0.0 | 29.2 | 0.0 | 0.0 | 1.3 |
| | 10 | 10 | 10000000000 | 9,259.3 | 2,025.5 | 2.0 | 2,916.7 | 2.6 | 0.0 | 133.3 |
| Alphanum | 4 | 36 | 1679616 | 1.6 | 0.3 | 0.0 | 0.5 | 0.0 | 0.0 | 0.0 |
| 10/26 letters | 6 | 36 | 2176782336 | 2,015.5 | 440.9 | 0.4 | 634.9 | 0.6 | 0.0 | 29.0 |
| | 7 | 36 | 78364164096 | 72,559.4 | 15,872.4 | 15.9 | 22,856.2 | 20.7 | 0.0 | 1,044.9 |
| | 8 | 36 | 2.82111E+12 | 2,612,138.8 | 571,405.4 | 571.4 | 822,823.7 | 746.3 | 0.7 | 37,614.8 |
| | 9 | 36 | 1.0156E+14 | 94,036,996.9 | 20,570,593.1 | 20,570.6 | 29,621,654.0 | 26,867.7 | 26.9 | 1,354,132.8 |
| | 10 | 36 | 3.65616E+15 | 3,385,331,888.9 | 740,541,350.7 | 740,541.4 | 1,066,379,545.0 | 967,237.7 | 967.2 | 48,748,779.2 |
| Alphanum | 4 | 62 | 14776336 | 13.7 | 3.0 | 0.0 | 4.3 | 0.0 | 0.0 | 0.2 |
| | 5 | 62 | 916132832 | 848.3 | 185.6 | 0.2 | 267.2 | 0.2 | 0.0 | 12.2 |
| 10/52 letters | 6 | 62 | 56800235584 | 52,592.8 | 11,504.7 | 11.5 | 16,566.7 | 15.0 | 0.0 | 757.3 |
| | 7 | 62 | 3.52161E+12 | 3,260,754.3 | 713,290.0 | 713.3 | 1,027,137.6 | 931.6 | 0.9 | 46,954.9 |
| | 8 | 62 | 2.1834E+14 | 202,166,764.4 | 44,223,979.7 | 44,224.0 | 63,682,530.8 | 57,761.9 | 57.8 | 2,911,201.4 |
| | 9 | 62 | 1.35371E+16 | 12,534,339,394.7 | 2,741,886,742.6 | 2,741,886.7 | 3,948,316,909.3 | 3,581,239.8 | 3,581.2 | 180,494,487.3 |
| Complex | 4 | 107 | 131079601 | 121.4 | 26.5 | 0.0 | 38.2 | 0.0 | 0.0 | 1.7 |
| | 5 | 107 | 14025517307 | 12,986.6 | 2,840.8 | 2.8 | 4,090.8 | 3.7 | 0.0 | 187.0 |
| | 6 | 107 | 1.50073E+12 | 1,389,565.1 | 303,967.4 | 304.0 | 437,713.0 | 397.0 | 0.4 | 20,009.7 |
| | 7 | 107 | 1.60578E+14 | 148,683,470.0 | 32,524,509.1 | 32,524.5 | 46,835,293.1 | 42,481.0 | 42.5 | 2,141,042.0 |
| | 8 | 107 | 1.71819E+16 | 15,909,131,294.7 | 3,480,122,470.7 | 3,480,122.5 | 5,011,376,357.8 | 4,545,466.1 | 4,545.5 | 229,091,490.6 |
| | | | | iOS on device | Android Amazon CPU | | CPU Price | Android Amazon GPU | | GPU Price |

TABLE I: Passcode brute-force times and costs: The passcode properties and number of possible combinations are listed in columns 1-4. The iOS brute-force times are listed in the fifth column and are followed by the Android brute-force times and the price for CPU and GPU instances (1 and 1000 instances).

### B. *Android KeyChain*

The Android KeyChain is the second encryption system offered by the Android operating system. It can be used even when the primary file-system encryption system is not activated. Android has implemented a low level credential store since Version 1.6. This credential store is a system daemon, providing its capabilities via a Unix socket interface. Until Version 4.0, only system applications were able to use the credential store for WiFi passwords or VPN credentials. In Version 4.0, a new KeyChain API[8] was introduced, which allows applications to use the credential store to store private key material and certificate chains. The KeyChain API was extended with hardware support in Android 4.1[9], which offers better protection for the stored keys, because they cannot be extracted from the device. In Android 4.3, another improvement has been included: The KeyChain API can now be used via the Java Cryptographic Extension (JCE) API, which simplifies the usage of asymmetric keys in Android applications. For the security analysis of the KeyChain, we need to consider the differences between software-only and hardware-based realizations.

The implementation [11] of the keystore API reveals details of the software- and hardware-based versions. For software-based keystores, the user's passcode is utilized to derive an AES 128 bit key (16 bytes), which is then used to encrypt a master key. When the credential store is unlocked, the plain master key is stored in the memory of the credential store. This master key is then used to encrypt and decrypt the stored secret values. These values are stored by application id and alias name as simple encrypted file. When hardware support is available, the detailed protection mechanisms depend on the specific implementation. However, in general the keys are stored on the file system and are encrypted with a master secret, which is only available within the trusted environment offered by the hardware element. When the keys are used for a cryptographic operation, they are fetched from the file system and utilized by the specific operation (e.g., signing), which is executed within the secure environment of the hardware element. In addition, the passcode of the user is also used for providing an additional protection layer. The key material is also encrypted with the 128 bit AES master key, which is itself protected by the key derived from the passcode.

*1) Assesssment:* Regardless of software-only, or hardware-based protection, the PBKDF2 KDF with HMAC SHA1 is used to derive the AES key from the passcode. The derivation process employs 8192 iterations. The master key file also includes an MD5 digest of the unencrypted master key. This digest allows to verify whether the derived key and thus the provided passcode is correct. If a software-only solution is used, the decrypted master key is used to decrypt the keys that are stored encrypted on the file system. In case of additional hardware support, the key material is also decrypted with the master key. However, due to the additional encryption layer provided by the key within the hardware element, the key material cannot be extracted and used off-device.

Regarding the assessment criterion *configuration capabilities*, the user/administrator influences the security of the KeyChain system (when used by an application) by choosing the passcode for locking the Android device. Similar to the file-system encryption system, the properties of this passcode (length, complexity) can be defined by selecting appropriate MDM rules. Also the developer is able to influence the security of this encryption system by deciding for own applications whether the KeyChain is used for storing the user's credentials.

*2) Attack Scenarios:* Based on the obtained assessment results, the following attack scenarios can be derived. Even though the KeyChain cannot be directly configured, its security is strongly influenced by the passcode chosen by the user. The

[8]http://developer.android.com/reference/android/security/KeyChain.html
[9]http://developer.android.com/about/versions/jelly-bean.html

strength of this passcode is determined by the requirements set by the deployed MDM-system, or in a BYOD scenario by the user's choice. Weak MDM rules or user choices hence represent an attack scenario. The developer needs to chose whether the KeyChain is used for storing the credentials. Poor developer choices (e.g., storing the credentials unencrypted on the file system) can lead to critical security issues and hence also represent an attack scenario.

Other attack scenarios target the KeyChain's KDF. Basically, the KeyChain system is susceptible to the same brute-force attack as described in Section IV-A2. However, certain aspects need to be considered. The KDF uses 8192 iterations (compared to 2000 PBKDF2 iterations for the 4.0 to 4.3 dm-crypt system). However, in contrast to the dm-crypt system, which generates 32 bytes (16 bytes key material, 16 bytes initialization vector), only 16 bytes are generated by the KeyChain derivation function. By considering these two differences it takes approximately the double amount of time to derive one key, in comparison to the KDF employed in the file-system encryption system. Thus, the brute-force times and prices presented in Table I need to be multiplied by two in order to get the appropriate values for the KeyChain. Thus, for keystores with hardware support and software support, the passcodes can be brute forced by using external ressources. On software-based systems, the keys within the keystore can then be extracted. While this is not possible for systems with hardware support, the keys can still be used for crypto operations on the device.

When the dm-crypt system is used to encrypt the file system, the attacker cannot directly extract KeyChain related files to carry out brute-force attacks. The attacker first needs to apply the brute-force attack on the dm-crypt system. The complexity of such an attack depends on the employed KDF (PBKDF2 vs. SCRYPT). However, the KeyChain uses the same passcode as the dm-crypt system and the screen-lock function. This eliminates the need to carry out a second brute-force attack, when the passcode for the dm-crypt system has already been determined. Thus, the brute-force attack on the KeyChain is only relevant for systems that are not encrypted with the dm-crypt system.

### C. Backup

Backup systems are not directly related to the encryption systems offered by the Android system. Still, backup mechanisms must be considered for security-critical deployment scenarios as an attacker might not need to circumvent the encryption systems, but directly access the data contained in non-protected backups.

The Android system offers two primary backup systems that are integrated within the Android operating system: **First,** the Android Backup Service, which allows application developers to implement backup facilities within their applications and store data on Google servers, and **second,** the ADB Backup component, which uses the Android Debug Bridge (ADB) to create complete system backups. As our analysis focuses on managed devices, it can be assumed that developer mode, which is a prerequisite for the ADB backup, is not activated. In managed scenarios, developer mode can either be deactivated through appropriate MDM rules or prevented by organizational means. Hence, we regard the ADB backup mechanism to be out of scope and solely focus on the Android Backup Service for this analysis.

Apart from the two standard backup systems mentioned above, device manufactures might implement their own backup services or extend the existing services according to their needs. Unfortunately, a generic security analysis for proprietary backup systems cannot be provided due to the freedom to implement such as a service according to the device manufacturer's needs. Thus, a specific security analysis must be conducted for the selected platform's backup solution.

The remaining backup mechanism, i.e. the Android Backup Service will be analyzed in the following in more detail. The Android Backup Service has been introduced with Version 2.2 of the Android operating system and provides a backup API that allows applications to backup and restore their data to and from the cloud[10]. The service itself is not intended for data synchronization between multiple devices but for restoring application data in case of a factory reset, or if the user switches to a different Android device. Apart form the application backup related functionality, the activation of the system also causes the Android system to backup WiFi passwords in plain text[11].

The backup system consists of two main parts: The *Backup Manager* and the *Backup Agent*. The latter is application-specific and has to be implemented by the application developer. It defines which data should be saved and performs the actual backup and restore operations for the given application. The *Backup Manager* acts system-wide and schedules all backup and restore operations for all applications by calling the corresponding *Backup Agents*. Then, the resulting data is transported to the cloud storage. It is not mandatory for a device manufacturer to implement the Android backup service. If the system is present, its features and security depend on the implementation of the device manufacturer. However, it is assured that the backup data cannot be accessed by other installed applications (assuming the device is not rooted).

*1) Assessment:* Regarding the assessment criterion *key derivation function*, it can be stated that the default system does not use an encryption system, and thus does not include a KDF. The backup system is only protected with the user's Google account and the associated passcode.

The backup system provides several configuration capabilities. The backup system can be activated by the user (located within the Android privacy settings). However, there is no rule for restricting this behavior within the standard MDM sub-system of Android. For most Android devices, Google provides a backup transport, the Android Backup Service which is available for most devices that support the Google Play Store[12]. In order to use this service (if it is available on the given device), the application has to be registered for Android Backup Service and the Backup Service Key must be added to the Android manifest via a *meta-data* tag. Backed up data is treated according to Google's privacy policy[13].

[10] http://developer.android.com/guide/topics/data/backup.html

[11] http://arstechnica.com/security/2013/07/does-nsa-know-your-wifi-password-android-backups-may-give-it-to-them/

[12] https://developer.android.com/google/backup/index.html

[13] http://www.google.com/policies/privacy/

*2) Attack Scenarios:* There are several security issues that represent strong reasons to deactivate the backup service, in security-critical deployment scenarios. **First,** if not otherwise implemented, the data is stored in plain text in the cloud. This is especially relevant for WiFi passwords that are stored in plain text in the cloud as soon as the backup system is activated. The security of the data depends on the security of the Google account passcode, which cannot be influenced by MDM systems. **Second,** the standard Google Android MDM rule-set does not include any rules that allow to configure the backup system. However, such rules are present in certain proprietary MDM extensions (e.g., Samsung SAFE). **Third,** the developer decides whether application data can be backed up with the backup system. A non-security-aware developer might chose to use the backup system without being aware of the consequences related to security.

## V. Security Analysis - Workflow

Based on the attack scenarios derived in the previous section, we propose a workflow that assists security officers of companies or public agencies in deploying Android devices in security-critical environments. In particular, following the proposed workflow allows security officers to systematically identify potential threats and helps them to find appropriate device configurations.

The presented workflow has been tailored to the general scenario defined in Section II and has been designed under the assumption that a thorough risk analysis within the context of the deployment scenario has already been conducted so that critical assets, threats, and risk factors are already known. Based on the risk analysis and the existing policies, the Android platform needs to be configured properly and the appropriate applications for handling the critical data need to be chosen. After selecting those applications and setting up a demonstration environment that already contains data to be protected, the workflow shown in Figure 2 can be applied.

In the following subsections, major analysis steps of the proposed workflow are introduced and discussed in detail. For each analysis step, implications of possible analysis results are summarized, which need to be considered by the security officer in charge. Potential threats (and their risk potential), which arise from possible implications are also presented in Figure 2 and discussed below.

### A. File-System Encryption Supported

Verification of support for file-system encryption represents the first analysis step of the entire workflow. The file-system based encryption system is supported since Android 3.x, which was only available on tablets. For smartphones, Android 4.0 was the first version to support the dm-crypt based encryption system. Unfortunately, Version 4.x does not necessarily indicate that encryption is supported by the device. There are also certain 4.0 devices, which do not support encryption. Hence, encryption support must be verified for each deployed device.

**Implications for unencrypted systems**: *No data encryption*: An attacker who gains access to the device can gain access to all the data stored on the device that is not encrypted by other means. Alternative means include for instance application-specific encryption systems, where an application implements the encryption and key-derivation process on its own. This scenario plays an important role for container applications, which need to provide their own security mechanisms due to their utilization in Bring-Your-Own-Device scenarios where no, or only limited assumptions can be made about the security level of the device. Even if file-system encryption is not supported by the given device, key material that is stored in the Android KeyChain is still encrypted via the mechanism described in Section IV-B.

### B. File-System Encryption Enabled

The file-system encryption is not enabled per default. Thus, the system must either be enabled by the user or enforced by setting the corresponding MDM rule. A security officer must assure that available file-system encryption systems are enabled.

**Implications for encrypted systems**: *Off-device brute-force attack*: As soon as the encryption system is enabled, a passcode must be chosen by the user. Since this passcode is used for both the lock screen and the file-system encryption system, a good balance between usability (short passcodes for fast unlocking) and security (long passcodes) must be found. The rules for selecting the passcode complexity must be defined within the MDM system according to the security level of the envisaged deployment scenario. Brute-force times presented in Table I and in Table II assist security officers in choosing appropriate passcode complexities.

**Implication for unencrypted systems**: *No data encryption*: The same implications as stated in the previous section apply.

### C. Android KeyChain Usage

Even if no files-system encryption is available, credentials can still be used by the Android KeyChain, if this is supported by the respective application. Depending on whether the KeyChain is used or not, different implications need to be considered by the security officer.

**Implications for key material protected by the KeyChain**: *Off-device brute-force attack*: Except for the number of iterations, the KeyChain is based on the same PBKDF2 function that is used for the file-system encryption system in Android 3.0 to 4.3. The passcode selected for the KeyChain is also used as passcode for unlocking the device, and for encrypting the file-system (if activated). This implies that the same considerations between security and usability must be taken, and the brute force related issues described in Section IV-B2 need to be considered. If the KeyChain also uses additional hardware support to protect keys, a successful brute-force attack allows using the keys only on the specific device. If a software-only keystore is used, the keys can also be extracted. If the Android system is already encrypted via the file-system encryption system, there is no additional protection offered by the Android KeyChain. If the passcode for the Android encryption system has been found via a brute-force attack, the same passcode can be used to derive the keys used by the KeyChain encryption system.

**Implications for other key material**: If the Android KeyChain is not used by an application that stores credentials,

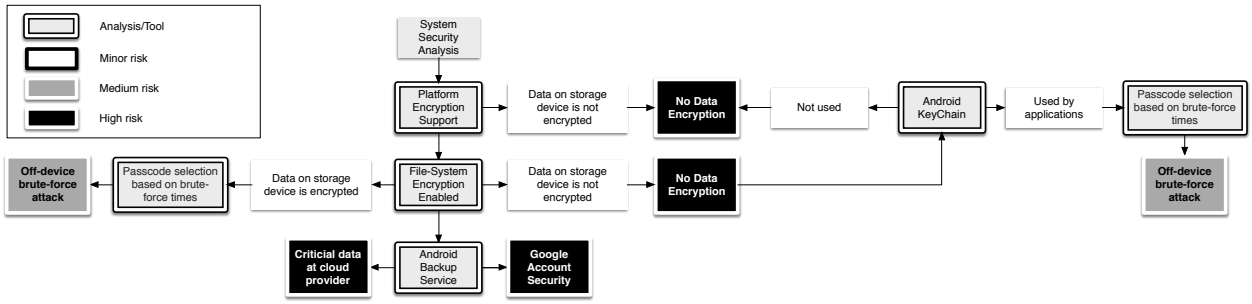


Fig. 2: The black, light-grey and white cells indicate critical, medium, and minor issues.

the developer could still provide an application-integrated system that provides its own KDF and encryption scheme. If such a scheme is not used and the credentials are stored unprotected on the device, then their security depends on the activation state of the file-encryption system.

### *D. Backup Usage*

The conducted assessment has shown that Android's backup mechanism bears the risk to undermine the security provided by enforced encryption mechanisms. The integrated backup facility does not provide encryption when storing data on Google's servers. Furthermore, credentials of the Android operating system (e.g. WiFi passcodes etc.) are stored in plain text at the cloud provider. Apart from these problems, the security of the backup depends on the strength of the passcode chosen by the user for the Google account (**Google Account Security**). Therefore, it is not recommended to use this facility in security-critical scenarios. In any case, the security officer has to be aware of usage of backup mechanisms by applications the process and store security-critical data.

## VI. Conclusions

Deploying Android in security-critical environments is a complex task, as confidential data might get compromised when being accessed, processed, and stored by insecure mobile devices. To facilitate this task, we have systematically analyzed and assessed different encryption systems of the Android platform, which provide the opportunity to protect security-critical and confidential data. From the obtained assessment results, potential attack scenarios have been derived. Finally, a workflow has been proposed, which assists in deploying and configuring Android devices in security-critical environments and applications.

Obtained assessment results have also shown one of the main challenges of the Android platform: In contrast to other mobile platforms such as iOS, BlackBerry, or Windows Phone, Android features many more different versions and hence shows a much higher fragmentation. This heterogeneity is mainly caused by device manufactures, who supply their devices with customized versions of the Android OS. Due to this heterogeneity, the deployment of Android devices in security-critical scenarios is a challenging task that requires an in-depth security analysis of the envisaged platform. The main difficulties are the various sub-systems that depend on the specific device manufacturer's implementation, the lack of MDM rules/restrictions to configure relevant aspects of the Android system, and the differences in the encryption systems.

Due to the given heterogeneity, the proposed workflow has been defined on a rather abstract basis and does not consider manufacturer-dependent features or limitations. Still, the proposed workflow – and also the results obtained from the conducted assessments – represents a useful basis that facilitates a correct use of Android's encryption systems in security-critical applications. Manufacturer- or version-dependent refinements of the proposed workflow are regarded as future work.

## References


[1] M. Paul, N. S. Chauhan, and A. Saxena, "A security analysis of smartphone data flow and feasible solutions for lawful interception," pp. 19–24, 2011. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6122788

[2] L. S. L. Shurui, L. J. L. Jie, Z. R. Z. Ru, and W. C. W. Cong, "A Modified AES Algorithm for the Platform of Smartphone," pp. 749–752, 2010. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5636941

[3] Y. C. Y. Chen and W.-S. K. W.-S. Ku, "Self-Encryption Scheme for Data Security in Mobile Devices," pp. 1–5, 2009. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4784733

[4] T. Müller and M. Spreitzenbarth, "FROST - Forensic Recovery of Scrambled Telephones," in *ACNS*, 2013, pp. 373–388.

[5] V. R. Pandya, "IPHONE SECURITY ANALYSIS," *Journal of Information Security*, vol. 1, no. May, pp. 74–87, 2008. [Online]. Available: http://www.scirp.org/journal/PaperDownload.aspx?DOI=10.4236/jis.2010.12009

[6] J.-B. Bedrune and J. Sigwald, "iPhone data protection in depth," Sogeti / ESEC, Tech. Rep., 2011. [Online]. Available: http://code.google.com/p/iphone-dataprotection/

[7] P. Teufl, T. Zefferer, C. Stromberger, and C. Hechenblaikner, "iOS Encryption Systems: Deploying iOS Devices in Security-Critical Environments," in *Proceedings of the 10th International Conference on Security and Cryptography - SECRYPT 2013*, Reykjavik, 2013.

[8] M. S. Turan, E. Barker, W. Burr, and L. Chen, "Recommendation for Password-Based Key Derivation Part 1 : Storage Applications," *Nist Special Publication*, no. December, p. 14, 2010. [Online]. Available: http://csrc.nist.gov/publications/nistpubs/800-132/nist-sp800-132.pdf

[9] B. Kaliski, "PKCS \#5: Password-Based Cryptography Specification Version 2.0," 2000. [Online]. Available: http://www.ietf.org/rfc/rfc2898.txt

[10] C. Percival, "Stronger key derivation via sequential memory-hard functions," *Self-published*, pp. 1–16, 2009. [Online]. Available: http://www.bsdcan.org/2009/schedule/attachments/87_scrypt.pdf$\delimiter"026E30F$nhttp://www.unixhowto.de/docs/87_scrypt.pdf

[11] Android, "KeyStore Implementation." [Online]. Available: https://android.googlesource.com/platform/system/security/+/master/keystore/keystore.cpp