Android full-disk encryption: a security assessment
Oliver Kunz

Technical Report
RHUL–ISG–2016–8
5 April 2016

Information Security Group
Royal Holloway University of London
Egham, Surrey, TW20 0EX
United Kingdom
Student Number: 100805788
Oliver Kunz

Android Full-Disk Encryption
A Security Assessment

Supervisor: Keith Martin

Submitted as part of the requirements for the award of the
MSc in Information Security
at Royal Holloway, University of London

I declare that this assignment is all my own work and that I have acknowledged all quotations from published or unpublished work of other people. I also declare that I have read the statements on plagiarism in Section 1 of the Regulations Governing Examination and Assessment Offences, and in accordance with these regulations I submit this project report as my own work.

Signed:......................................................(Oliver Kunz)

Date:.....................................................................
Acknowledgements

I would like to thank my supervisor Keith Martin for his support and guidance throughout this thesis.

Special thanks go to everyone who supported me during the studies of this MSc, especially my parents and Isabelle.
# Contents

List of Figures iv
List of Tables v
List of Abbreviations vi
Executive Summary ix

1 Introduction 1
   1.1 Motivation ................................................................. 1
   1.2 Objectives ................................................................. 2
   1.3 Structure of the Report ................................................. 2

2 Background 3
   2.1 Android Operating System ............................................. 3
      2.1.1 Openness ............................................................. 3
      2.1.2 Fragmentation ....................................................... 3
      2.1.3 Nexus and Stock Android ........................................ 5
   2.2 Encryption ................................................................. 5
      2.2.1 Cipher Block Chaining ............................................ 5
      2.2.2 XEX Encryption Mode with Tweak and Ciphertext Stealing .... 7
      2.2.3 Device-Mapper and dm-crypt .................................... 8
   2.3 Password Stretching ...................................................... 9
      2.3.1 Password-Based Key Derivation Function 2 .................... 9
      2.3.2 scrypt .................................................................. 10
   2.4 Chapter Summary ........................................................... 11

3 Android Full-Disk Encryption 12
   3.1 Compatibility Issues ..................................................... 12
   3.2 Implementation in Android 5.0 ......................................... 13
      3.2.1 Key Management ....................................................... 13
      3.2.2 Encrypting on First Boot ......................................... 16
      3.2.3 Decryption on Subsequent Boot .................................. 17
      3.2.4 Comparison with Android 4.4.4 ................................ 17
   3.3 Threat Model ................................................................. 19
      3.3.1 Threat Modelling Questionnaire ................................ 19
      3.3.2 Security Services Not Provided .................................. 20
4 Security Assessment

4.1 Attack Scenarios

4.2 Existing Generic Countermeasures

4.2.1 Locked Bootloader

4.2.2 Disabled Android Debug Bridge

4.2.3 Android Device Manager

4.3 Attacks on Cryptography

4.3.1 Watermarking and Movable Cipher Block Attacks

4.3.2 Malleability Attack

4.3.3 Content Leak Attack

4.4 Attacks on the Implementation

4.4.1 Online Exhaustive Password Search Attack

4.4.2 Offline Exhaustive Password Search Attack

4.4.3 Semi-Offline Exhaustive Password Search Attack

4.4.4 Static Default Password and Master Key

4.4.5 Cold-Boot Attack

4.4.6 Evil-Maid Attack

4.4.7 Potential Weaknesses

4.5 Bypass Lock-Screen by Attacking Smart Lock

4.5.1 Trusted Places

4.5.2 Trusted Devices

4.5.3 Facial Recognition

4.5.4 Accelerometer

4.5.5 Countermeasures

4.5.6 Attack Surface

4.6 Concluding Remarks

5 Conclusion

5.1 Our Contribution

5.2 Outlook on Android’s Full-Disk Encryption

5.3 Future Work

References

Appendices

A Source Code

B Partition Imaging

C Device Specifications
List of Figures

2.1 XTS Ciphertext Stealing, based on [27, Fig. 2] ................................. 7
2.2 XTS Encryption Process, based on [27, Fig. 3] ................................. 8
3.1 Key Hierarchy ............................................................................. 14
3.2 Key Encryption Key and Master Key Generation ............................. 15
3.3 Confirmation Screen ................................................................... 18
4.1 Client-Server Communication ....................................................... 34
4.2 Visualization of Data Decay in Memory [29, p. 50] ......................... 39
4.3 Time Dependent Trusted Place Extension ...................................... 46
4.4 Face Unlock Test Scenario Three ................................................ 49
List of Tables

2.1 Android OS Version Distribution [19] ............................................. 4
4.1 Attack Surface: Overview ............................................................. 24
4.2 Attack Surface: Malleability Attack .............................................. 26
4.3 Attack Surface: Content Leak Attack ............................................ 27
4.4 Average PIN Input Duration per Attempt ...................................... 29
4.5 Nexus 6 Online Exhaustive Password Search Attack ...................... 29
4.6 Countermeasure Identification of Online Exhaustive Password Search Attack 30
4.7 Attack Surface: Online Exhaustive Password Search Attack .......... 31
4.8 Offline Exhaustive Password Search Attack .................................. 32
4.9 Attack Surface: Offline Exhaustive Password Search Attack .......... 33
4.10 Nexus 6 Semi-Offline Exhaustive Password Search Attack ............ 35
4.11 Time Comparison of Different Exhaustive Password Search Attack Methods 35
4.12 Attack Surface: Semi-Offline Exhaustive Password Search Attack ...... 36
4.13 Attack Surface: Static Default Password and Master Key ................ 38
4.14 Attack Surface: Cold-Boot Attack ................................................. 40
4.15 Attack Surface: Evil-Maid Attack ................................................ 41
4.16 Potential Attack Surface: Predefined Cryptographic Primitives ...... 42
4.17 Potential Attack Surface: Lack of Entropy in Master Key ............... 43
4.18 Potential Attack Surface: Unencrypted Data Residue on Flash-Memory .. 44
4.19 Potential Attack Surface: Trusted Execution Environment Vulnerabilities . 44
4.20 Attack Surface: Smart Lock Misuse ............................................. 51

C.1 Attack Host Specifications .......................................................... 66
C.2 Nexus 6 Specifications ............................................................... 67
C.3 Nexus 4 Specifications ............................................................... 67
List of Abbreviations

adb  Android Debug Bridge
ADM  Android Device Manager
AES  Advanced Encryption Standard
AOSP Android Open-Source Project
API  Application Programming Interface
ASCII American Standard Code for Information Interchange
ARM  Advanced RISC Machines (company name and CPU architecture)
BYOD Bring Your Own Device
CBC  Cipher Block Chaining
CDD  Compatibility Definition Document
CMC  CBC-Mask-CBC
CPU  Central Processing Unit
DEK  Disk Encryption Key
dm   Device-Mapper
dm-crypt Device-Mapper Module for Linux Kernel Crypto API
DRM  Digital Rights Management
ECB  Electronic Code Book
EME  Encrypt-Mix-Encrypt
eMMIC embedded MultiMedia Card
ESSIV Encrypted Salt-Sector Initialization Vector
F2FS Flash-Friendly File System (file system format)
FAT  File Allocation Table (file system format)
FDE  Full-Disk Encryption
**FROST** Forensic Recovery of Scrambled Telephones

**FTL** File Translation Layer

**Git** An open-source distributed revision control system

**GPL** GNU Public License

**GPS** Global Positioning System

**HMAC** Hash-based Message Authentication Code

**IEEE** Institute of Electrical and Electronics Engineers

**iOS** Operating System for Apple mobile devices (e.g. iPhone)

**IV** Initialization Vector

**JTAG** Joint Test Action Group (debug and test interface for boards)

**KDF** Key Derivation Function

**KEK** Key Encryption Key

**LRW** Liskov-Rivest-Wagner (block cipher mode)

**LUKS** Linux Unified Key Setup

**MAC** Message Authentication Code

**nc** netcat

**NFC** Near Field Communication

**NIST** National Institute of Standards and Technology

**OS** Operating System

**OTG** On-The-Go (allows USB devices to act as host)

**PBKDF2** Password-Based Key Derivation Function 2

**PKCS** Public-Key Cryptography Standards

**PIN** Personal Identification Number

**PoC** Proof-of-Concept

**PRF** Pseudo-Random Function

**PRNG** Pseudo-Random Number Generator

**PXE** Preboot Execution Environment

**QSEE** Qualcomm Secure Execution Environment

**RSA** Rivest-Shamir-Adleman (Public-Key Cryptosystem)
SD card  Secure Digital card (non-volatile memory card)

SDK  Software Development Kit

SHA  Secure Hash Algorithm

SIM  Subscriber Identity Module

SoC  System on a Chip

SSH  Secure Shell

TEE  Trusted Execution Environment

tmpfs  temporary file system

UID  user identification

USB  Universal Serial Bus

vold  Volume Daemon

Wi-Fi  Wireless Fidelity Synonym for wireless network

XEX  XOR-Encrypt-XOR

XTS  XEX encryption mode with Tweak and ciphertext Stealing
Executive Summary

Mobile phones evolved from basic telecommunication devices to smartphones which are, in essence, pocket computers. With this technological evolution their usage also changed. Nowadays users do not just keep contact details and text messages but also e-mails, chat communications, documents, browsing history and other data stored on their mobiles.

Different actors are interested in this data: criminals, competitors, as well as law enforcers. In 2014, Google announced that it would enable Android's full-disk encryption by default. Encryption is a dual-used good, since it protects data from invasive third parties but at the same time there are situations where law-supported third parties need access to that data. Law enforcers face increasing difficulties in collecting evidence for the prosecution of criminals.

This project aims to understand how Android's full-disk encryption feature is implemented in the broader context of the framework and to assess the security of this feature. We analyse the source code which provides the encryption functionality, manages the encryption keys and triggers decryption on device boot. We researched known vulnerabilities to cryptographic primitives employed as well as to similar full-disk encryption implementations in Linux and Android.

According to Google, Android 5.0 was improved to prevent Offline Exhaustive Password Search Attacks to recover the screen lock method which is also used to protect the master key for disk encryption. In our research we confirm this statement while presenting an alternative attack approach we call Semi-Offline Exhaustive Password Search Attack. In contrast to the Offline approach, the smartphone is used for a particular step in the attack, hence the name Semi-Offline. Our attack takes five times longer than the Offline attack but is more than ten times faster than an Online Exhaustive Password Search Attack.

The threat model covered by Android, as we identified, only protects data at rest. This requires the smartphone to be shut down, a rather rare state for mobile phones. We therefore included in our assessment the attack scenario Device-ON. With transparent encryption, each disk read is decrypted and each write encrypted. An attacker who manages to bypass lock-screen authentication therefore has full access to the otherwise encrypted data.

We assessed the misuse potential of Smart Lock, an authorized lock-screen bypass feature introduced in Android 5.0, based on configurable trust agents. A trust agent can for example be a location, device, face or body movement. We were able to demonstrate in three of the four categories how an adversary can misuse the feature and described a potential scenario for the last.

For various attacks and vulnerabilities, we proposed new countermeasures or improvements to existing ones.
Chapter 1

Introduction

1.1 Motivation

Android Full-Disk Encryption (FDE) was introduced in Android 4.0, which was released in 2011. Enabling the feature must be done manually by the user and it is likely that the majority of users did not follow that procedure and used the smartphone unencrypted. In 2013, news coverage of the technical capabilities of intelligence agencies like the NSA or GCHQ [28] raised public discussion about data privacy and information security. As a consequence of this discussion and news articles, technology companies such as Apple, Google or Cisco were accused of co-operating with intelligence agencies to collect and observe data from users around the world.

As an aftermath, Google announced the expanded use of encryption to protect customers’ data. Part of this initiative was the announcement about enabling FDE in Android 5.0 by default on all devices shipped with that version [59]. This move by Google, as well as Apple’s encryption, caused a reaction by law enforcement agencies. In a public speech [6], FBI Director James B. Comey expressed his concern about preventing access to information needed for prosecution and argued that not only good people are protected by these default encryption settings, but also criminals. We discussed [60] this issue with Stephan Walder, Head of Zurich’s Cybercrime Competence Center. He stated that collecting evidence becomes more difficult with encryption. Criminals and especially white-collar criminals are keen on leaving no paper trail and often digitalize the information needed for their crimes. Nowadays, these data is often held on smartphones. By having default encryption activated, Walder expects to face greater difficulties in collecting evidence, even when a service provider is cooperative.

Only a few months after the announcement about enabling FDE by default, journalists reported that Google revised its decision [7]. It remains unclear whether this is true or not. The security enhancement list for Android [21] still lists default encryption for “devices that ship L (Android 5.0) out-of-the-box”, while the Compatibility Definition Document (CDD) [18] does not list FDE enabled by default as a binding criterion.

Regardless, the discussion about default encryption and the improvements in Android’s implementation [21] were the core motives to devote this thesis to analysing that feature and assessing the security of FDE both in isolation and in a wider context in an Android smartphone.
1.2 Objectives

The main objectives of this thesis are:

- Identifying the key technologies employed with the Android FDE
- Understanding and describing the implementation of Android FDE
- Identifying the threat model covered with this implementation
- Conducting a security assessment on the implementation of FDE
- Understanding and describing known attacks against Android FDE
- Proposing enhancements to existing countermeasures

1.3 Structure of the Report

This report is structured as follows: Chapter 2 introduces background information on Android, the cryptographic primitives for encryption and password stretching, as well as the Kernel module used to implement FDE.

In Chapter 3 we analyse and describe how the feature is implemented in the Android source code, what compatibility issues might arise and how the default encryption process works. Additionally, we discuss the subsequent decryption process, compare the enhanced implementation of Android 5.0 with the previous version Android 4.4.4, and outline the threat model we analysed.

Chapter 4 contains our security assessment, in which we introduce known attacks and an attack we developed against FDE. We further describe vulnerabilities which do not directly lead to an attack but could potentially weaken the cryptosystem as it is. The screen lock authentication method in Android is employed in the FDE process. With Android 5.0, a bypassing feature is introduced, which we have also analysed.

This report ends with our conclusion in Chapter 5, summarizing our contributions and results, as well as providing suggestions for further research in this area.
Chapter 2

Background

2.1 Android Operating System

Android is in many ways different from the Operating System (OS) of other mobile platforms, such as Apple’s iOS. Two major differences are:

1. Openness  —  Android is an open-source project
2. Fragmentation  —  Android is fragmented in terms of OS implementation and hardware platform

2.1.1 Openness

In the Android Open-Source Project (AOSP), Google is the main editor and most source code written is published under the Apache Software License Version 2 or the GNU Public License (GPL). However, some parts of the source code are not open-source, such as Digital Rights Management (DRM) and radio link implementations [11].

In addition, Android builds on top of a customized Linux kernel which is mainly released under GPL. Power-users and application developers exploring the device and its OS in detail will certainly recognize this quickly when accessing an Android smartphone via the Android Debug Bridge (adb) by starting a remote shell. This shell is similar to a Linux terminal, offering a subset of command line tools also present in Linux and using the same file permissions. This leads to the conclusion that features in Android can be related to those of a Linux system. FDE is a good example; the implementation on a Linux system is likely to use the kernel module dm-crypt (see Section 2.2.3) in combination with key management software Linux Unified Key Setup (LUKS). Android does not use LUKS for key management but dm-crypt serves as the backbone in Android’s FDE implementation.

2.1.2 Fragmentation

Fragmentation affects Android in two ways. First, the software platform as such, and secondly in the hardware platform engineered by the manufacturers.

Android on one device is not the same as Android on another device. This can easily be observed by comparing two smartphones from different brands side-by-side. Manufacturers introduce new layouts, customize the OS for their hardware, and even network providers may request further changes.
Customization is therefore not in control of Google but mostly carried out by the manufacturers. Backporting a new release to an older model certainly presents economic aspects for consideration. Manufacturers save money by not adapting the stock Android release. Moreover, this might lead to higher sales numbers for the new flagship model, as users who would like to have the latest Android features would have to buy a newer model.

The Google Nexus product line is different. This is the device platform used to develop, test and maintain Android. Nexus owners are more likely to receive upgrades for longer periods than other device owners. For example, the Nexus 4 which was released with Android 4.2 in Q4 2012 received several updates. The latest, as of this writing, is Android 5.1.1 released in Q1 2015. Other Android smartphones released in Q4 2012, such as the Samsung Galaxy S3 model GT-I9305, did not receive an update to Android 5.0 or newer.

Non-Nexus owners often wait longer for a release (including security fixes) to arrive or in the worst-case scenario do not receive an update at all. However, these devices often remain in use for several months or even years as Table 2.1 shows. These are the results of a distribution analysis that ended on May 4th, 2015 [19]. The results indicate that the majority of Android devices had Android KitKat installed, followed by two even older versions codenamed Jelly Bean. The first version of Jelly Bean was released in July 2012 and still covers more devices than Lollipop, released in Q4 2014 and Q1 2015.

<table>
<thead>
<tr>
<th>Version</th>
<th>Codename</th>
<th>API</th>
<th>Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2</td>
<td>Froyo</td>
<td>8</td>
<td>0.3%</td>
</tr>
<tr>
<td>2.3.3 – 2.3.7</td>
<td>Gingerbread</td>
<td>10</td>
<td>5.7%</td>
</tr>
<tr>
<td>4.0.3 – 4.0.4</td>
<td>Ice Cream Sandwich</td>
<td>15</td>
<td>5.3%</td>
</tr>
<tr>
<td>4.1.x</td>
<td></td>
<td>16</td>
<td>15.6%</td>
</tr>
<tr>
<td>4.2.x</td>
<td>Jelly Bean</td>
<td>17</td>
<td>18.1%</td>
</tr>
<tr>
<td>4.3</td>
<td></td>
<td>18</td>
<td>5.5%</td>
</tr>
<tr>
<td>4.4</td>
<td>KitKat</td>
<td>19</td>
<td>39.8%</td>
</tr>
<tr>
<td>5.0</td>
<td></td>
<td>21</td>
<td>9.0%</td>
</tr>
<tr>
<td>5.1</td>
<td>Lollipop</td>
<td>22</td>
<td>0.7%</td>
</tr>
</tbody>
</table>

Table 2.1: Android OS Version Distribution [19]

Despite the software aspect, fragmentation also affects the hardware. Different manufacturers target a great variety of users with devices ranging from cheap low-end to expensive high-end. This may have an influence on the device’s appearance but potentially also on hardware-based features. For example, many chip manufacturers create flash-memory modules. These are built to conform to a certain standard but there may also be differences in the interpretation of these standards.

Controlling fragmentation is an important aspect for Google, not just from a user’s but also from a developer’s perspective. Third party developers are a core element for a smartphone platform’s success. Because a platform with fewer applications is less attractive for users, a mobile platform divided in many different derivatives would not be as strong as a unique one.

In order to cope with fragmentation and to give users and developers a clearer perspective when dealing with Android, Google specified the Android Compatibility Definition Document (CDD) [18].
2.1.3 Nexus and Stock Android

The project focuses on Android 5.0 as with this version Google Nexus devices are shipped with default encryption and the implementation was improved in terms of security [20]. We also discuss Android 4.4.4 because it is currently the most widely deployed version with 39% of all Android devices running it (see Table 2.1).

Experiments, if not otherwise stated, will be carried out on Android 5.0 on a newly purchased Nexus 6 (see Appendix C). This is a smartphone that ships with default encryption enabled. The second Android version will be tested with a Nexus 4. This device was used for six months without FDE enabled and in the process of the project the feature was manually activated. In contrast to the other device, the Nexus 4 has a custom recovery image installed. In both devices, prior to any other activity, we unlocked the bootloader (see Section 4.2.1).

2.2 Encryption

Cryptography is the cornerstone of full-disk encryption. The block cipher currently in use is Advanced Encryption Standard (AES), which is based on a substitution-permutation network with a 16-byte (128-bit) block size and three different key lengths (128-bit, 192-bit and 256-bit) [39]. We present two block cipher operation modes: Cipher Block Chaining (CBC) which is currently used in the stock Android, and XEX encryption mode with Tweak and ciphertext Stealing (XTS), which is a named alternative in the CDD.

2.2.1 Cipher Block Chaining

The block cipher mode CBC [40] is one of the traditional, well-known modes of operation. In addition to being used for encryption, CBC is used to build other cryptographic primitives, such as the Message Authentication Code (MAC) CBC-MAC.

CBC prevents ciphertext block manipulations – such as block swapping, block insertion or block deletion – and frequency analysis or dictionary attacks with message dependency by XORing the previous block with the current block. This gives CBC the advantage of being a self-synchronizing mode, meaning sender and receiver do not have to synchronize an additional value like a counter or sequence number. Since the first block does not have a previous block to rely on, an Initialisation Vector (IV) is required.

The properties of an IV are [35]:

- Should only be used once with a certain key
- Can be publicly known
- Should be an unpredictable value of block size of the encryption algorithm
- Must be known by the encrypting and decrypting party
The following operations describe the encryption with CBC where $E_k()$ is the encryption transformation, $P_n$ the plaintext block to encrypt, and $C_{n-1}$ the previously encrypted block for message dependency.

Initialization ($n = -1$):  
$$C_{-1} = IV$$

For each plaintext block ($n > -1$):  
$$C_n = E_k(P_n \oplus C_{n-1})$$

Decryption, described by the following operations, works in reverse order to encryption. For the first block, the same IV value is used. After the current ciphertext block $C_n$ is decrypted with the decryption transformation, $D_k()$, it forms the value $C_{n-1}$ for the following ciphertext block to be decrypted.

Initialization ($n = -1$):  
$$C_{-1} = IV$$

For each ciphertext block ($n > -1$):  
$$P_n = D_k(C_n) \oplus C_{n-1}$$

A drawback of CBC is the *error propagation*. If in ciphertext block $C_n$ a 1-bit error occurs, the plaintext block $P_n$ will have approximately 50% erroneous bits. Due to the nature of a block cipher, inputs differing by one bit will have no apparent relationship after transformation. Additionally to that faulty plaintext block, block $P_{n+1}$ will have a 1-bit error in the same position as the ciphertext block $C_n$. The reason for this faulty block is the XOR operation introduced through message dependency.

Error propagation also influences the use of CBC for storage application. For example, a 1 GB disk is divided by the cipher block size into different blocks. Each block is then encrypted as described above. At a later point in time a file, stored somewhere in the middle of the disk, changes. Due to CBC’s message dependency, each subsequent block would need to be decrypted and re-encrypted from that changed block onwards. To counter this problem, CBC does not encrypt a whole disk as single plaintext, but rather encrypts each disk sector as individual plaintext.

The earlier list of IV properties mentions that an IV should only be used once per key. However, if there exists only one IV and a disk is encrypted using CBC sector-by-sector, the IV would be used many times with the same key. If multiple sectors contain the same data, for example zero in an empty disk, these sectors encrypt to the same ciphertext. An adversary accessing the encrypted disk can observe the similar ciphertexts and conclude that their plaintexts are equal.

### 2.2.1.1 Encrypted Salt-Sector Initialization Vector

Encrypted Salt-Sector Initialization Vector (ESSIV) [16], designed by Clemens Fruhwirth, is an algorithm to create a per-sector IV based on the encryption key and the sector number. This is a solution to counter the issue of repeating IV with the same key when using AES-CBC to encrypt storage devices on a sector-by-sector basis. Like other FDE implementations, Android makes use of this method when CBC block cipher mode is used [49].
ESSIV first calculates a salt by hashing the encryption key. The design does not specify which hash function has to be used. The only restriction from the perspective of ESSIV is the output size. Because of the next step, the output has to be of equal size as the block cipher algorithm’s block size.

\[
salt = H(key)
\]

This key-based salt value is then used as the encryption key to encrypt the sector-number, resulting in the IV for that specific sector.

\[
IV(\text{sector } - \text{number}) = E_{\text{salt}}(\text{sector } - \text{number})
\]

This mode is considered a private IV generation method since the secret encryption key is involved in the process.

### 2.2.2 XEX Encryption Mode with Tweak and Ciphertext Stealing

An alternative mode to CBC is XTS, defined by the IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices (IEEE Std. 1619) [27]. It is based on Phillip Rogaway’s XOR-Encrypt-XOR (XEX) [53] mode of operation with an additional tweak parameter and ciphertext stealing. The tweak parameter is a 128-bit value used to represent the logical position of the data being processed by the block cipher.

Ciphertext stealing (see Figure 2.1) is introduced to process plaintexts which do not divide into cipher block size (128-bit) evenly. Instead of including padding in the last block, the last two ciphertext blocks will be reordered. The ciphertext of the second to last plaintext block \((P_{n-1})\) is split after \(m\)-bits into \(C_{n-1}\) and \(C^*_{n-1}\), where \(m\) is calculated as \(128\) - \(\text{length}(P_n)\). The last plaintext block \(P_n\) is concatenated with \(C^*_{n-1}\) to form a full-sized block which can be encrypted. In a final step, the full block \(C_n\) and the remaining bits of \(C_{n-1}\) are swapped. The same procedure is used for decryption to process the last two ciphertext blocks.

![Figure 2.1: XTS Ciphertext Stealing, based on [27, Fig. 2]](attachment://image.png)
According to the IEEE standard, XTS mode has either a 256-bit or a 512-bit key. The key will be split in half and used for two individual AES transformations. Hence XTS-256 will consist of two AES-128 and XTS-512 of two AES-256 transformations.

The first AES operation is used to encrypt the tweak value with the second half of the encryption key ($K_2$). Then, the second AES operation is used to encrypt the plaintext block XORed with the encrypted tweak value. The encryption process shown in Figure 2.2 is a simplified version of what is defined in the IEEE Std. 1619. The decryption process follows the same scheme but uses the AES decrypt transformation.

![Figure 2.2: XTS Encryption Process, based on [27, Fig. 3]](image)

With XTS processing a storage media sector-by-sector, the mode has similarities to the Electronic Code Book (ECB). The encrypted disk will leak information if a block in the sector contains the same data. It is therefore considered to be an ECB-like mode of operation [54, p. 60].

### 2.2.3 Device-Mapper and dm-crypt

Device-Mapper (dm) is part of the Linux kernel as of version 2.6. It is a versatile module and not only engaged in FDE. Its core functionality is the mapping of block devices, such as mapping a virtual onto a physical block device. For dm, several virtual device targets were developed to implement a variety of mappings.

Dm-crypt is one of these targets and maps an encrypted block device transparently onto a virtual block device. The meaning of *transparent encryption* is such that read operations get decrypted in place and disk writes get immediately encrypted prior to storing on disk. The advantage of this approach is that the upper layers, in particular OS and applications, do not have to be aware of the encryption process. The disadvantage is that the cryptographic key material needs to be held active (e.g. in main memory) for as long as the encrypted device is mapped.
The design of dm-crypt is flexible in terms of cryptographic primitives usage. The Linux kernel’s crypto API [36] is used and thus all the primitives (e.g. the hash algorithms, ciphers and IV generators, etc.) developed for this API could potentially be used with dm-crypt. On a running system, the command `cat /proc/crypto` executed in a terminal will output what primitives are supported.

## 2.3 Password Stretching

Passwords consisting of visible ASCII characters do not allow the user to create a password using the full keyspace with large entropy. The functionality of password stretching is used to convert a user-defined password into a cryptographic key. Android employs Password-Based Key Derivation Function 2 (PBKDF2) and scrypt during the key management procedure discussed in 3.2.1.

For the scope of this thesis the term password, if not stated otherwise, refers to the screen lock methods PIN, password or pattern.

### 2.3.1 Password-Based Key Derivation Function 2

PBKDF2 was designed by RSA Laboratories and is part of Public-Key Cryptography Standards (PKCS) #5 [30]. The National Institute of Standards and Technology (NIST) special publication 800-132 recommends PBKDF2 for the derivation of master keys [41].

PBKDF2 takes four input values:

- **Password** — User-provided input
- **Salt** — Per-user diversification value, randomly generated
- **Count** — Iteration counter
- **dkLen** — Length of the derived key

At the core of PBKDF2 is a Pseudo-Random Function (PRF). Both the PKCS #5 standard and the NIST recommendation define this as a Hash-based Message Authentication Code (HMAC) with the SHA-1 hash function. Nevertheless, other hash functions or cryptographic primitives that have a pseudo-random output are possible as well.

The salt value is used as a diversification value to prevent the identification of equal passwords among users. Additionally, an attacker who would like to precompute all possible keys corresponding to a password dictionary must do this for every possible salt, increasing storage usage for precomputation tables.

The counter is the cost parameter which slows down the derivation process. It defines how many times the PRF is executed to generate a block of the stretched key.

This does not only slow down an attacker but also regular users entering their password once. The slowing down of one entry should not be noticeable by the user. An attacker, however, who will most likely have to enter many passwords before identifying the correct value, is slowed down on each individual attempt. In the NIST recommendation, a minimum of 1 000 for standard and 10 million iterations for critical applications is specified.
increasing computational power, the number of iterations will also have to increase to keep the slowing down effect constant.

A drawback of PBKDF2 is its limited focus on being only computationally intensive. Creating special hardware running the transformation in parallel is possible at relatively low cost.

### 2.3.2 scrypt

In [43], Colin Percival introduces the concept of memory-hard algorithms and sequential memory-hard functions, and describes with scrypt a new Key Derivation Function (KDF) based on them. In contrast to the previously described PBKDF2, scrypt is designed to be not just computationally intensive but also memory-hard. This should make the use of parallelization hardware for exhaustive key search harder.

The scrypt algorithm takes the following input:

- **Password** — User-provided input
- **Salt** — Per-user diversification value, randomly generated
- **$N$** — CPU/Memory Cost Parameter
- **$r$** — Block size parameter
- **$p$** — Parallelization parameter
- **dkLen** — Length of the derived key

Overall, scrypt is a combination of the following algorithms:

- **Salsa20/8 Core** — Round-reduced variant of Salsa20 Core
- **scryptBlockMix** — Block mix algorithm with Salsa20/8 Core as hash function which in turn serves as hash function in scryptROMix
- **scryptROMix** — Algorithm for computing large random values and accessing them randomly
- **PBKDF2** — With HMAC_SHA256 as PRF

To compute the derived key with scrypt, the first step is to process the PBKDF2 with the counter value set to one and the output length parameter calculated as the product: $p \times 128 \times r$.

The output is the concatenation of blocks 0 to $p - 1$. On each block, the algorithm scryptROMix is executed, which will execute scryptBlockMix and this in turn executes Salsa20/8 Core iteratively.

In order to have a similar impact with increasing computing resources, scrypt can be tweaked via three of its input values ($N$, $p$ and $r$).
2.4 Chapter Summary

The Android mobile OS platform is not a heterogeneous construct. Multiple players interfere in the development and engineering of devices. This complicates the process of continued development, release of new versions and upgrade of active devices. The issue is named fragmentation and can also affect FDE.

Cryptography is elementary for this application. Stock Android uses AES-CBC with the private IV generation scheme ESSIV. This chapter also introduced XTS, an alternative CBC. In contrast to CBC this mode is relatively new and specifically designed for storage media encryption.

Lastly, two password stretching mechanisms, PBKDF2 and scrypt, used in Android’s FDE key management, were introduced. Scrypt also employs PBKDF2 but enhances the derivation process to be computationally expensive and memory-hard. It is believed to be harder to develop parallelization hardware for scrypt due to it being not only computationally intensive.
Chapter 3

Android Full-Disk Encryption

This chapter will discuss how FDE was implemented in Android 5.0, the differences with Android 4.4.4, and the underlying threat model.

3.1 Compatibility Issues

Full-disk encryption aligns with the objective “Protect user data”, mentioned first in the platform security architecture documentation [22].

Despite the media announcement [59] and the listing of Android 5.0’s security enhancements [21], FDE is not a default feature for all devices shipped with Android 5.0. It appears that the current state is yet another factor affected by fragmentation (see Section 2.1.2). The degree to which all manufacturers must offer FDE is covered in Section 9.9 of the CDD [18] and summarized as follows.

**MUST requirements:**

1. Support FDE, if a lock screen is implemented
2. Encrypt the `/data` partition and non-removable internal SD card partitions
3. Use AES with minimum 128-bit key length
4. Use a symmetric block cipher mode designed for storage (e.g. AES-CBC-ESSIV or AES-XTS)
5. Cryptographically bind the password stretching algorithm to a hardware-backed keystore, if present

**MUST NOT requirements:**

1. Write the encryption key to storage without it being encrypted
2. Send off the encryption key from the device, under any circumstance
3.2 Implementation in Android 5.0

SHOULD requirements:

1. Enable FDE at all times
2. AES encrypt the disk encryption key, if not in active use
   (a) Use a stretched lock-screen passcode with a slow stretching algorithm (e.g. scrypt or PBKDF2)
   (b) Use a default password, if no passcode is specified by the user

The interpretation of the categories (MUST, MUST NOT and SHOULD) follows the declaration in RFC 2119 [3].

The CDD specifications leave room for interpretation. FDE is only mandatory for devices with a lock-screen (display requires password to unlock). This makes sense since data on devices without lock-screen will always be accessible. However, this point is unlikely to affect smartphones. Users will expect a lock-screen and therefore smartphones will offer FDE.

The second MUST requirement is important. The partition /data stores installed applications and their internal data. The internal SD card holds data stored by the user, for example when using the device as a USB disk. Hence only user data will be encrypted and not the Android framework itself.

Furthermore, the fifth MUST requirement allows to encrypt the storage without the central enhancement in Android 5.0 [21], which essentially reduces FDE to pre-Android 5.0.

The first SHOULD requirement probably led to public criticism by journalists [7] saying that Google takes back their previous announcement. However, given that many manufacturers have an interest in Android, it is more likely that certain manufacturers needed time to take this into consideration for their new devices. Without doubt FDE is an additional task for the device which can affect its computational speed or battery usage. The CDD states clearly that this point is likely to become a MUST requirement in the near future.

3.2 Implementation in Android 5.0

For this section, the source code of Android 5.0 for the Nexus 6 device was downloaded from the official Git repository¹ and reviewed. The relevant branch name is android-5.0.0_r3.0.01 [44], respectively android-5.0.0_r3 in the repository. Where applicable, the code for Android 4.4.4 for the Nexus 4 device (branch android-4.4.4_r1) was reviewed and compared with Android 5.0.

3.2.1 Key Management

Android’s FDE employs a two-level key hierarchy as shown in Figure 3.1. The first level is the Disk Encryption Key (DEK), also called master key, used to encrypt the disk partition. The second level of the hierarchy is another key used to protect the master key, called Key Encryption Key (KEK).

¹android.googlesource.com
3.2. Implementation in Android 5.0 3. Android Full-Disk Encryption

Key generation is handled by the function `create_encrypted_random_key` [46, line 1413]. It will read the master key and the salt, both 16-byte (128-bit) long, from `/dev/urandom`, a standard Linux pseudo-random number generator interface. We could not find a function providing a key change mechanism in the Android source code. Therefore the master key, as well as the salt, appear to be static for as long as the encrypted partition is not wiped and re-encrypted.

In order to preserve the security of the master key, the KEK needs to be at least as equally strong. The generation of the 128-bit KEK involves the password and a KDF function. In versions before Android 4.4, the default KDF was PBKDF2 (Section 2.3.1). Since then, scrypt was included (Section 2.3.2) and as of Android 5.0 the KDF cryptographically binds the KEK with the device. Both Nexus devices offer this hardware feature.

The function `scrypt_keymaster` [46, line 1196] implements this requirement. First, scrypt is applied on the password resulting in the output value $I_{KEY}$. This output is then passed to the keymaster with the request to sign. The keymaster will return the $SIGNATURE$, which is fed into another scrypt operation. The output of the second scrypt application is the hardware-backed KEK that is then used in the function `encrypt_master_key` [46, line 1245] to encrypt the master key with AES-CBC.

Figure 3.2 visualizes the process described above. The values salt, encrypted master key, RSA signature key blob and the KDF type are stored in the data structure `crypto footer` [47, line 87]. Part of the Rivest-Shamir-Adleman (RSA) signature key blob is the encrypted private exponent, protected with an internal, static keymaster key.
In the case of default FDE, the password used during KEK generation is not yet set by the user. For that purpose, a hexadecimal encoded default password (ASCII hex encoded default_password) is stored in the source code [46, line 74]. If the user specifies a password after the first boot or at a later point in time, a key translation is performed. Key translation is the process of decrypting a ciphertext with the current key and re-encrypting it with another key. In Android, this functionality is provided by the function cryptfs_changepw [46, line 3258]. The function first verifies that the valid master key is decrypted, then encrypts the master key with the new KEK derived from the user password and writes the result back into the crypto footer.
3.2.2 Encrypting on First Boot

Devices like the Nexus 6 that have default FDE will encrypt the storage on first boot. A high-level description of this process is given in [49]. Our analysis, however, is primarily source code based.

The system property `ro.crypto.state` is set to unencrypted. As the device starts, the `fstab` file, which contains the configuration for partition mounting, is processed. When the userdata entry has the flag `forceencrypt` set, Android will start the encryption process on first boot.

The system property `vold.decrypt` is set to `trigger_encryption`. The startup script, `init.rc` [51], contains an action for this property that will start the service `encrypt` that executes `/system/bin/vdc --wait cryptfs enablecrypto inplace default`.

The command-line tool `vdc` is part of the Volume Daemon (vold) and provides a series of `cryptfs` operations, specified in `CommandListener.cpp` [45]. The password was omitted because the argument `default` identifies that the default password is used.

Using the default password will trigger the function `cryptfs_enable_default` [46, line 3252], calling `cryptfs_enable_internal` [46, line 2887] with the parameters:

- operation type — `inplace`
- crypt_type — `CRYPT_TYPE_DEFAULT`
- passwd — `DEFAULT_PASSWORD`
- allow_reboot flag — `false`

The function `cryptfs_enable_internal` now coordinates all the required steps to create the encrypted partition. First, it will shut down the main class services of `init.rc` by setting the property `vold.decrypt` to `trigger_shutdown_framework` and then unmount the userdata partition and mount `/data` on a temporary file system (tmpfs). This allows the framework to run without userdata mounted for the duration of encryption.

At the beginning of the encryption process is the creation of the crypto footer using the function `cryptfs_init_crypt_mnt_ftr` [46, line 2097]. The path after the `forceencrypt` flag in the `fstab` defines the location – in the case of the Nexus 6, this is the metadata partition. With the crypto footer initialized, the master key is created as described in Section 3.2.1. The encrypted master key is decrypted again by calling the function `decrypt_master_key` [46, line 1393].

Next, `create_crypto_blk_dev` [46, line 1052] is called to create a virtual block device for dm-crypt. The configuration for the device mapping is loaded with `load_crypto_mapping_table` [46, line 972], part of which comes from the crypto footer, such as the `crypto_type_name` parameter which specifies the block cipher, block cipher mode and IV generation mode (`aes-cbc-essiv:sha256`). This string is statically defined in the Android source code [46, line 3078] and while the CDD allows the use of a different cipher, mode and IV generation setting, the user cannot influence it.

With these steps carried out, the next function called by `cryptfs_enable_internal` will lead to the call of `encrypt_groups` [46, line 2320], where the unencrypted partition is read from block device and written to the dm-crypt virtual device. With transparent encryption, each write operation automatically encrypts the data.
The final steps are to undo the dm-crypt mapping, delete the virtual block device (function call \texttt{delete\_crypto\_blk\_dev} [46, line 1126]) and continue the first boot sequence. This includes deriving the KEK again from the password in order to decrypt the master key and create the dm-crypt device which is then mounted on \texttt{/data}.

### 3.2.3 Decryption on Subsequent Boot

The following procedure is also documented in [49] and [12] on a higher level. We will again provide more detailed information by reviewing the source code.

Android now detects on start-up if the userdata partition is encrypted. This functionality is provided by the \texttt{cryptfs} command \texttt{cryptocomplete} which reads out the system property \texttt{ro.crypto.state}. The function not only detects if the encryption is set; it also checks for flags indicating if the encryption process is still ongoing or has failed.

Assuming the encryption finished successfully, the next step is mounting the \texttt{/data} on a tmpfs so that the framework can proceed with the boot sequence. When \texttt{/data} is mounted, the init.rc service class main is restarted and the \texttt{CryptKeeper} [48] prompt (hereafter \textit{Start-Up} prompt) is displayed. CryptKeeper will call the \texttt{decryptStorage} [52, line 2183] function of the \texttt{MountService} class, which in turn calls the native code command \texttt{cryptfs} passing the action \texttt{checkpw} as an argument.

The \texttt{checkpw} action, defined in \texttt{CommandListener.cpp} [45, line 571] calls the function \texttt{cryptfs\_check\_passwd} [46, line 1992], passing the password to check. The function first detects whether the userdata partition is unmounted, reads the crypto footer from its location, and passes the password and the crypto footer to the function \texttt{test\_mount\_encrypted\_fs} [46, line 1721]. This function invokes the master key decryption by calling \texttt{decrypt\_master\_key} and creates the virtual dm-crypt.

When the previously discussed actions are successful, \texttt{decryptStorage} will call the native code command \texttt{cryptfs} passing the action \texttt{restart} as an argument. That action causes the function \texttt{cryptfs\_restart\_internal} [46, line 1542] to be called, which in turn sets the \texttt{vold.decrypt} system property to \texttt{trigger\_reset\_main}. As a consequence, the init.rc process stops the main class services, which allows to unmount \texttt{/data} from the tmpfs and re-mount to \texttt{/data} on the dm-crypt virtual device for userdata, offering transparent decryption. Afterwards, the full framework can start and finish the boot procedure.

If the user did not specify a password after the default encryption of userdata, there will not be any interaction with the CryptKeeper user interface. The vold will set the \texttt{vold.decrypt} system property to \texttt{trigger\_default\_encryption}, causing init.rc to start the service \texttt{defaultcrypto}. This service will run the procedure described above, providing the default password where otherwise a user’s password was used.

### 3.2.4 Comparison with Android 4.4.4

#### 3.2.4.1 Code Base

The core functionality for managing FDE, as described in Sections 3.2.2 and 3.2.3, is provided by cryptfs.c [46]. We compared this file of each of the relevant branches for Android 4.4.4 and 5.0 and identified over 1 000 new lines of code.

The majority of these lines deal with the new KDF type KDF\_SCRYPT\_KEYMASTER,
which stands for the hardware-backed KEK generation described in Section 3.2.1. It is therefore clear why the Android 4.4.4 of our Nexus 4 does not provide hardware-backing although the device’s hardware would be capable of doing so. An update of the system would certainly include this feature. Since Android 4.4.4 does not offer hardware-backing, FDE in this version is prone to Offline Exhaustive Password Search Attacks (see Section 4.4.2). Furthermore, new code was introduced to cope with the encryption of Flash-Friendly File System (F2FS), while the support of the File Allocation Table (FAT) file system was removed. Another part of the new code was introduced to support pattern screen lock authentication with FDE, such as the function `adjust_passwd` [46, line 1953] that converts the hexadecimal representation of a pattern to a different format.

### 3.2.4.2 User Interaction

The clearest difference to users is that Nexus 4 owners had to manually enable FDE and had to set a screen lock in that process. There was no default password and users could only choose between Personal Identification Number (PIN) and password. With Android 5.0, the pattern screen lock can be chosen as well.

Android 4.4.4 users could also configure Face Unlock, a lock-screen authentication bypass method based on facial recognition. In Android 5.0, the latter is not listed as a screen lock method any more. However, it is not removed either. It was rearranged into the new Smart Lock feature, a built-in application which allows users to define certain trust agents. When such a trust agent is presented, the screen lock changes to swipe-only and the users do not have to enter their password. We are going to discuss Smart Lock and its potential misuse in more detail in Section 4.5.

Another noticeable difference is that with the new Android, users can choose the options Swipe or None for screen lock. However, in those cases there is no password which can be used to derive the KEK from.

The last variation to mention in conjunction with user interaction is the confirmation screen shown in Figure 3.3. Pre-Android 5.0 users could not activate FDE and opt out of entering the password on device start-up. If a user chooses the second option, “No thanks”, the smartphone starts without request for the password. Our tests showed that the master key is again encrypted using the default password.

![Figure 3.3: Confirmation Screen](image-url)
3.3 Threat Model

In this section we will discuss the threat model at the foundation of Android’s FDE, which we identified during our analysis. We will identify which security services are covered but may be desirable.

3.3.1 Threat Modelling Questionnaire

**Question 1:** Which are the assets protected?

Despite the name, we have identified in Section 3.2 that not the whole disk but only the userdata partition is encrypted. Therefore, the assets that should be protected by FDE are data stored on that partition. There is no protection towards data stored on any other partition. The data stored in userdata are mainly what users create and use; for example, the installed applications or the internal, simulated SD card.

**Question 2:** What actions should be protected against?

In Section 3.2.1, we identified that the userdata partition is encrypted using CBC, a block cipher mode which only provides data confidentiality, a means to prevent unauthorized parties from observing any unencrypted information.

**Question 3:** Who are the adversaries the disk has to be protected from?

Android FDE should provide confidentiality of information against an active adversary who has physical access to the smartphone. Such adversaries can disassemble the smartphone, analyse single components of the hardware, power on or off the device and access it for as long as needed. Nevertheless, such an attacker will always be able to mount an exhaustive key search attack.

**Question 4:** What are the attack scenarios the disk has to be protected against?

Full-disk encryption aims to protect data at rest. We differentiate three states of data:

- in motion — Data is moving from location to another location
- in use — Data is employed in an operation
- at rest — Data is stored on disk

As dm-crypt is used to transparently encrypt the userdata partition (see Section 3.2), the data is in use as long as the partition is decrypted and mounted in the file system. At rest would require the userdata partition to be unmounted, but that is only the case when the device is shut down.

**Question 5:** What are the attack scenarios the disk cannot be protected against?

The answer to the previous question already states the limitation with the current approach. Hence the attack scenario not protected against is when an adversary has access to the device while it is turned on. The last scenario is an installed malicious application elevating privileges and breaking out of its runtime sandbox, enabling it to access user data on a running device. We will not consider this scenario further as this not an issue of FDE but more of an issue for the application security model of Android.
3.3.2 Security Services Not Provided

Currently, data integrity and authentication are security services not provided. Data integrity provides a means to detect unauthorized changes to the data. FDE with AES-CBC does not provide such means. An adversary with access to the encrypted userdata partition can modify the ciphertext. The cipher does not detect this alteration and hands over this burden to any application operating on top, accessing data on the partition. This also indicates why authentication is not provided. Otherwise, the adversary would not be in the position to alter any bits.

Additionally, as we already outlined above, the current implementation does not offer protection to data in any other state than at rest. Hence protection for data in motion and in use requires other means.

3.4 Chapter Summary

The CDD [18] defines the compatibility requirements that affect FDE. These requirements leave room for interpretation and could therefore be affected by fragmentation.

In Android 5.0 the possibility exists to encrypt userdata of a new device on first boot. If that is configured, a tmpfs is mounted on /data so that the framework can start and have a dm-crypt virtual device for userdata created. The unencrypted data is then read from userdata and written to the virtual device. Due to transparent encryption, each write operation on dm-crypt encrypts the data prior to flushing on disk. After the encryption is completed, the framework is restarted and the device will be ready for use.

When the encryption is successful, each subsequent boot will again create the dm-crypt virtual device and mount /data on it. As with transparent encryption, the decryption operation reads the encrypted data from disk and decrypts it prior to passing it to the upper layer.

The threat model covered with Android 5.0 FDE is only concerned with data confidentiality of data at rest. Therefore, an adversary can manipulate the encrypted data on disk and the current implementation has no means to detect this. With transparent encryption employed, the encrypted userdata partition is readable when mounted, which is the case as long as the device is powered on. The screen lock password, however, builds a fundamental element in FDE. This password is used to derive the KEK which is used to encrypt the master key. Moreover, this is the only means that protects the data if the device is powered on.
Chapter 4

Security Assessment

This chapter introduces the attack scenarios we considered and existing countermeasures. Next follows the introduction of different known attacks on underlying block cipher mode AES-CBC and highlights where the alternative mode, AES-XTS, has similar issues. Our assessment continues with attacks against the implementation and finishes with a discussion about the potential misuse of Smart Lock.

There is a given similarity with FDE in other computing systems and especially with Linux systems (see Section 2.2.3). Therefore the attacks discussed in this security assessment might originate in the Linux world but have a significance for Android as well.

4.1 Attack Scenarios

In Section 3.3, we identified the threat model of Android’s implementation. Based on that, we define one attack scenario which is covered and another one which is not covered by the threat model.

In both attack scenarios, the adversary is in possession of the smartphone and will have as much time as needed to carry out his attack. We therefore assume that the smartphone was stolen, lost or came under his control otherwise.

Table 4.1 on page 24 shows which attacks or weaknesses are affected by which attack scenario.

Device-OFF

This first attack scenario is based on the fourth question in the threat model evaluation. We have outlined that FDE in general protects only data at rest, for which a smartphone must be powered off. This is therefore the attack scenario covered by the threat model.

Device-ON

The opposite to the above attack is a powered on device scenario, which the identified threat model does not cover. We decided to consider this scenario as well because of its significance. A UK-based study [42, p. 48] conducted in 2011 states that 81% of smartphone users do not shut down their devices. This means that data stored on these smartphones would not be protected by FDE alone.

Additionally, the screen lock method is incorporated in the key management process supporting FDE. Therefore there is a strong link between attacks against lock-screen au-
thentication and FDE. Moreover, FDE is implemented as a means to protect a user’s data, which would be undermined if lock-screen authentication is easily circumvented.

4.2 Existing Generic Countermeasures

Both Android versions 4.4.4 and 5.0 have generic countermeasures employed which can mitigate some of the attacks and weaknesses discussed hereafter. Table 4.1 on page 24 gives an overview of their applicability.

4.2.1 Locked Bootloader

Smartphones can boot into a bootloader by pressing a combination of buttons. For the Nexus 4 and 6 these were the Power and the Volume Down. Devices booted into bootloader are accessible from any USB connected host with the developer tool `fastboot`. It is part of the Android Software Development Kit (SDK) and can be used to either boot from a specific image or flash a new image onto the device. This, however, is only possible if the bootloader is unlocked.

Although not mentioned in the current CDD [18], our personal experience is that Android smartphones are normally shipped with a locked bootloader. However, the Samsung Galaxy SII is an example of a smartphone shipped with an unlocked bootloader [58]. We believe that future revisions of this document should cover this aspect and advise a locked bootloader.

The procedure of locking and unlocking the bootloader can be done with the `fastboot` tool as well. Executing the command `fastboot oem unlock` with a connected smartphone booted in bootloader will unlock it. The action `lock` will revert the unlock. In both Android versions under our investigation, executing the above command triggers a wipe on the userdata partition. Despite the possibility of data residue due to the File Translation Layer (FTL), which only affects devices without default FDE as our Nexus 4 (see Section 4.4.7.3), there is the threat of a flaw in the OS not actually carrying out the wipe operation [57].

The consequence is that an adversary carrying out an attack which requires to boot from or flash a custom image must have an unlocked bootloader. If that is the case for the target device or if they know that the wipe command will fail, they can continue without risking to lose any data. Otherwise, they would need to image the userdata partition beforehand (see Appendix B). Locking the bootloader may not protect against an attacker with the knowledge and tools to attack the hardware layer, for example using a present JTAG interface or unsoldering the flash-memory chip.

4.2.2 Disabled Android Debug Bridge

The Android Debug Bridge (adb) is another tool from the Android SDK. In contrast to fastboot, the smartphone does not need to be booted in a special state. The tool is versatile, allowing to push or pull files, execute commands on the device’s shell or simply connect to the Android with a remote shell.

The smartphone’s adb daemon starts the remote shell with root-privilege. However, on normal builds shipped with devices these privileges are dropped to the user identification (UID) 2000, named shell [12]. Execution of root-privileged commands is therefore prevented.
Nevertheless, even with UID 2000, adb is a powerful tool. Downloading of user specific data such as photos or text messages is possible with restricted privileges.

Android is not shipped with the adb daemon enabled by default on a smartphone and certainly most users will not activate it by mistake. First, the activation requires the user to perform two actions in the Settings menu. Secondly, prior to finishing both of the below outlined steps, the user will be notified and can abort the activation.

1. **Activate Developer Options** — The user must tap seven times on the build number field in the system information screen under the device settings menu

2. **Activate USB Debugging** — The user must tick the checkbox in the developer options menu

Android has introduced an additional security measure as of Android 4.2.2. Previous to that version, an adb service enabled device could be accessed by any host and the adb tool. In order to mitigate this vulnerability, a host authentication similar to SSH was introduced. On the first connection attempt, a smartphone displays the host’s RSA key fingerprint and asks the user to either accept or deny the connection. Users can suppress this request by ticking the checkbox about storing the key fingerprint. Otherwise, each time the adb service restarts, the user will have to confirm the host’s authentication request. Devices which do not have a screen lock set are vulnerable to bypass this authentication, since anyone with physical access to the device can confirm the authentication request. This is another reason why lock-screen authentication bypassing is considered in Section 4.5.

### 4.2.3 Android Device Manager

The Android Device Manager (ADM) is a web service that allows the user to perform three actions – **Ring, Lock** and **Erase** – by remote. The feature is enabled by default, but users can disable it in the Security menu under Device administrators.

From a security perspective, the first command is irrelevant. The Lock command locks the smartphone screen with the active screen lock. If the device has no screen lock set, Lock will configure a password that the user provides in the web interface. The Erase action is especially valuable when a device is lost or stolen so as to perform a factory reset and wipe the userdata partition.

A major drawback is the required network access. If the smartphone is not connected to the internet, there is no communication path from the web service to instruct the action. As a consequence, devices like our smartphones which have an external SIM-card slot can easily be disconnected.
### 4.2. Existing Generic Countermeasures

Generic Countermeasures Attack Scenarios Android

<table>
<thead>
<tr>
<th></th>
<th>Locked</th>
<th>Disabled</th>
<th>ADM†</th>
<th>Device-Off</th>
<th>Device-On</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attacks on</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cryptography</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Malreability Attack</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Content Leak Attack</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Attacks on the Implementation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Online Exhaustive Password Search Attack</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Offline Exhaustive Password Search Attack</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Semi-Offline Exhaustive Password Search Attack</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>Static Default Password and Master Key</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>Cold-Boot Attack</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Evil-Mail Attack</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Potential Weaknesses</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Predefined Cryptographic Primitives</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Lack of Entropy in Master Key</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Unencrypted Data Residue on Flash-Memory</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Trusted Execution Environment Vulnerabilities</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>Bypass</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trusted Places</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
</tr>
<tr>
<td>Trusted Devices</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
</tr>
<tr>
<td>by Attacking</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Facial Recognition</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Accelerometer</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

† access via hardware-layer possible † device needs network connectivity ● affects devices without default encryption

**Table 4.1:** Attack Surface: Overview

Mapping of Attacks and Weaknesses with Countermeasures, Attack Scenarios and Affected Versions
4.3 Attacks on Cryptography

In the following sections known attacks against Android’s default block cipher mode CBC [18] will be discussed. Where applicable, we will examine whether XTS resolves the issue or is also prone to an attack. A key point we have to consider here is that the end-user has no control over the cryptographic means used on his device. The OS developers have full control in this aspect.

Before discussing these attacks, it is worth mentioning the article [2] by Bogdanov, Khovratovich and Rechberger describing the biclique cryptanalysis on full AES with lower computational complexity than exhaustive key search attacks. However, to cite the authors: “As our attacks are of high computational complexity, they do not threaten the practical use of AES in any way” [2, p. 1]. The reason for mentioning this research is to illustrate that absolute security is an unrealistic state.

4.3.1 Watermarking and Movable Cipher Block Attacks

These two attacks on the CBC mode (see Section 2.2.1) are not affecting the implementation in Android OS. We mention them for the sake of completeness on the CBC discussion.

Watermarking attacks [16, p. 64] allow an attacker to tag data which will be discoverable while the data is encrypted. For this kind of attack, the IV must be predictable. As Android uses ESSIV (see Section 2.2.1.1), a non-public IV mode, this attack will not work.

Movable Cipher Block attacks [16, p. 69] allow a legitimate user of a multi-user system with FDE to cut a sector containing data (e.g. a file) which he cannot access and then paste the encrypted sector on a disk location which will decrypt to a file accessible for them. Although Android is capable of serving as a multi-user system, the majority of users will probably not make use of this feature to share the device among different untrusted users. If a such a smartphone is shared, the users most likely have some sort of relation and are thus trusted. We therefore do not consider this kind of attack relevant for Android devices at the moment.

4.3.2 Malleability Attack

4.3.2.1 Introduction

The non-malleability property of a cryptographic algorithm is informally defined by Dolev, Dwork and Naor [10] who argue that “[... ] given the ciphertext it is impossible to generate a different ciphertext so that the respective plaintexts are related.” The goal of the malleability attack, as described in [16, p. 69], is exactly the opposite. The attacker attempts to create a different ciphertext which manipulates the plaintext in a meaningful manner during decryption.

In terms of CBC, as decryption is dependent on previous ciphertext blocks, the attacker generates a ciphertext block $C_{n-1}$ that will influence the decryption of $P_n$. In this process, two original plaintext blocks will be altered. First, $P_{n-1}$, as its ciphertext was manipulated to influence $P_n$, and of course $P_n$ itself.
4.3.2.2 Attack Analysis

The practicality of CBC malleability attacks was demonstrated against a Linux (Ubuntu 12.04) installation with FDE [33]. The author successfully manipulated an encrypted file through manipulation of a previous ciphertext block. Additionally, apart from modifying code, an attacker could target configuration files and potentially manipulate security settings.

The alternative mode, AES-XTS, does not prevent this kind of attacks completely. In their response to NIST’s request to comment on the recommendation of AES-XTS as a mode of operation for disk encryption, Bharadwaj and Ferguson [1] described malleability attacks. Rogaway also argues that an ECB-like mode such as AES-XTS is most likely to be vulnerable to malleability attacks [54].

4.3.2.3 Countermeasures

Solving the underlying issue of the CBC malleability attack can be done by extending the security service from confidentiality-only to involve data integrity (see Section 3.3.2). An example would be the use of an authenticated-encryption mode. Adding data integrity for disk encryption must be done in a form that does not extend the disk unit available for storing the encrypted data. Some block modes exist which do prevent malleability attacks directly. As outlined in [15], Encrypt-Mix-Encrypt (EME), Liskov-Rivest-Wagner (LRW), or CBC-Mask-CBC (CMC) are non-malleable.

4.3.2.4 Attack Surface

Table 4.2 gives an overview of the attack surface for this vulnerability. The attack is mounted against the encrypted data, therefore the attack scenario is Device-OFF and if the bootloader is locked an adversary is restricted from accessing the storage. As both versions use this block cipher mode, both are vulnerable to this kind of attacks.

<table>
<thead>
<tr>
<th>Generic Countermeasures</th>
<th>Attack Scenarios</th>
<th>Android</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locked</td>
<td>Device-OFF</td>
<td>4.4.4</td>
</tr>
<tr>
<td>Disabled</td>
<td>Device-ON</td>
<td>5.0</td>
</tr>
<tr>
<td>Bootloader†</td>
<td>adb</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓</td>
<td>–</td>
<td>✓</td>
</tr>
<tr>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

† access via hardware-layer possible

Table 4.2: Attack Surface: Malleability Attack

4.3.3 Content Leak Attack

4.3.3.1 Introduction

Another attack on CBC mode, discussed by Clemens Fruhwirth in his paper on new hard disk encryption methods [16, p. 60], is the Content Leak Attack.

If we recall the decryption operation of CBC, a preceding ciphertext block influences the decryption of the current plaintext block. The influence itself is rather simple by an XOR operation. This, as well as the fact that an attacker has access to all ciphertext blocks, is the foundation of the attack.
4.3.3.2 Attack Analysis

If the attacker is able to identify two equal ciphertext blocks \( (C_a \text{ and } C_b) \) on the hard disk, he can compute the following equation.

\[
P_a \oplus C_{a-1} = P_b \oplus C_{b-1}
\]

This equation translates to:

\[
C_{a-1} \oplus C_{b-1} = P_a \oplus P_b
\]

An attacker with access to the encrypted disk knows the left hand-side of the equation. The right hand-side of the equation might be solvable if they have access background information of possible plaintexts. For example, if an attacker happens to know that one of the plaintext is a NULL block, the other plaintext can be recovered rather quickly.

To our understanding, this does not affect AES-XTS in the same manner. With the ECB-like behaviour of this mode, it will probably leak whether two ciphertext blocks represent the same plaintext block and how often a plaintext block occurs [54].

4.3.3.3 Countermeasures

The solution to prevent this kind of attack would require migrating to another block cipher mode like EME, LRW, or CMC [15]. The solution could also be AES-XTS, but its ECB-like behaviour has other issues.

4.3.3.4 Attack Surface

Table 4.3 gives an overview of the attack surface for this vulnerability. The attack is mounted against the encrypted data, therefore the attack scenario is Device-OFF and if the bootloader is locked, an adversary is restricted from accessing the storage. As both versions use this block cipher mode, both are vulnerable to this kind of attacks.

<table>
<thead>
<tr>
<th>Generic Countermeasures</th>
<th>Attack Scenarios</th>
<th>Android</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locked Bootloader†</td>
<td>Disabled</td>
<td>ADM</td>
</tr>
<tr>
<td>adb</td>
<td>Device-OFF</td>
<td>Device-ON</td>
</tr>
<tr>
<td></td>
<td>4.4.4 5.0</td>
<td></td>
</tr>
<tr>
<td>✓</td>
<td>–</td>
<td>✓</td>
</tr>
</tbody>
</table>

† access via hardware-layer possible

Table 4.3: Attack Surface: Content Leak Attack
4.4 Attacks on the Implementation

Having discussed the attacks against the cryptographic foundation of Android’s FDE, we will now focus on the implementation of this feature.

4.4.1 Online Exhaustive Password Search Attack

4.4.1.1 Introduction

The concept of an Exhaustive Password Search Attack (also brute-force attack) is to choose a password from all possible passwords and then test whether it can be used to authenticate or not. In the simplest form, each possible value is tested after another while progressing linearly through the password-space. The term Online means that the password is entered on the system the same way as a user would enter a password.

In Android, we can launch this attack against the Start-Up prompt during boot (see Section 3.2.3) or against the Screen-Unlock prompt when the device is running. Attacking the former would suit the Device-OFF and the latter, the Device-ON attack model.

Our first experiments showed that countermeasures are in place to slow down the brute-force attempt (see Table 4.6 on page 30).

In general, the attacker would have to manually enter a large amount of passwords one after another. This is cumbersome, error-prone and especially challenging for concentration as the attack can take many hours. In order to prevent mistyping and keep the process as efficient as possible we developed a script, online_bruteforce.py (see Appendix A), to perform the attack. The duration results will be used as benchmark values in comparison with an Offline (Section 4.4.2) and Semi-Offline (Section 4.4.3) attack variant.

4.4.1.2 Attack Script

Android smartphones can be commanded via the adb tool. The Android command line program input can be used to simulate user interaction over keys or touchscreen. It offers a great variety of options and arguments and is very versatile.

The first option we used was tap, which takes two screen coordinates as arguments (e.g. input tap 340 1040). It simulates a finger pressing the touchscreen. We have mapped out the number pad of the Nexus 6 and instructed the script to type a four digit PIN, followed by pressing the enter key on the screen. However, this solution is quite slow and takes even longer than entering a PIN by hand. Of course, over a large number of candidate values this time difference would fade and the automation would still offer more accuracy by removing human error. Despite the timing disadvantage, tap has some more issues. First, we would have to map out the number pad for each device. This is an issue every solution working by entering the password via screen share. Secondly, the time and complexity demands are higher for alphanumerical passwords.

The next option we tested was text. It takes a string as argument which will be written in a focused input textfield on screen. The Screen-Unlock prompt is a textfield that can be filled using text. Therefore the script was improved to work with this option. Focusing the textfield can be achieved by using another input option, keyevent, with the argument KEYCODE_ENTER. This solution eliminated all disadvantages of tap.
Table 4.4 shows the results of a comparison between the two methods. We measured the average duration per PIN attempt for 100 entries. By changing to text, we increased the efficiency of our brute-force script by over 80%.

<table>
<thead>
<tr>
<th>Method</th>
<th>Nexus 4</th>
<th>Nexus 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>input tap</td>
<td>5.1 s</td>
<td>3.3 s</td>
</tr>
<tr>
<td>input text</td>
<td>2.8 s</td>
<td>1.8 s</td>
</tr>
</tbody>
</table>

Table 4.4: Average PIN Input Duration per Attempt

Essential for automating a brute-force attack is the detection of a successful attempt. Detecting an unlocked screen appears easier than it is in reality. One could record the screen and look for indication of a successful attempt or monitor the logging facility for certain strings. The latter was our first solution, but it proved to be too slow and we had to clear the log before testing a new candidate or else the script failed. Especially the last point caused us to look for a better method.

Our final and novel solution calls the command-line program `dumpsys`. When the screen is locked, the output, in both Android versions, will contain the string `mLockScreenShown`, otherwise that string is not present.

The last aspect to consider is the timeout which is used to slow down this kind of attacks (see Section 4.4.1.4). We included a sleep for 30 seconds after every fifth failed attempt.

### 4.4.1.3 Attack Analysis

We ran the attack script on the Nexus 6. The results for three different PINs configured are shown in Table 4.5. With the values from Table 4.4 above, we can estimate that for the Nexus 4 the attack duration increases approximately by one second per candidate value.

<table>
<thead>
<tr>
<th>PIN</th>
<th>Number of Timeouts</th>
<th>Duration of Attack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1986</td>
<td>397</td>
<td>16 163 s (~4 h 30 min)</td>
</tr>
<tr>
<td>5000</td>
<td>1 250</td>
<td>40 826 s (~11 h 20 min)</td>
</tr>
<tr>
<td>9999</td>
<td>2 000</td>
<td>81 549 s (~22 h 40 min)</td>
</tr>
</tbody>
</table>

Table 4.5: Nexus 6 Online Exhaustive Password Search Attack

Although we improved the efficiency for each digit entry, the script is only a Proof-of-Concept (PoC) and searches linearly through the keyspace. This is suitable for the purpose of determining the possibility of the attack. With a brute-force strategy more complex than linear search, further improvements are feasible. For example, testing possible values in the form of a dictionary first and then dividing the password space and running each individually.

Additionally, our script handles only PINs but can be extended to alphanumerical passwords easily, since those were the only two screen lock methods available in Android 4.4.4 with FDE, which would be suitable. However, since Android 5.0 also supports the pattern screen lock, another solution would be needed to cover patterns as well. An adb-based solution is possible through the `swipe` option of `input`. The dots used to draw the pattern on
the display are at the same position as the numbers in the number pad, which is why this solution shares the same disadvantage as tap in terms of number pad mapping.

Our solution requires adb to be active and the host authenticated in order to be successful. However, other solutions exist that do not pose these requirements. Android smartphones support the On-The-Go (OTG) protocol. OTG allows to attach a peripheral Universal Serial Bus (USB) device to a smartphone and use it as such, hence the smartphone acts as a USB host. We tested if OTG is active on a locked screen and were able to authenticate using an attached regular computer mouse. A solution would therefore need two elements. First, a programmable device that can act as a mouse and enter the password, and second, a module that can recognize the unlocked screen (e.g. a camera).

An alternative approach which does not need connecting a peripheral device to the smartphone was presented at DEFCON 21. Engler and Vines demonstrated their electromechanical solution [13], a delta robot that presses buttons, taps on the screen and recognizes screen changes.

**4.4.1.4 Countermeasures**

**Identified Specific Countermeasures**

Table 4.6 shows the different timeout and remediation actions we encountered during our analysis.

<table>
<thead>
<tr>
<th></th>
<th>Nexus 4</th>
<th>Nexus 6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Start-Up</td>
<td>Screen-Unkock</td>
</tr>
<tr>
<td>Timeout Threshold</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>Timeout Duration</td>
<td>30 s</td>
<td>30 s</td>
</tr>
<tr>
<td>Action Threshold</td>
<td>30</td>
<td>&gt;100</td>
</tr>
<tr>
<td>Action</td>
<td>Failed Wipe</td>
<td>None</td>
</tr>
</tbody>
</table>

**Table 4.6: Countermeasure Identification of Online Exhaustive Password Search Attack**

The two password prompts, Start-Up and Screen-Unkock, differ in certain aspects but the behaviour does not depend on the OS version itself. Both have a 30 second timeout, one is activated after ten and the other after five failed attempts.

One difference can be noted after exceeding the action threshold (number of failed attempts). We flashed a custom recovery image on our Nexus 4 and are thus not able to verify if the wipe would have been successful with the original recovery. We were simply able to restart the smartphone before the wipe command was issued.

Of greater interest is the difference between the two prompts themselves. While the first one is used solely for the FDE feature, the latter is mainly used to serve as a user authentication handler when the Android framework runs.

It is understandable that the device does not issue a wipe after a few failed screen unlock attempts. However, even after fully brute-forcing 10000 PINs and 2000 timeouts, there was no penalising action at all. We have to assume that the Screen-Unkock prompt has no additional countermeasure to the timeout.
Countermeasure Improvements

The thresholds, timeout duration and penalising action for the Start-Up prompt are considerably reasonable.

More concerning is the configuration for the Screen-Unlock prompt. We believe it would be suitable to increase the timeout from time to time. For example, plus 30 seconds after every 10th failed attempt. Another countermeasure could be a penal action similar to the Start-Up prompt. In combination with an increasing timeout, this could be done after the 50th failed attempt. Furthermore, we believe it would be a generally useful feature if users could choose the timeout duration, threshold values and penal action themselves or at least were given a selection to choose from. Such configuration options are often present for administrators in mobile management software for Bring Your Own Device (BYOD) programmes in enterprises.

The problem of large threshold is the possibility that users may choose simpler passwords (e.g. a PIN such as 0000 or their birth year). Entering complex, strong passwords is cumbersome, error-prone and time-consuming on a smartphone display. However, if the threshold for a penalising action is too large or non-existent an attacker is likely to succeed at some point.

4.4.1.5 Attack Surface

Both attack scenarios are relevant, although Device-OFF is with 30 attempts clearly limited in contrast to Device-ON. Two generic countermeasures have mitigating effects on the attack limiting an attacker’s options. In the described variant of the attack, Disabled adb already counters the approach. Both Android versions are vulnerable, as we have seen that both versions employ the same specific countermeasures.

<table>
<thead>
<tr>
<th>Generic Countermeasures</th>
<th>Attack Scenarios</th>
<th>Android</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locked Bootloader</td>
<td>Disabled adb</td>
<td>ADM†</td>
</tr>
<tr>
<td></td>
<td>Device-OFF</td>
<td>Device-ON</td>
</tr>
</tbody>
</table>

† device needs network connectivity

|                     | ✓              | ✓              | ✓              | ✓              | ✓              | ✓              | ✓              |

Table 4.7: Attack Surface: Online Exhaustive Password Search Attack

4.4.2 Offline Exhaustive Password Search Attack

4.4.2.1 Introduction

This section will discuss the Offline approach to an Exhaustive Password Search Attack, in which timeout countermeasures from the Online approach have no effect.

Offline brute-forcing means to run the attack on a different attack host than the target. The host can have sufficiently more computational power or memory (see Appendix C).

The goal remains the same as with the Online approach: recover the password used in KEK generation and for lock-screen authentication. The attack is only possible against devices running Android pre 5.0, which are over 90% of the devices (see Section 2.1), since these do not cryptographically bind the KEK to the device (see Section 3.2.1). The attack was therefore carried out against our Nexus 4.
The device is only needed for as long as the partitions required (see list below) are not copied from the smartphone. Thus both attack scenarios are plausible.

1. The metadata partition, containing the crypto footer
2. The userdata partition

Different partitioning methodologies are discussed in Appendix B.

### 4.4.2.2 Attack Analysis

Santoku Linux ships with the PoC script brute-force_stdcrypto.py. The script was initially developed by Cannon and Bradfrod. Several authors have contributed since, such as Nikolay Elenkov whose branch was chosen to fork for our developments [5]. Elenkov accepted our improvements and created a pull request to the Santoku Linux repository to submit our new code.

The script parses the crypto footer from the metadata partition and then runs a linear brute-force attack through the PIN-space. Each candidate value runs through scrypt with the parameters stored in the crypto footer (Android 4.4.4 defaults: \( N = 32768 \), \( r = 8 \), \( p = 2 \)). The derived candidate key is used to decrypt the encrypted master key, which in turn is then used to decrypt the first 32 bytes of the encrypted userdata partition. The script reports successful PIN recovery if the higher 16 bytes are all \texttt{NULL}. The lower or first 16 bytes are random garbage, because the AES-CBC decryption context is not initiated with the correct ESSIV value.

The rationale behind the 16-byte NULL is the ext4 file system (format of userdata). The ext4 disk layout has a 1024-byte block NULL before the ext4 superblock [31]. However, there is no certainty that this is the case, as the padding block can have other values as well.

We improved the detection procedure of the script and introduced a check for the ext4 magic signature, 0xEF53 at offset 0x38 within the ext4 superblock [31]. With the 1024-byte padding, this results in the offset 0x1080 from the beginning of the partition. Hence, the script needs to decrypt 1088 bytes from userdata to be able to check for the signature.

Table 4.8 shows the results of an Offline Exhaustive Password Search Attack with the same PINs as in Section 4.4.1. The attack is clearly an improvement over the Online approach. Searching through the complete four digit PIN-space took less than an hour, compared to over 22 hours of the aforementioned Online attack.

<table>
<thead>
<tr>
<th>PIN</th>
<th>Duration of Attack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1986</td>
<td>609 s (~10 min)</td>
</tr>
<tr>
<td>5000</td>
<td>1543 s (~25 min)</td>
</tr>
<tr>
<td>9999</td>
<td>3068 s (~51 min)</td>
</tr>
</tbody>
</table>

**Table 4.8:** Offline Exhaustive Password Search Attack

### 4.4.2.3 Countermeasure Improvements

Google improved the KEK generation with a hardware-binding (see Section 3.2.1) to prevent Offline Exhaustive Password Search Attacks.

We will discuss potential improves of the countermeasures in Section 4.4.3.4 after the discussion of the Semi-Offline attacks in Section 4.4.3.
4.4.2.4 Attack Surface

It is difficult to define a clear attack surface in this occasion, as shown in Table 4.9. The only challenge for an attacker is to image the partitions. Appendix B describes various methods which fall under the attack scenario Device-ON for which all three generic countermeasures would apply. However, the Device-OFF scenario is also plausible, as an attacker could for example image the partition via JTAG.

<table>
<thead>
<tr>
<th>Generic Countermeasures</th>
<th>Attack Scenarios</th>
<th>Android</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locked</td>
<td>Disabled</td>
<td>ADM†</td>
</tr>
<tr>
<td>Bootloader‡</td>
<td>adb</td>
<td>Device-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ON</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

‡ access via hardware-layer possible † device needs network connectivity

Table 4.9: Attack Surface: Offline Exhaustive Password Search Attack

4.4.3 Semi-Offline Exhaustive Password Search Attack

4.4.3.1 Introduction

In Section 3.2 we discussed the implementation of FDE in version 5.0. There was a significant change in the way the KEK is derived from the password. The attack approach of Section 4.4.2 does not work any more. To recap, the reason is the newly hardware-backed keymaster that is employed to sign an intermediate value of the KEK generation.

Within our analysis of the source code and the previously discussed Offline attack, we drew the idea of a protocol to brute-force Android 5.0’s encrypted master key. The main issue was overcoming the hardware-backed keystore signature. In the analysis of the crypto footer, we could understand that the cryptographic material used for the signature is also stored within the footer. This is the public and encrypted private RSA exponent and the RSA modulus.

Hardware-backing could be overcome if the encrypted private RSA exponent was recovered. We have no insights on how the keymaster encrypts this. However, we figured that if the key within the keymaster is fixed, we can make use of it.

The adversary is in possession of the device and after he has imaged the partitions there is no need to keep the device untouched. We thus describe here our attack that uses the device to recover the master key and call this brute-force attack approach Semi-Offline.

4.4.3.2 The Attack Application

In the KEK generation, only the RSA signing operation needs the keymaster and thus the device. The two scrypt computations can run externally on a more powerful host with more memory (see Appendix C).

In the process of analysing this attack, we developed an application to understand how Android handles the KEK generation and parsed the crypto footer on a device to verify that our computations were correct. The foundation of that application was cryptfs.c [46] from the AOSP. We analysed the output at different stages of the KEK generation and key management.
With deep understanding of the connection between different source files and functions, we aimed to develop a PoC application (see A) that provides the following functionalities:

1. Initialize the keymaster with a previously imaged key blob
2. Retrieve the intermediate value after first scrypt operation
3. Sign the intermediate value with the keymaster
4. Return the signature
5. On an external host, generate candidate passwords
6. On an external host, calculate the scrypt operation
7. On an external host, decrypt the master key
8. On an external host, test the master key for correctness

The result was a client-server application, with the server being run on the smartphone and listening on a port to retrieve the information. The communication between client and server is visualised in Figure 4.1. The first message will notify the length of the base64 encoded keymaster key blob following in the second message. Afterwards, the server will listen for intermediate values, issue the signing and send back the signature. The client then serves for points 5. to 8. of the above list.

![Diagram](Client-Server Communication)

This application follows the same PoC approach as the two previous discussed attack scripts in Sections 4.4.1 and 4.4.2. The brute-force runs linearly through the password-space. Also no parallelization is employed.

We only implemented PIN brute-forcing. Alphanumerical passwords and patterns would require further code but could be extended.

### 4.4.3.3 Attack Analysis

The application successfully allowed recovering the master key on an Android 5.0. Table 4.10 shows the result of our experiments.
We have wiped the Nexus 6 and could run the attack with a previously captured crypto footer. This leads to the conclusion that the keymaster key protecting the private RSA exponent used for signing is not destroyed in the wipe process. We also confirmed this assumption by re-visiting the source code.

<table>
<thead>
<tr>
<th>PIN</th>
<th>Duration of Attack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1986</td>
<td>1535 s (~26 min)</td>
</tr>
<tr>
<td>5000</td>
<td>3872 s (~1 h 5 min)</td>
</tr>
<tr>
<td>9999</td>
<td>7622 s (~2 h 8 min)</td>
</tr>
</tbody>
</table>

Table 4.10: Nexus 6 Semi-Offline Exhaustive Password Search Attack

Table 4.11 shows a comparison of the three attacks targeting the largest possible four digit PIN. The Online attack is the slowest, which was no surprise given that a single PIN entry took almost 2 seconds. Also not very surprising is that the Offline attack is quicker than the Semi-Offline attack. After all, the KEK generation prior to Android 5.0 only used one slow KDF function. The newer approach takes two KDF function calls, plus a public-key operation. Additionally, the attack has a certain delay because of the network communication between the client-server applications.

It is interesting to note the difference compared with the Offline attack, which is supposed to be mitigated in Android 5.0. The Semi-Offline attack takes almost two hours longer. Focusing on a single PIN, the Semi-Offline attack takes almost five times longer than the Offline attack. Nevertheless, considering the higher complexity in key derivation, the Semi-Offline attack procedure overall is a success. Especially in comparison with the Online attack which takes more than ten times longer. We certainly demonstrated that another attack vector, more efficient than Online brute-forcing, still exists.

<table>
<thead>
<tr>
<th>Attack Method</th>
<th>PIN 9999</th>
</tr>
</thead>
<tbody>
<tr>
<td>Online</td>
<td>81549 s  (~22 h 40 min)</td>
</tr>
<tr>
<td>Offline</td>
<td>1543 s   (~25 min)</td>
</tr>
<tr>
<td>Semi-Offline</td>
<td>7622 s   (~2 h 8 min)</td>
</tr>
</tbody>
</table>

Table 4.11: Time Comparison of Different Exhaustive Password Search Attack Methods

4.4.3.4 Countermeasure Improvements

Our Semi-Offline attack is an adoption of the Offline attack (see Section 4.4.2). A key issue affecting both attacks is the attacker’s ability to copy a partition from the smartphone. Appendix B describes several approaches, most of which can be prevented by generic countermeasures. However, unsoldering flash-memory or accessing the device via JTAG interface is not mitigated with these countermeasures.

We do not have a solution that would prevent imaging directly. The keymaster involvement was thought to resolve the Offline brute-force attacks. In reality, this is not the case, as it only increases the attack duration and requirements. We would, however, suggest that the locked bootloader becomes mandatory by definition in the CDD [18]. Our suggestions to improve the countermeasures to mitigate Semi-Offline attacks therefore focus on the hardware-binding and not on imaging prevention.
The Semi-Offline attack was possible because the internal keymaster key used to encrypt the private RSA exponent is static. We therefore believe that key destruction or key change for this internal, static key should be considered. This would increase the attack complexity. An attacker would have to brute-force the keymaster key or the private RSA exponent first. An attacker who is able to do this in a reasonable time is likely able to brute-force the master key directly and skip the KEK derivation entirely.

The key question is: When is the internal keymaster key destroyed? This is rather complex, but we would suggest certainly on every userdata wipe instruction. Hence, an attacker can only run our client-server application on a rooted smartphone without disabled adb or on a smartphone with an unlocked bootloader. It would prevent him from unlocking the bootloader himself.

### 4.4.3.5 Attack Surface

As for Offline, estimating the attack surface for the Semi-Offline attack is not straightforward. The core issue again is imaging the partitions and as discussed in Appendix B it is possible in several forms. The generic countermeasures do not prevent but remediate the attack by increasing the complexity for the adversary.

Table 4.12 shows only Android 5.0 is affected by this attack, since this is the alternative to the Offline approach which Androids pre 5.0 are vulnerable to.

<table>
<thead>
<tr>
<th>Generic Countermeasures</th>
<th>Attack Scenarios</th>
<th>Android</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locked Bootloader‡</td>
<td>Disabled adb</td>
<td>ADM†</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

‡ access via hardware-layer possible † device needs network connectivity

**Table 4.12:** Attack Surface: Semi-Offline Exhaustive Password Search Attack

### 4.4.4 Static Default Password and Master Key

The implementation as of Android 5.0 employs a known password and a static but secret master key. We will first consider each aspect individually and then assess the potential threat posed by their combination.

#### 4.4.4.1 Static Default Password:

The KEK generation, discussed in Section 3.2.1, requires a password. In Section 3.2.4.2, we outlined that in three situations the default password defined in the source code is used.

- Encryption on first boot
- Screen lock method Swipe or None used
- Do not ask for password during device boot

In the first two situations, there is no protection gained by FDE. Anyone who has physical access to the smartphone is in full control. No password is required on start-up, on
4.4. Attacks on the Implementation

4.4. Security Assessment

screen unlocking or when a screen lock is set. This also means that an adversary, in our two defined attack scenarios, can use the PoC from Section 4.4.3 and change the script to use the default password and recover the decrypted master key.

4.4.4.2 Static Key Material

We did so far not cover any key change aspects for the master key, as such functionality is not implemented. The master key is read once from /dev/urandom and then remains static until the smartphone is wiped. Key change is a vital part in the key life-cycle. Users are often advised to change their passwords and keys regularly. Enterprises will most likely define key life-time in their policies.

The absence of such a functionality limits the user’s activity radius. In case of a key compromise he must backup all data, wipe the smartphone and then restore all data. The same procedure must be followed if the policy-defined key life-time is about to expire and demands a key change. We have to assume that in the latter case an exception in the policy will be granted. Considering the key length of 128 bits of a master key, it is unlikely that such keys can be recovered by cryptanalysis today.

4.4.4.3 Combined Analysis

We analysed each aspect individually and already identified potential threats. However, by combining these two issues we can draw a more serious threat scenario.

Let’s assume a user buys a Nexus 6, boots up for the first time and default encryption is started with the default password being used, after the smartphone finishes the encryption and boots, the user does not immediately set a screen lock password.

If now an adversary can access the smartphone, he can image the two partitions (userdata and metadata), adapt the client-server PoC and use the default password to recover the master key.

Next, let’s assume the smartphone is used by its owner who even sets a password to protect against unauthorized physical access.

After some time has passed, the adversary again gains access to the smartphone and only re-images the userdata partition. With the previously imaged crypto footer and recovered decrypted master key, the adversary can decrypt and mount userdata.

4.4.4.4 Countermeasure Improvements

In order to limit the threat of each of these issues or their combination, we propose changes to the current procedures.

We believe the reason behind the default password is twofold. First, to simplify the first boot encryption process and second, it allows screen lock methods Swipe and None.

Probably the weakest solution is to replace the default password with an initial password the user sets on first boot. That password will be stored in the crypto footer. Encrypting this password with the keymaster would little increase the complexity to retrieve that password. An attacker could use a similar procedure as in our Semi-Offline attack to retrieve this password.

From a security perspective, a better solution is to require the user to set a screen lock on the first boot before the master key is generated. It could be that this solution was
rejected by Google, as this probably needs more changes to the code base. If we recall, the encryption process starts without the full Android framework started. Another drawback of this solution is that users have to set and remember a password. Some users might not appreciate that.

In terms of the static master key, we think implementing a key change functionality would be a desired option. LUKS, the Linux key management implementation for FDE, offers this functionality via the tool `cryptsetup-reencrypt`. Potential issues with this concept are the duration of re-encrypting a large set of data or the procedure itself. This would require to create another dm-crypt virtual device, read from the encrypted partition (equals a decryption operation) and then write to the new device (equals an encryption operation). A problem arises if the smartphone has not enough capacity to store the data twice for as long as the re-encryption takes place.

Considering both aspects, the ideal solution (ignoring all drawbacks) would be a default password and key change functionality combined. Every time the password changes from or to the default password, a key change is initiated. In all three of the above situations, users should be given a notification similar to Figure 3.3 on page 18 that FDE is not effectively employed to protect data. However, because of the key change, using a default password does not pose a threat to the master key any more.

### 4.4.4.5 Attack Surface

Table 4.13 shows what default countermeasures mitigate the threat of these weaknesses, which attack scenarios are considerable and what OS version is affected.

<table>
<thead>
<tr>
<th>Generic Countermeasures</th>
<th>Attack Scenarios</th>
<th>Android</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locked</td>
<td>Disabled</td>
<td>ADM†</td>
</tr>
<tr>
<td>Bootloader‡</td>
<td>adb</td>
<td>OFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| ✓                        | ✓                | ✓       | ✓       | ✓       | –       | ✓       |

‡ access via hardware-layer possible † device needs network connectivity

**Table 4.13:** Attack Surface: Static Default Password and Master Key

### 4.4.5 Cold-Boot Attack

#### 4.4.5.1 Introduction

In order to provide transparent encryption of the storage device, the encryption key needs to be present in the system’s main memory (hereafter *memory*). Memory is a volatile storage medium. It was believed that after powering off, the data in memory is lost immediately. In practice, this is not entirely true and Halderman et al. visualized this phenomenon in their paper [29] with the following picture series (see Figure 4.2).
The first image shows the original loaded into memory, the second shows the recovered picture after a shutdown duration of 30 seconds, the third after 60 seconds and the last after 5 minutes. The duration for which data remains in memory can be increased when the memory module is cooled down. This increases the attack window on the volatile data and a variety of imaging procedures were suggested, from PXE network boot to the boot from an external hard drive.

In their paper Halderman et al. demonstrate an attack on AES encrypted hard disks, encrypted using different software products, including Linux’s dm-crypt which is also used in Android. With their program *keyfind*, the authors were able to recover an AES-128 key. The vulnerability at the core of Cold-boot Attacks is the key material that has to be kept in memory to allow the transparent encryption and decryption of disk operations.

### 4.4.5.2 Attack Analysis

That the concept of the Cold-Boot attack also works on Android smartphones with FDE was demonstrated by Müller, Spreitzenbarth and Freiling in their study [37] with Forensic Recovery of Scrambled Telephones (FROST) targeting a Samsung Galaxy Nexus smartphone.

FROST is an Android recovery image that can be flashed onto the smartphone after physical access is established. The authors outlined that they were also able to recover the key after the device had undergone wiping. The question is therefore whether the encrypted userdata partition can be extracted using other methods before the wipe happens (see Appendix B).

### 4.4.5.3 Countermeasures

ARMORED [25] is an approach to counter Cold-Boot Attacks by holding the encryption key in CPU registers rather than memory. The authors tested it against FROST and successfully prevented the recovery of AES key material. The restriction of ARMORED is that it only allows to hold a 128-bit key, suitable for current Android FDE with AES-CBC but not good enough for future developments.

In [38], Nilsson et al. present their approach to counter Cold-Boot Attacks. The authors utilize the ARM CPU’s debug registers to store the encryption key which allows to use a 256-bit key.
4.4.5.4 Attack Surface

The attack surface is visualised in Table 4.14. An adversary would need an unlocked bootloader and the encryption key must be held active in memory before freezing. We could not identify any difference in regard to Android handling the encryption key differently in memory and thus believe that both versions are vulnerable to this attack.

<table>
<thead>
<tr>
<th>Generic Countermeasures</th>
<th>Attack Scenarios</th>
<th>Android</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locked Bootloader‡</td>
<td>Disabled</td>
<td>ADM†</td>
</tr>
<tr>
<td>adb</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

‡ access via hardware-layer possible † device needs network connectivity

Table 4.14: Attack Surface: Cold-Boot Attack

4.4.6 Evil-Maid Attack

4.4.6.1 Introduction

The concept of the Evil-Maid Attack [56] targets a travelling person leaving their notebook with FDE behind in a hotel room. In the time of the person’s absence, an adversary, for example an evil maid, gets access to the room and therefore the system.

A full-disk encrypted system will require a certain part of the hard disk to be unencrypted in order to boot the system. This can for example be a notebook’s boot partition or in the case of Android any partition other than the encrypted userdata. With physical access to the unencrypted data location, the evil maid is in total control and can install a keylogging malware and leave the room. When the owner later returns and starts his system, his key stroke are logged and stored in a log file on the unencrypted location. The evil maid then regains access to the room with the system, when the person left on a second occasion. She can now recover the user password to unlock the encrypted hard disk.

4.4.6.2 Attack Analysis

In the case of an Android system, the attack differs in certain aspects compared to a notebook. Installing malware is more complex than booting from a malicious USB stick and installing the software in the unencrypted partition. The Android approach would be flashing a customized system partition (requires unlocked bootloader). EvilDroid [24] is such a customized Android kernel which includes the capability of keystroke logging.

The authors describe in [24] another attack, suitable for situations where the smartphone has a locked bootloader and cannot be flashed. This approach is more intrusive, as the target will realize that they have been victimized.

After identifying the victim’s smartphone model, the evil maid would prepare an exact duplicate model with EvilDroid and an Android application that starts and displays first during the boot sequence. The application simulates the Start-Up prompt to pass as genuine Android. When the victim enters his password, the application records the value and sends it to the adversary.
The duplicate device with the malicious system and application will be swapped with the target’s smartphone while he is absent. On return, the target will enter the password, which will be sent to the evil-maid who is in possession of the encrypted smartphone. The evil maid can enter the password sent from the victim and access the decrypted storage.

In this attack, a core factor is that the victim leaves the encrypted smartphone in a hotel room. While this is fairly possible for larger devices such as notebooks or tablets, it is debatable whether this holds true for smartphones as well or if they are less likely to be left behind.

4.4.6.3 Countermeasure Improvements

The first Evil-Maid Attack is covered by the existing countermeasures. The second approach works because password prompts do not offer any personalization feature [24, p. 92]. With a personalized password prompt at boot, the attacker would need either more inside knowledge about his target or the victim would be able to identify the attack.

We believe another potential countermeasure for the second attack approach could be the use of a Trusted Path to invoke the authentication, similar to the sequence Ctrl+Alt+Del for Windows OS. The Orange Book [8] defines Trusted Path as: “A mechanism by which a person at a terminal can communicate directly with the Trusted Computing Base. This mechanism can only be activated by the person or the Trusted Computing Base and cannot be imitated by untrusted software”.

4.4.6.4 Attack Surface

Evil-Maid Attacks can target both Android versions and with the two attack approaches in [24] both attack scenarios are possible as well. A disabled adb does not have any influence in either of the attack scenarios.

<table>
<thead>
<tr>
<th>Generic Countermeasures</th>
<th>Attack Scenarios</th>
<th>Android</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locked</td>
<td>Disabled</td>
<td>ADM†</td>
</tr>
<tr>
<td>Bootloader‡</td>
<td>adb</td>
<td>Device-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OFF</td>
</tr>
<tr>
<td>✓</td>
<td>–</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 4.15: Attack Surface: Evil-Maid Attack

4.4.7 Potential Weaknesses

We found during our analysis several aspects that could lead to potentially exploitable vulnerabilities. A thorough practical assessment of these issues would require further resources and knowledge beyond our current means. We were therefore not able to cover them deeper in the current paper, but believe they are worth investigating in future analysis.

4.4.7.1 Predefined Cryptographic Primitives

End-users cannot influence the choice of cryptographic primitives or parameters used to encrypt the disk. This appears to be an attempt to resolve the fundamental dilemma of
4.4. Attacks on the Implementation

security, defined in [17] as “Security-unaware users have specific security requirements but usually no security expertise”. However, this leaves out users with advanced knowledge. Only developers can decide what block cipher to use in combination with which block cipher mode and key length. Users who would like to change this need to change the source code and recompile Android themselves.

It is understandable from Google’s perspective to keep security as simple as possible, while offering default settings that suit a large number of customers. If a user or enterprise, however, defines a policy to use AES with 256-bit keys or use XTS instead of CBC, an exception for Android smartphones must be defined.

We believe that users could be given the current configuration as default but provided with other ciphers and modes in an advanced configuration menu. This means that the code for FDE increases and the process of encrypting a smartphone requires more user interaction compared to now. We admit that any new line of code additionally points a risk to contain a new vulnerability. Moreover, both Android versions share this property.

<table>
<thead>
<tr>
<th>Generic Countermeasures</th>
<th>Attack Scenarios</th>
<th>Android</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locked Bootloader</td>
<td>Locked</td>
<td>4.4.4</td>
</tr>
<tr>
<td>Disabled adb</td>
<td>Device-OFF</td>
<td>5.0</td>
</tr>
<tr>
<td>ADM</td>
<td>Device-ON</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.16: Potential Attack Surface: Predefined Cryptographic Primitives

4.4.7.2 Lack of Entropy in Master Key

The master key is created by reading from /dev/urandom (see Section 3.2.1). This device reads from the same Pseudo-Random Number Generator (PRNG) as /dev/random. Their difference is that urandom does not block if it runs out of entropy [32], which could lead in less randomized values than believed. To preserve the entropy pool, the latest value is saved in a file on shutdown and used to initialize urandom at next boot. In Android that file is /data/system/entropy.dat. The value is copied to /dev/urandom with a command in init.rc [51, line 239].

It is therefore clear that after reboot a device should have suitable random entropy. However, in case the device is encrypted on first boot, this file is non-existent and the entropy pools has to be built up. FDE is not be the only process that will ask for random values on start-up. In fact, every process will read from /dev/urandom to create a random canary value for stack guarding [9].

To what extent this affects the randomness of the master key and salt is hard to foretell. If the values are predictable, we see a potential advantage for an attacker. Instead of searching for the valid password to recover the master key, an attacker could directly target the 128-bit AES key. This could subvert the need to use the keymaster in the Semi-Offline attack and allow to run an Offline brute-force attack targeting the master key instead of the password.

A thorough analysis would require to customize the vold daemon, specifically the function create_encrypted_random_key in [46, line 1413]. When the key and salt are read from urandom, the value should be logged for later analysis. The process of starting with no entropy and creating the master key should be repeatedly performed and after each master key creation the value should be saved externally.
Even though the SDK emulator does not encrypt the storage, it might be preferable to have the emulator’s system image changed so that it at least aborts the process after creating the key material. Alternatively, a physical device can be used. Further changes are that the default encryption process starts on every boot and does not write the entropy data file, preventing urandom’s initialisation. The drawback of this solution is the life-time reduction of the flash-memory due to many rewrites.

For this potential weakness, the generic countermeasures cannot offer protection (see Table 4.17). In terms of the attack scenario, we believe it is more suitable for the Device-OFF since it targets the master key directly. In terms of the Android version, both would be affected as the implementation does not differ here.

<table>
<thead>
<tr>
<th>Generic Countermeasures</th>
<th>Attack Scenarios</th>
<th>Android</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locked Bootloader</td>
<td>Disabled adb</td>
<td>ADM</td>
</tr>
<tr>
<td></td>
<td>Device-OFF</td>
<td>Device-ON</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 4.17: Potential Attack Surface: Lack of Entropy in Master Key

4.4.7.3 Unencrypted Data Residue on Flash-Memory

Flash arranges memory capacity in sectors, blocks and pages, whereas multiple pages form a sector. A great difference in contrast with magnetic hard-disks is that data erasure only happens on a block-by-block basis, while writes can happen on a sector basis. However, a sector can only be written to an erased block, but flash-memory chips have only a limited erase-cycle per lifetime. This has two consequences: first, in-place update of data is not possible and thus the updated information is written to a new location while the previous is marked as deleted, but the data remains. Secondly, memory controllers introduce logic to spread programming commands (write/erase) over the full storage capacity to rewrite blocks evenly.

Memory logic for flash-storage is more complex than for legacy magnetic storage. In [61], the authors describe the observation of up to 16 stale copies of 1000 small files created on a solid state disk. The reason for these copies was the memory logic that created them during garbage collection. The logic responsible for these algorithms is called File Translation Layer (FTL).

The study [57] by Simon and Anderson focused on the secure erase commands from embedded MultiMedia Card (eMMC) used in Android factory resets. The authors discovered that there is a great discrepancy between devices of different manufacturers, whether the factory reset is successful or not. One reason for this result is seen in the non-presence of embedded MultiMedia Card (eMMC) commands at the wipe commands. For the Nexus 4, the authors even discovered that the last 16 KB were not sanitized at all [57].

Another important factor for flash-memory is that the FTL may differ from one chip manufacturer to another, another issue of fragmentation (see Section 2.1.2).

The consequences for FDE are as follows. A device used without encryption for any period of time and encrypted at a later time might allow the recovery of sensitive information on a lower level by accessing the flash-chip directly without FTL. Encryption at a later time does not wipe a partition. We believe this is not possible, because the data to be wiped are
the same as those to be encrypted. On the contrary, if the memory is encrypted before first use, all data written on the storage is encrypted. Residue data is thus protected as well.

Table 4.18 indicates the potential attack surface for this weakness.

<table>
<thead>
<tr>
<th>Generic Countermeasures</th>
<th>Attack Scenarios</th>
<th>Android</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locked</td>
<td>Device-OFF</td>
<td>4.4.4</td>
</tr>
<tr>
<td>Disabled</td>
<td>Device-ON</td>
<td>5.0</td>
</tr>
<tr>
<td>adb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADM OFF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADM Device-ON</td>
<td></td>
<td></td>
</tr>
<tr>
<td>only devices without default encryption</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.18: Potential Attack Surface: Unencrypted Data Residue on Flash-Memory

4.4.7.4 Trusted Execution Environment Vulnerabilities

The Trusted Execution Environment (TEE) is an isolated execution environment in System on a Chip (SoC) systems that only runs pre-approved applications and provides services to applications and the host OS in user space. The internals of such a TEE are closed-source and independent researchers often only have the option to reverse-engineer the environment. Dan Rosenberg analysed Qualcomm’s implementation of the ARM TEE, called Qualcomm Secure Execution Environment (QSEE) and found it vulnerable for an integer overflow that can be exploited by an attacker with kernel-level privileges [55, p. 2].

With Android 5.0 and the current CDD [18], TEE became a core element for Android. The keymaster application used in the KEK generation runs in this execution environment. It is the only non-public element in FDE and vulnerabilities such as the one found by Rosenberg are certainly an issue for such an important element. The perfect software does not exist and security researchers, as well as adversaries, will continue to analyse it.

Table 4.19 visualises our estimation for the potential attack surface. It certainly will not target Android 4.4.4, since this version does not use the keymaster in FDE. Moreover, we estimate it targets primarily the Device-ON scenario, as only in a powered on status the TEE applications are accessible.

<table>
<thead>
<tr>
<th>Generic Countermeasures</th>
<th>Attack Scenarios</th>
<th>Android</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locked</td>
<td>Device-OFF</td>
<td>4.4.4</td>
</tr>
<tr>
<td>Disabled</td>
<td>Device-ON</td>
<td>5.0</td>
</tr>
<tr>
<td>adb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADM OFF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADM Device-ON</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.19: Potential Attack Surface: Trusted Execution Environment Vulnerabilities
4.5 Bypass Lock-Screen by Attacking Smart Lock

Lock-screen authentication is essential for FDE. It is the password users enter on start-up and is used to derive the KEK, which protects the master key used for encrypting the disk (see Section 3.2.1). We have seen in Section 3.2.4.2 that a user can opt out and only set the password for screen unlocking. In that case the only means preventing an adversary from accessing the data is lock-screen authentication. Then, due to the transparent encryption, each read from the encrypted partition is decrypted on run-time.

We therefore include in our security assessment the analysis of Smart Lock. This feature builds on top of trust agents and was introduced with Android 5.0. The core functionality is an authorized bypassing of the lock-screen by presenting a trust agent. The only relevant attack scenario is therefore Device-ON.

Features that are designed to bypass an authentication in specific situations are a potential threat to the underlying security. If an attacker is able to misuse Smart Lock by either manipulating the trust agent or presenting the correct trust agent, he will bypass the authentication too.

Smart Lock uses four different trust agent categories: Trusted Places, Trusted Device, Facial Recognition and Accelerometer. We are going to discuss their functionality and assess how they can be misused by an unauthorized third party to bypass the lock-screen. The consequence of a successful attack is that an adversary gains access to the transparently decrypted user data partition but is not in possession of the password. This gives an adversary access to the device on software level. The attacker cannot change the screen lock method, since doing so would require to authenticate with the set method first.

4.5.1 Trusted Places

4.5.1.1 Introduction

This category uses Android’s location service, which is either based on Wi-Fi, mobile network, Global Positioning System (GPS) or a combination of these. Trusted Places is only available if the device is configured with a Google account. With an internet connection on a smartphone, this trust agent will access the home location configured in the account and advertise it as the first option to choose. Different locations can be chosen by the user, where the trusted location is visualized as a circle on the map.

Choosing a trusted place as a trust anchor is rather imprecise. The location service of smartphones is not very accurate, as a navigation test with a smartphone showed. The support documentation [23] states that “it could keep your device unlocked within a radius of up to 80 meters”. For other trust anchors, like Bluetooth, the user is notified on inaccuracies upon activation.

4.5.1.2 Misuse Analysis

During our testing, we used the home location of the Google Maps application and specified two other locations using the option Add trusted place. We observed that the device was unlockable in a radius of more than 80 meters. In ten repetitions of the test, we could identify that this only holds true for about 60 seconds. Afterwards, we were again asked for the lock-screen password. It therefore appears that the
location is not refreshed at the time the Screen-Unlock prompt is displayed but cached for one minute.

Our test reveals an issue that is covered neither in the support documentation nor in a warning or notification message on the device. There is not just a known inaccuracy in the location service itself, but also an additional inaccuracy because of the delayed location actualisation. This can lead to a significantly larger radius.

Considering a 1.5 meter per second walking speed [34] and the identified 60 seconds caching time, the diameter possibly increases by another 90 meters. Figure 4.3 visualizes this, where the inner circle is the initial trusted place and the outer circle visualizes the Extended Trusted Place, available for 60 seconds with last device localising on the inner circle line.

The estimation of 90 m is only theoretical, as it would require knowledge of the furthest edge of a trusted area and before leaving it, that the location is actualised a last time. Nevertheless it has a practical element as shown in our experiment.

Additionally, we identified that Trusted Places will timeout after four hours without an unlock attempt. Afterwards the user must enter the password again. Before that, any unauthorized third party which happens to have access to the smartphone while it is present in the trusted place, or in the extended trusted place, can bypass the lock-screen authentication and gain OS-level access and thus observe, modify or delete user data.

4.5.2 Trusted Devices

4.5.2.1 Introduction

The trust anchor in this category is either a passive Near Field Communication (NFC) tag or a Bluetooth device. An active NFC device, such as another smartphone, could not be paired in our experiments. Compared to Trusted Places and Bluetooth, the distance between a smartphone and the NFC trust anchor is the smallest of all three.

For Bluetooth trust anchors, users are notified upon activation that connecting devices
which are often kept in physical proximity of the smartphone should be avoided. Furthermore, different Bluetooth versions have different security properties. Such devices can communicate with each other but only the lowest security level is provided, which could lead to a spoofed connection. Lastly, Bluetooth connections can reach up to 100 meters.

4.5.2.2 Misuse Analysis

In our experiments, we used a card containing a passive NFC tag. Registering is straightforward and recognition by the smartphone is notified with an audio signal tone. We had to move the smartphone over the tag at short distance and it worked best when we swiped the tag at the back of our device.

An unauthorized third party could misuse the NFC trust agent if he is also in physical possession of the tag. Otherwise, if he can clone the tag, probably after observing it once before, he could impersonate the original trust anchor. It is therefore important to control access to the tag as well.

The Bluetooth test revealed that between the smartphone and the trust anchor, a permanent connection is required. We paired the Nexus 4 and only after tethering its internet connection to the Nexus 6 we were able to use it as trust anchor.

We could hold the connection outdoors for approximately 55 meters. When the Nexus 4 was inside the building, the connection and thus the trust anchor was lost after about 15 meters. This is less compared to the distance of the trusted place but enough to allow an adversary to get access by entering the proximity of the trust anchor while in possession of the smartphone.

4.5.3 Facial Recognition

4.5.3.1 Introduction

This biometric authentication method is based on Face Unlock. Biometric authentication requires first the registration of a template. Hence, a picture is taken and processed. This might include some picture enhancement steps but certainly the recognition and extraction of distinctive characteristics. These extracted values form a template which is stored in a smartphone and compared against in any subsequent authentication request.

The measurement of biometric characteristics will never reproduce the template exactly. This can be due to several reasons, for example the natural ageing process or environmental changes like lighting. Therefore, successful authentication requires the definition of an acceptance threshold. This defines how much the current sample can differ from the original template. Defining a too restrictive threshold will likely reject a valid user’s authentication attempt, whereas a too large threshold would likely allow access to an unauthorised third party with similar characteristics. Upon activation, Android informs the user that this feature is less secure than any screen-locking mechanism.
4.5.3.2 Misuse Analysis

We analysed Facial Recognition in three unlock test scenarios:

1. Unlock with the person registered
2. Unlock with a digital picture of the user
3. Unlock with the person registered, but partially covered face

During our experiments, we observed that Facial Recognition asks the user to authenticate with their password after three failed attempts. The counter only increases if a face was detected. Presenting a blank paper or a scenery picture did not raise the counter.

Scenario 1 In our first test set of 20 tries, we did not change the location or move the body after registration and could successfully authenticate 20 times. We then moved to a dimmer spot and failed in all of the next set’s attempts. In the last set, we repositioned after every attempt and observed that turning the body by $90^\circ$ can make the difference between failure and success. It is very likely that the cause was again a change in lighting.

Scenario 2 We took a picture and displayed it on the computer screen. First, we tried to position the phone and the picture correctly by hand. In a set of 20 attempts, we successfully authenticated once, but failed in all other attempts.

In order to exclude any mistake, we used cardboard to create a primitive holding mechanism for the smartphone. Assuming an attacker has access to a second smartphone of the same model, we used the face registration function to calibrate the smartphone and the picture. In the following test set, we were able to authenticate in every attempt.

We were also able to reverse that scenario by registering the digital picture and successfully authenticate with the person directly. The device was again kept in location with the holding mechanism.

Scenario 3 This experiment showed that mouth, eyes and nose must be visible or similar to the time of registration. This means that if the registration was done with glasses or sunglasses on, these are necessary for authentication. However, it is possible to authenticate with a smiling mouth even when registration took place with a neutral expression.

These limitations identified, we started to cover parts of the picture and retried to authenticate. The result is shown in Figure 4.4. Area (1) was covered first. After a successful authentication, we inserted area (2). We could still successfully authenticate with both areas covered. Covering further larger areas resulted in failed attempts.
Any unauthorized third party with physical access to the smartphone and a photograph of the owner may be able to bypass the lock-screen with the above outlined procedure.

4.5.4 Accelerometer

4.5.4.1 Introduction

The accelerometer is a sensor in smartphones used to detect movement. In Smart Lock this is named on-body detection. The user first has to enter his password in the Screen-Unlock prompt. Afterwards, the accelerometer detects if the device is still laid down or not. Any subsequent attempt will not trigger the Screen-Unlock prompt but directly allow swiping to unlock the screen.

Upon activation, Android notifies the user that it cannot distinguish between the user and an unauthorized third party. Therefore, if a third party takes the device after successful unlock, there is no mechanism to stop them from further accessing the phone via this feature.

4.5.4.2 Misuse Analysis

Consider the situation of a user unlocking his smartphone, checking some information, then disabling the screen by pressing the Power button and storing the device in a bag. Without on-body detection, the smartphone will lock. However, with on-body detection activated, a thief can steal the bag and may not be asked to authenticate.

We tested that situation, unlocking the Nexus 6, pressing the Power button and placing it in a backpack. We then walked approximately 1 km and as expected were not asked to provide the password. In another test, we increased the duration and walked for 30 minutes. Afterwards, we tried to unlock the screen and again were not asked to provide the password. Consequently, we have to conclude that the device does not have a timeout that forces the user to re-authenticate.

Any unauthorized third party with physical access to a smartphone that was recently unlocked and has since been under constant movement will be able to bypass the authentication of the lock-screen.

Figure 4.4: Face Unlock Test Scenario Three

Any unauthorized third party with physical access to the smartphone and a photograph of the owner may be able to bypass the lock-screen with the above outlined procedure.
4.5. Countermeasures

4.5.1 Identified Specific Countermeasures

We could identify several existing countermeasures. For Trusted Places, there is a timeout of four hours that, when reached, will trigger the lock-screen authentication. In the case of Facial Recognition, we observed an attempt threshold of three. After that, Facial Recognition is disabled until the next successful lock-screen authentication.

4.5.2 Countermeasure Improvements

Users should get clear and direct notifications about potential consequences when enabling a trust agent. At the moment only certain warnings are displayed and more information is given in the online support documentation. However, a user might not be able to realize the consequences and Smart Lock undermines the security effort of Android. This comes down to the aforementioned fundamental dilemma of security [17].

Enterprises which use a mobile management system to manage large number of devices that can also enforce security policies have a broader control over the security of a smartphone. Giving users the possibility to do the same by default, built into Android, would allow the same. Android could offer default policy templates and the opportunity to adjust them in an advanced mode. An example policy could be enforcing FDE with a password of certain complexity and disabling inaccurate trust agents such as Accelerometer or Trusted Places.

In regard to different trust agents:

- **Trusted Places** could reduce caching time or disabled it entirely. The latter would eliminate extending Trusted Places, but may lead to false non-recognition of the location and the user would have to authenticate by password. Users might oppose to that, as is weakens the usability factor.

- **Trusted Devices** users should be made aware of the potential threat of cloned NFC tags. Furthermore, Bluetooth devices could be restricted to a version with appropriate security level.

- **Accelerometer** could enforce a timeout set by the user. The timeout could be paired with other trust agents and for example allow a longer duration if the device was unlocked using a NFC tag as trust agent.
4.5.6 Attack Surface

Table 4.20 visualizes the attack surface for the Smart Lock trust anchor categories.

<table>
<thead>
<tr>
<th>Generic Countermeasures</th>
<th>Attack Scenarios</th>
<th>Android</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Locked Bootloader</td>
<td>Disabled adb</td>
</tr>
<tr>
<td>Trusted Places</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Trusted Devices</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Facial Recognition</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Accelerometer</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

† device needs network connectivity

Table 4.20: Attack Surface: Smart Lock Misuse

4.6 Concluding Remarks

In retrospect to our FDE analysis and security assessment, we would like to express our view on the protection this feature offers.

Providing a general statement valid for all devices of Androids running a specific version is impossible due to fragmentation and therefore we can only comment on our Nexus 4 and 6 with Android 4.4.4 and 5.0 respectively.

Both versions use the same cryptographic primitives, with a suitable key length and symmetric block cipher. However, the decision to employ CBC means that in theory some attacks are possible (see Section 4.3). XTS was specifically developed for storage encryption and it is also potentially vulnerable to some of CBC’s attacks or other issues. Regardless, both modes do not offer the data integrity security service, which leaves them vulnerable to manipulation without detection.

The implementation as transparent storage encryption introduces weakness against Cold-Boot Attacks and limits data confidentiality to data at rest, the state where the device must be powered off. This is the classic threat model addressed by FDE, however, smartphones are no classic computing devices. Unlike notebooks or desktop computers, a smartphone is rarely shut down. As a consequence one could argue that FDE is less a protection mechanism than a computational overhead.

To assess this issue, we included the Device-ON attack scenario and demonstrated in the Online Exhaustive Password Search Attack that brute-forcing a four digit PIN is feasible. Furthermore, users of Android 5.0 devices can configure trust agents for Smart Lock to bypass the lock-screen authentication, which undermines the protection offered by FDE. This is especially critical when users opt out of password entering on device start-up. The Start-Up prompt will trigger a wipe after 30 failed attempts and is not bypassable with Smart Lock.

With Android 5.0, Google addressed the Offline Exhaustive Password Search Attack which we can confirm that only our Nexus 4 running Android 4.4.4 was vulnerable to. However, the solution is not permanent and we found a way to circumvent the measures. The only scenario in which the data are protected is when an attacker images the partitions and
immediately must hand the device back to the victim. In cases where the attacker remains in possession of the device, he can involve the smartphone in our Semi-Offline attack. The mitigation of Offline attacks by cryptographically binding the KEK on the device is therefore from our point of view not satisfactory.
Chapter 5

Conclusion

5.1 Our Contribution

This project aimed to assess the security of the FDE feature implemented in Android. In
the course of our analysis the threat model was identified, the source code reviewed and
the functionality tested with a Nexus 4 and a Nexus 6. The former runs Android 4.4.4, the
version currently distributed most widely. The latter runs on Android 5.0 out-of-the-box
with Google’s intended FDE improvements; default encryption on first boot and prevention
of Offline brute-force attacks.

We would like to highlight four main contributions from our research:

- Development of Semi-Offline Exhaustive Password Search Attack, an
  approach to circumvent Android 5.0 enhancements to FDE.

- Improvement of the existing PoC for pre-Android 5.0 Offline Ex-
  haustive Password Search Attacks, for a more precise recognition of
  a decrypted ext4 partition. We included an additional check if the
  ext4 magic signature is present after decryption.

- Identification of the misuse potential of Smart Lock for lock-screen
  authentication bypassing. We identified that Trusted Places might
  have more than double the intended radius. We could also demon-
  strate that Face Unlock can be defeated by digital photographs and a
  holding mechanism. Lastly, the most dangerous trust agent appears
  to be Accelerometer, used for on-body detection, which allowed us to
  unlock after a long period of inactivity.

- Alongside the identification and presentation of attacks and vulnerab-
  ilities, we proposed countermeasure improvements whenever possible.

In Chapter 3 we outlined the understanding of the source code reviewed and explained
the process of first-boot encryption and subsequent decryption activities. These procedures
were discussed in greater detail than ever before and based on our understanding of the
implementation and functionality we could form a description of the underlying threat model,
which is only concerned with the data confidentiality at rest. That means a smartphone is
only protected if it is switched off – a rather rare state for smartphones. However, the user
does not differentiate if the lost or stolen device is switched on or not. As FDE employs
the screen lock password in the KEK generation and Android allows the user to disable
that on screen lock setting, the device will fall back on the default password and not ask
the user to enter their password for decryption during boot. Hence, the only means for
protecting a user’s data is in that case the lock-screen authentication. Furthermore, the
only defence between a transparently encrypted, powered on device and an adversary is also
the lock-screen. We decided therefore to consider the attack scenario Device-ON as well as
lock-screen bypassing specifically.

The main work of this thesis posed the three Exhaustive Password Search Attacks (see
Sections 4.4.1, 4.4.2 and 4.4.3). For all three attack variants we created an attack PoC or
enhanced an existing solution from the security community. The Online approach is clearly
unpractical with an attack duration of over 22 hours. Additionally, our implementation of
this attack is well-protected by the generic countermeasures we described in Section 4.2.
The Offline version of this attack is much more efficient, mostly because there is no timeout
after five failed passwords. Google announced that this vulnerability was addressed with
improvements in the KEK generation process. Indeed, purely Offline is not possible any
more as of Android 5.0. However, we were able to find a new attack approach and developed
a PoC application to demonstrate and compare our method – Semi-Offline – with the known
ones. The attack requires the smartphone to run a server which will perform the keymaster
signing in the KEK generation. We successfully recovered the decrypted master key in
2 hours 8 minutes, more than double the time of Offline, but still hours below the Online
attack.

In addition to those three attacks, we conducted a misuse analysis on the new Smart
Lock feature in Section 4.5. With Smart Lock, an authorised lock-screen authentication by-
pass functionality was introduced. In experiments carried out with our Nexus 6 test device,
we could identify that due to a caching mechanism the radius for Trusted Places more than
doubled for the cache period. We could also confirm the low security level of Facial Recogni-
tion that Android itself warns about. This trust agent is vulnerable to authentication misuse
by presenting a digital photograph of the user. We also identified that it is very sensitive in
the case of illumination but requires distinctive parts of the face (e.g. lips) to be uncovered.
By far the largest impact to undermine FDE security poses the Accelerometer trust agent.
Our experiment demonstrated that a thief could grab-and-run with a smartphone user’s bag
or purse and successfully bypass authentication with Smart Lock after more than 30 minutes.

5.2 Outlook on Android’s Full-Disk Encryption

Earlier in 2015, the Android source code master branch received new files and features.
The file Ext4Crypt.cpp [50] was part of this new code and appears to introduce file system
encryption with ext4. We could not verify in which Android version this should be released
and whether or not FDE as studied in this project will continue to be in use.

The difference between file system and FDE is that in the former the files are encrypted
within the file system which in turn is stored on an encrypted or unencrypted disk. If the disk
itself is unencrypted, then often the file system metadata (e.g. owner, file access permission,
file name, etc.) are unencrypted.
5.3 Future Work

We already covered potential areas for future research in Section 4.4.7. An entropy analysis of the master key would be especially interesting. We explained that the master key is read from /dev/urandom, which does not lock when it runs out of entropy. There is also a potential to increase the research of flash-memory management and its influences with FDE. Additionally, reverse-engineering the TEE’s keymaster application could be the subject for further research. This is the only non-public core element of the FDE. With AOSP containing the code for ext4 file system encryption, it is likely to believe that FDE is either enhanced or replaced by this. A security assessment of the feature itself and the implementation as such, as we did here, could also be of interest.
References


Appendices
Appendix A

Source Code

We developed, or improved, various scripts and applications throughout this project. Apart from the improvements made to Santoku Linux’s script in Section 4.4.2, we are not going to publish source code for Sections 4.4.1 and 4.4.3.

The reason is that we would like to control access to our attack scripts for research purposes only and protect ourselves from any liability under the UK Computer Misuse Act 1990 Section 3A [26].

However, interested parties may request access by contacting Oliver Kunz via campus e-mail: Oliver.Kunz.2014 (at) live.rhul.ac.uk.
Appendix B

Partition Imaging

Imaging a partition from an Android smartphone can be done in many different ways. In the scope of this thesis, there are some limitations:

- The image cannot be created and stored on internal memory, this would be userdata itself
- adb is disabled or screen lock activated that prevents acknowledging the host authentication
- The device may not have an unlocked bootloader
- Unlocking the bootloader will wipe the userdata partition.

Imaging Without Screen Lock

If the device has no screen lock configured, an adversary could activate the adb daemon himself with the outlined steps of Section 4.2.2. Should it already be activated, they can accept the host authentication, which was introduced to prevent any host to communicate with a adb-enabled but locked smartphone.

Step-by-Step Instructions

On The Host

1. Open a command-line terminal and start adb on the host with the command `adb devices`. This command will query for connected devices, which in turn activates the host authentication on the smartphone (accept on smartphone, see below)

2. Start a listener on the smartphone on port `DEVICE_PORT` and configure a reverse forwarding to the attack host on port `HOST_PORT`
   
   `adb reverse 1 tcp:DEVICE_PORT tcp:HOST_PORT`

3. Start a network listener with netcat (nc), listening on port `HOST_PORT` and specify the location for the downloaded image
   
   `nc -l HOST_PORT | dd of=<path-to-destination>`

\(^1\)Requires adb version 1.0.32 or higher
4. Open another command-line terminal and start a remote shell on the smartphone
   adb shell
   
   (a) Verify the configured forwarding
   netstat -an | grep -iw DEVICE_PORT
   
   (b) Identify the path for userdata and metadata partition (e.g. using ls)
   
   (c) Start partition imaging and send data to host
   dd if=<path-to-partition | nc 127.0.0.1 DEVICE_PORT

   Repeat from point 3 for the second partition.

**On The Smartphone**

The Developer Options are enabled, adb is activated and the smartphone connected via USB to the host.

1. The host authentication request is displayed on screen. Press OK to confirm the access from the host and return to host.

**Imaging with Screen Lock**

The solution is similar to the aforementioned, the difference here being that the screen lock prevents the adversary from enabling adb and accepting the host authentication. However, we are going to assume that the smartphone has an unlocked bootloader, allowing the adversary to boot a custom recovery image (see Section 4.2.1).

**Step-by-Step**

**On The Smartphone**

1. Reboot the smartphone into the bootloader. On Nexus 4 and 6.
   
   (a) Power off the device
   
   (b) Press the Power button to start the device
   
   (c) Press the Volume Down button immediately after the Power button

2. When booted into bootloader, change to the host

**On The Host**

We assume that the adversary downloaded or created a custom recovery image for the device. We used the recovery image from Team Win². As outlined in Section 4.2.1, there is no countermeasure such as host authentication etc. to prevent this when the bootloader is unlocked.

1. Open a command-line terminal and boot the smartphone with the custom recovery image
   
   fastboot boot <path-to-recovery.img>

---

²https://twrp.me/
2. Start a listener on the smartphone on port \textit{DEVICE\_PORT} and configure a reverse forwarding to the attack host on port \textit{HOST\_PORT}

\begin{verbatim}
adb reverse \texttt{tcp:DEVICE\_PORT tcp:HOST\_PORT}
\end{verbatim}

3. Start a network listener with \texttt{nc}, listening on port \textit{HOST\_PORT} and specify the location for the downloaded image

\begin{verbatim}
nc -l HOST\_PORT | dd of=<path-to-destination>
\end{verbatim}

4. Open another command-line terminal and start a remote shell on the smartphone

\begin{verbatim}
adb shell
\end{verbatim}

(a) Verify the configured forwarding

\begin{verbatim}
netstat -an | grep -iw DEVICE\_PORT
\end{verbatim}

(b) Identify the path for userdata and metadata partition (e.g. using \texttt{ls})

(c) Start partition imaging and send data to host

\begin{verbatim}
dd if=<path-to-partition | nc 127.0.0.1 DEVICE\_PORT
\end{verbatim}

**Imaging via Hardware-Level**

If both generic countermeasures (see Section 4.2) are active, a more complex solution must be used by finding a way to access the flash-memory and bypass the OS. One solution could be to access the data via test and debug interface, JTAG. A generic process for imaging partitions of embedded devices is given in [4]. The author describes how to identify the JTAG interface on the board and how the \texttt{extest} (external test) and \texttt{debug} mode can be used to image internal memory. In [14], the imaging procedure for a Nexus 4 smartphone is documented. In general, these methods require electronic and soldering skills from an attacker. This is clearly more than the knowledge required for the previous two described imaging solutions. However, if accessing the memory via JTAG fails, there is another potential solution. The flash-memory is unsoldered and accessed using special hardware, like \textit{Ming the Merciless}, discussed in [61].

\footnote{Requires adb version 1.0.32 or higher}
Appendix C

Device Specifications

Attack Host

We used the attack host in Section 4.4.2 and 4.4.3 to perform attacks offline or partially offline. Table C.1 shows the configuration of our attack host.

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manufacturer</td>
<td>Lenovo</td>
</tr>
<tr>
<td>Model</td>
<td>T420s</td>
</tr>
<tr>
<td>OS</td>
<td>Ubuntu 14.04 LTS</td>
</tr>
<tr>
<td>CPU</td>
<td>Intel i7-2620M dual-core 2.70 GHz</td>
</tr>
<tr>
<td>Architecture</td>
<td>64-bit</td>
</tr>
<tr>
<td>System Memory</td>
<td>4.00 GB</td>
</tr>
</tbody>
</table>

Table C.1: Attack Host Specifications
Nexus 6

The information for our Nexus 6 test device comes from the *About Phone* section in the settings menu and www.gsmarena.com.

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manufacturer</td>
<td>Motorola</td>
</tr>
<tr>
<td>Model</td>
<td>Nexus 6</td>
</tr>
<tr>
<td>OS</td>
<td>Android 5.0</td>
</tr>
<tr>
<td>Build Number</td>
<td>LRX21O</td>
</tr>
<tr>
<td>CPU</td>
<td>Quad-core 2.7 GHz Krait 450</td>
</tr>
<tr>
<td>Chipset</td>
<td>Qualcomm Snapdragon 805</td>
</tr>
<tr>
<td>System Memory</td>
<td>3.00 GB</td>
</tr>
<tr>
<td>TEE-enabled</td>
<td>Yes</td>
</tr>
<tr>
<td>Bluetooth</td>
<td>v4.1</td>
</tr>
<tr>
<td>NFC</td>
<td>Yes</td>
</tr>
<tr>
<td>GPS</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Table C.2: Nexus 6 Specifications**

Nexus 4

The information for our Nexus 4 test device comes from the *About Phone* section in the settings menu and www.gsmarena.com.

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manufacturer</td>
<td>LG Electronics</td>
</tr>
<tr>
<td>Model</td>
<td>Nexus 4</td>
</tr>
<tr>
<td>OS</td>
<td>Android 4.4.4</td>
</tr>
<tr>
<td>Build Number</td>
<td>KTU84P</td>
</tr>
<tr>
<td>CPU</td>
<td>Quad-core 1.5 GHz Krait</td>
</tr>
<tr>
<td>Chipset</td>
<td>Qualcomm APQ8064 Snapdragon S4 Pro</td>
</tr>
<tr>
<td>System Memory</td>
<td>2.00 GB</td>
</tr>
<tr>
<td>TEE-enabled</td>
<td>Yes</td>
</tr>
<tr>
<td>Bluetooth</td>
<td>v4</td>
</tr>
<tr>
<td>NFC</td>
<td>Yes</td>
</tr>
<tr>
<td>GPS</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Table C.3: Nexus 4 Specifications**