IT-Security

Android Encryption Attack Analysis

Umgehen der Geräteverschlüsselung von Android Geräten

Diplomarbeit

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

eingereicht von

Christof Kainrath, BSc.

IS151509

im Rahmen des

Studienganges IT-Security an der Fachhochschule St. Pölten

Betreuung

Betreuer/in: FH-Prof. Dipl.-Ing. Dr. Sebastian Schrittwieser, Bakk.

St. Pölten, 13. Oktober 2017 _____ _____

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Fachhochschule St. Pölten GmbH, Matthias Corvinus-Straße 15, A-3100 St. Pölten,T: +43 (2742) 313 228, F: +43 (2742) 313 228-339, E:office@fhstp.ac.at, I:www.fhstp.ac.at

Ehrenwörtliche Erklärung

Ich versichere, dass

- ich diese Diplomarbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich sonst keiner unerlaubten Hilfe bedient habe.
- ich dieses Diplomarbeitsthema bisher weder im Inland noch im Ausland einem Begutachter/einer Begutachterin zur Beurteilung oder in irgendeiner Form als Prüfungsarbeit vorgelegt habe.
- diese Arbeit mit der vom Begutachter/von der Begutachterin beurteilten Arbeit übereinstimmt.

Der Studierende/Absolvent räumt der FH St. Pölten das Recht ein, die Diplomarbeit für Lehre- und Forschungstätigkeiten zu verwenden und damit zu werben (z.B. bei der Projektevernissage, in Publikationen, auf der Homepage), wobei der Absolvent als Urheber zu nennen ist. Jegliche kommerzielle Verwertung/Nutzung bedarf einer weiteren Vereinbarung zwischen dem Studierenden/Absolventen und der FH St. Pölten.

Ort, Datum *Unterschrift*

Kurzfassung

Die aktuellen Zahlen des Smartphonemarktes lassen eine klare Monopolstellung von Android Smartphones erkennen. Das Android Betriebssystem wird von Google entwickelt und von vielen Smartphoneherstellern für ihre Geräte adaptiert. Die Enthüllungen von Edward Snowden und der durch den Terroranschlag in San Bernadino verursachte Rechtsstreit zwischen Apple und dem US-amerikanischen Department of Justice, lenkt den Fokus der Öffentlichkeit auf die Sicherheit ihrer Smartphones. Durch die Aufmerksamkeit der Öffentlichkeit und der Tatsache, dass die größten Technologiekonzerne in den Vereinigten Staaten von Amerika beheimatet sind, müssen sich Technologieriesen wie Apple, Google, Cisco oder Microsoft die Frage nach einer heimlichen Zusammenarbeit mit den amerikanischen Staatsbehörden wie NSA oder FBI gefallen lassen. Der Wunsch der Öffentlichkeit nach mehr Sicherheit, wird zunehmend größer, weshalb Google, im Bezug auf Geräteverschlüsselung, das Android Betriebssystem mehrmals verbessert hat. Darunter jüngst mit der Veröffentlichung von Android 7.0 und der damit verbundenen Einführung einer dateibasierten Geräteverschlüsselung.

Diese Arbeit betrachtet mögliche Bedrohungen und Angriffe für Smartphones mit den Android Versionen 6.0 bis 7.1 und beschreibt deren Arten der Geräteverschlüsselung, welche aktuell in den genannten Versionen implementiert sind. Dabei werden unter anderem die essentiellen Technologien erklärt, welche für die jeweilige Geräteverschlüsselung benötigt werden. Auch werden die bisher bekannten Angriffe erklärt und am Ende der Arbeit eine Sicherheitsüberprüfung durchgeführt, in welcher ein Angriff als Proof of Concept für Android Version >7.0 implementiert wird.

Als Ergebnis wurde ein Überblick der bereits bekannten Angriffe und den dazugehörenden betroffenen Android Versionen erstellt und erfolgreich ein Proof of Concept für das Umgehen des Fingerabdrucksensors für Android >7.0 implementiert und getestet. Die gewonnenen Erkenntnisse zeigen, dass es stark vom Smartphonehersteller abhängt, welche Art von Geräteverschlüsselung letztendlich vom Gerät unterstützt und verwendet wird und wenn die dementsprechenden Voraussetzungen erfüllt werden, die Geräteverschlüsselung von Android 6.0 bis 7.1 umgangen werden kann.

Abstract

The current figures of the smartphone market show a clear monopoly of Android smartphones. The Android operating system is developed by Google and adapted by many smartphone manufacturers for their devices. The revelations of Edward Snowden and the lawsuit between Apple and the US Department of Justice, caused by the terrorist attack in San Bernadino, is driving the public's attention to the security of their smartphones. With the attention of the public and the fact that the largest technology companies are located in the United States of America, technology giants like Apple, Google, Cisco or Microsoft have to face the questions for a secret collaboration with the American state authorities such as NSA or FBI raised by the public. The public's desire for more security is increasing, which is why Google has improved the Android operating system several times in terms of device encryption. Including the recent release of Android 7.0 and the introduction of a file-based device encryption within this version.

This work examines possible threats and attacks for smartphones with the Android OS versions 6.0 to 7.1 and describes their types of device encryption, which are currently implemented in the mentioned versions. Among other things, the essential technologies that are required for the respective device encryption are explained. Also the previously known attacks are explained and at the end of the work a security check will be carried out, in which an attack as Proof of Concept for Android version >7.0 is implemented.

As a result, an overview of the known attacks and the affected Android versions was created and a Proof of Concept for bypassing the fingerprint sensor for Android >7.0 was successfully implemented and tested. The findings show that it depends heavily on the smartphone manufacturer what type of device encryption is ultimately supported and used by the device, and if the corresponding requirements are met, device encryption can be bypassed from Android 6.0 to 7.1.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Relevanz des Themas 1

1.2 Forschungsleitende Fragestellung 3

1.3 Aufbau der Arbeit 4

2 Stand der Technik **5**

2.1 Android Betriebssystem 5

2.1.1 Offenheit des Android Betriebssystems 6

2.1.2 Fragmentierung 7

2.2 Android Compatibility Definition Document (CDD) 9

2.2.1 CDD Versions Vergleich 10

2.3 Password Stretching Funktionen 16

2.3.1 Password Based Key Derivation Function 2 16

2.3.2 scrypt 17

2.4 Android Full Disk Encryption 18

2.4.1 Device-Mapper und dm-crypt Funktion 19

2.4.2 Schlüsselmanagement FDE 20

2.4.3 Verschlüsselung bei initialen Bootvorgang 21

2.5 Android File Based Encryption 24

2.5.1 Mehrere Verschlüsselungsareale 24

2.5.2 Schlüsselmanagement FBE 25

3 Verwandte Arbeiten **28**

4 Bedrohungsanalyse **30**

4.1 Bedrohungsmodell 30

5 Bedrohungslage **33**

5.1 Angriffsszenarien . . . 33

5.2 Implementierte Gegenmaßnahmen . . . 34

5.2.1 Gesperrter Bootloader . . . 34

5.2.2 Deaktivierte Android Debug Bridge . . . 35

5.2.3 TEE . . . 37

5.3 Bekannte Angriffe auf die Implementierung der Verschlüsselung . . . 38

5.3.1 Online Brute Force Angriff . . . 38

5.3.2 Offline Brute Force Angriff . . . 40

5.3.3 Semi-Offline Brute Force Angriff . . . 43

5.3.4 TEE Schwachstelle . . . 46

5.3.5 Cold-Boot Angriffe . . . 47

5.3.6 Statisches Default Passwort und Master Key . . . 49

5.3.7 Evil-Maid Angriff . . . 52

6 Sicherheitsüberprüfung **54**

6.1 Methodik . . . 54

6.2 Setup der Android Smartphones . . . 55

6.2.1 Entsperren des Bootloaders . . . 55

6.2.2 Rooten des Gerätes . . . 55

6.2.3 Android Quellcode Repository . . . 56

6.3 Partition Imaging . . . 56

6.3.1 Imaging ohne Sperrbildschirm . . . 57

6.3.2 Imaging mit Sperrbildschirm . . . 58

6.3.3 Imaging via Hardware-Level . . . 59

6.4 Umgehen des Fingerabdrucksensors . . . 60

6.4.1 Software Komponenten . . . 60

6.4.2 Angriffsanalyse . . . 61

6.4.3 Gegenmaßnahmen . . . 63

6.4.4 Angriffsfläche . . . 64

6.5 Übersichtstabelle der Angriffe . . . 65

7 Fazit und Ausblick **67**

Abbildungsverzeichnis **70**

Tabellenverzeichnis **71**

Literatur **76**

1 Einleitung

Dieses einführende Kapitel hat in erster Linie den Zweck, die Relevanz und Aktualität des in der Arbeit zu behandelnden Themas zu verdeutlichen und geht auf die Motivation zur Erstellung der Arbeit ein. Ferner werden die forschungsleitenden Fragestellungen angeführt, welche durch die Arbeit geklärt werden, und ein Überblick über den Aufbau des Dokumentes gegeben.

1.1 Relevanz des Themas

Seit dem Release der ersten Version von Android im Jahr 2008 bemühen sich seine Entwickler jede neue Version ihres Betriebssystems für mobile Geräte mit Verbesserungen und neuen Features zu erweitern. Im Jahr 2011 und in der Version 4.4 von Android wurde es um die Funktion der Full Disk Encryption (FDE), also Festplattenverschlüsselung, erweitert. Das neue Feature musste manuell vom Benutzer aktiviert werden, daher wurde es zum Zeitpunkt der Einführung der Full Disk Encryption von den meisten Benutzern ignoriert, da auch der Großteil der Benutzer nicht wusste, dass ihre Geräte diese neue Funktion besaßen. Daher verwendeten die Benutzer ihre Smartphones weiterhin unverschlüsselt.

Im Jahr 2013 wurde die öffentliche Meinung und das Bild der IT-Sicherheit grundlegend durch die Veröffentlichungen von Edward Snowden geändert. Diese Enthüllungen beschreiben die technischen Fähigkeiten von nationalen Geheimdiensten wie der NSA oder dem GCHQ[1] und warfen damit neue Fragen im Bezug der Überwachungsmöglichkeiten dieser Staaten auf. Darunter die Frage inwiefern Technologieriesen und Branchenführer in diversen IT Sektoren wie Apple, Google, Cisco oder Microsoft mit diesen Geheimdiensten zusammenarbeiten um deren Benutzer auszuspionieren oder auch müssen, wenn sie nicht das Schicksal von beispielsweise dem Maildienstanbieter Lavabit[2] teilen wollen. Gegen diese Unternehmen wurde ein Durchsuchungsbeschluss sowie die Herausgabe aller SSL-Schlüssel erwirkt. Das Unternehmen gab zunächst sämtliche Schlüssel als Ausdruck in Miniaturschrift (Schriftgröße 4pt) heraus, das Gericht ordnete dann aber eine Auslieferung in brauchbarer digitaler Form an. Bei Zuwiderhandlung wurde eine Strafe von 5.000 Dollar pro Tag angedroht. Lavabit stellte daraufhin den Betrieb ein.

Als eine der Konsequenzen veröffentlichte Google ein Statement indem das Unternehmen ankündigte mit der Veröffentlichung der Version 5.0 von Android die persönlichen Daten der Benutzer in jedem Smartphone mit dieser Version zu verschlüsseln.[3] Dieser Schritt von Google und auch die Verschlüsselung von Apples IPhones wurde von den Geheimdiensten und anderen Gesetzesorganen nicht sonderlich wohlwollend aufgenommen. In einer öffentlichen Rede bekundete FBI Leiter James Cormey seine Bedenken über dieses Vorgehen und betonte dass das Verhindern des Zugriffs auf die Informationen eines Smartphones eine Gefahr darstellt, da nicht nur die Daten von anständigen Bürgern durch diese Maßnahmen geschützt werden, sondern auch die von Kriminellen und Terroristen.[4]

Ein paar Monate nach der Ankündigung seitens Google die Festplattenverschlüsselung als Standard out-of-the-box[1] Konfiguration zu verwenden, meldeten diverse Stellen, dass Google seine Ankündigung reversiert hat.[5]

Zum damaligen Zeitpunkt war es nicht ganz klar ob das der Wahrheit entsprach oder nicht, da unterschiedliche Quellen nicht das selbe berichteten. Einerseits wurde in der Liste der Sicherheitsverbesserungen von Android 5.0, das im Januar 2015 veröffentlicht wurde, klar die out-of-box FDE Verschlüsselung angeführt.[6]

Andererseits kann man im Compatibility Definition Document zu Android 5.0 nachlesen, dass das Feature nicht als Muss gelistet wird, sondern als “Strongly Recommended“da es in künftigen Versionen als Muss gehandelt werden soll.[7][S.59] Erst mit Erscheinen von Android 5.1 im Juli des selben Jahres wurde die Festplattenverschlüsselung für Android Geräte als Muss deklariert.[8][S.55]

Erneute Medienpräsenz erlangte das Thema Festplattenverschlüsselung von Smartphones traurigerweise durch die Terroranschläge von San Bernadino und den daraus resultierenden Rechtsstreit zwischen Apple und dem FBI. Bei diesem Rechtsstreit versuchte das FBI Apple gerichtlich dazu verpflichten, ihnen zu helfen das sichergestellte Smartphone zu entschlüsseln. Im Konkreten handelte es sich um ein Iphone 5c mit iOS 9, das mit einem 4 - Ziffern Passcode gesichert war. Nach dem 10. Fehlversuch würde sich ein Teil des Schlüssels zum entschlüsseln löschen und die Entschlüsselung des Smartphones wäre unmöglich. Das FBI forderte daher ein temporäres Software Update von Apple das diverse Sicherheitsmaßnahmen des Gerätes umging. Apple weigerte sich vehement gegen diese Forderung und Apple CEO Tim Cook verkündete in einen offenen Brief an seine Kunden, Apple werde keine Backdoor[2] für das

[1] Bezeichnet eine Konfiguration eines technischen Gerätes, das frisch aus der Verpackung genommen wurde ohne eine manuelle Änderung des Endkunden

[2] Eine Backdoor (engl. Hintertür) bezeichnet einen (oft vom Entwickler eingebauten) Teil einer Software, der es dem User ermöglicht, unter Umgehung der normalen Zugriffssicherung Zugang zum Computer oder einer sonst geschützten Funktion eines Programms zu bekommen.

FBI entwickeln, da eine solche Backdoor im Allgemeinen viel zu gefährlich sei und in falschen wie auch richtigen Händen ein zu großes Missbrauchspotential besäße.[9]

Knapp einen Monat später vermeldete das US-amerikanische Department of Justice dass das besagte Smartphone auch ohne die Hilfe von Apple geknackt wurde. Im Zuge dessen wurde die Klage gegen Apple zurückgezogen. Dabei wurde weder geklärt wie das Iphone geknackt wurde, noch ob die Daten, welche sich darauf befanden, hilfreich waren um zu klären ob die beiden Attentäter mit terroristischen Gruppierungen im Kontakt standen. Es wird gemutmaßt, dass die Hilfe des israelischen Sicherheitsunternehmens Cellebrite vom Department of Justice in Anspruch genommen wurde, um das Smartphone zu knacken.[10]

Daher ist das Thema Verschlüsselung im Bezug auf Smartphones, insbesondere Android Geräte, in den Augen des Autors ein sehr wichtiges und ernst zu nehmendes Thema, da das Smartphone in seinem alltäglichen Gebrauch viele heikle Daten seines jeweiligen Besitzers verwaltet. Der Fokus auf Android Geräte in dieser Arbeit erklärt sich daher, dass sie einen fast monopolartigen Marktanteil[11] besitzen und es sich bei Android um eine freie Software handelt, die quelloffen[12] entwickelt und von vielen unterschiedlichen Herstellern für ihre Smartphones adaptiert wird. Einerseits besitzen die meisten neu erworbenen Android Geräte eine Verschlüsselung um die Daten des Benutzers zu schützen und die Anzahl der Geräte ohne diese Funktionalität wird in den nächsten Jahren aller Voraussicht stark sinken. Andererseits wird eine wirklich flächendeckende Etablierung von verschlüsselten Smartphones und die breite Nutzung ihrer zahlreichen Vorteile auch nur dann möglich sein, wenn sich die Menschen in der Verwendung ihrer Smartphones und deren Verschlüsselungstechnologien sicher fühlen und diese bedienen können.

1.2 Forschungsleitende Fragestellung

Diese Arbeit beschäftigt sich mit den Verschlüsselungstechniken von aktuellen Android Geräten. Es werden die verschiedenen Schlüsseltechnologien der Android Geräteverschlüsselung identifiziert und deren Funktionsweise sowie Prinzipien erklärt. Dabei werden die vorhandenen Schutzmechanismen dieser Verschlüsselungen analysiert, mögliche Bedrohungen definiert und getestet ob, und in mit welchem Aufwand diese Mechanismen umgangen werden können oder nicht.

Basierend auf diesen Analysen kann bestimmt werden, wie gut die Daten der Benutzer auf ihrem Android Smartphone geschützt sind oder nicht. Des Weiteren kann bestimmt werden welche Faktoren vorherrschen müssen, um die Verschlüsselung zu umgehen und welche weiteren Maßnahmen getroffen werden sollten, um dies zu verhindern.

Die forschungsleitenden Fragestellungen sollen, durch die im Zuge der Arbeit gewonnenen Erkenntnisse, beantwortet werden:

- *Inwiefern lässt sich die Geräteverschlüsselung in Android 6.0 bis 7.1 umgehen?*
  - *Welche Schlüsseltechnologien werden zur Umsetzung der Geräteverschlüsselung in Android 6.0 bis 7.1 verwendet?*
  - *Welche bereits bekannten Angriffe existieren für die Geräteverschlüsselung in Android 6.0 bis 7.1?*

1.3 Aufbau der Arbeit

Das nächste Kapitel der Arbeit wirft einen grundlegenden Blick auf den aktuellen Stand der Technik im Bereich des Android Betriebssystems und den Verschlüsselungstechniken welches dieses besitzt, um dadurch einen konkreten Wissenstand für die Analyse zu erlangen und um gewisse Abgrenzungen für diese Analyse zu treffen. Dazu wird zuerst geklärt was genau Android ist, welche Eigenschaften und Architektur es besitzt. Des Weiteren wird behandelt was die Offenheit des Betriebssystems bedeutet und welche Besonderheiten sich durch die Fragmentierung von Android ergeben und wie Google durch die Compatibility Definition Documents diesem Umstand entgegenwirkt. Das restliche Kapitel wird die verschiedenen Arten der Geräteverschlüsselung bei Android Geräten behandeln und die Theorie dahinter aufzeigen.

Dem Stand der Technik folgt ein Kapitel über verwandte und vorangegangene Arbeiten zum Forschungsthema.

Der nächste Teil der Arbeit widmet sich den Bedrohungen und Angriffen auf die Geräteverschlüsselung von Android. Hierzu werden die Angriffsszenarien auf Smartphones bestimmt und des weiteren welche generischen Gegenmaßnahmen in Android Geräten implementiert sind um Angreifer daran zu hindern unbefugten Zugriff auf die Daten des Gerätes zu erlangen. Das restliche Kapitel beschäftigt sich mit den bisher bekannten Angriffe gegen die Geräteverschlüsselungen von Android. Hierzu werden die Attacken vorgestellt, deren Funktionsweise, welche Gegenmaßnahmen zu der jeweiligen Attacke existieren und welche Angriffsfläche die jeweilige Attacke besitzt.

Zu guter Letzt werden die gewonnenen Erkenntnisse verwendet, um eine Sicherheitsüberprüfung einer aktuellen Android Version (7.1.1 und 7.1.2) durchzuführen und einen Proof-of-Concept[3] für eine Attacke zu erstellen um damit die Geräteverschlüsselung eines Nexus 5X Smartphones zu umgehen.

[3] Proof-of-Concept bezeichnet in der IT-Sicherheit die prinzipielle Durchführbarkeit eines Angriffes, ohne eine wirkliche Schadfunktion auszulösen, um zu beweisen, dass ein Sicherheitsproblem besteht.

2 Stand der Technik

Dieses Kapitel betrachtet den aktuellen Stand der Technik im Bereich Android und dessen Geräteverschlüsselung. Zuallererst wird auf die Eigenheiten von Android eingegangen die es von anderen mobilen Betriebssystemen unterscheidet. Im Anschluss werden die Hilfsfunktionen erklärt, die zusätzlich nötig sind um aus dem Geheimnis des Benutzers (PIN, Passwort oder Muster) einen Schlüssel für die Verschlüsselung zu generieren. Mit diesem Wissen widmen wir uns anschließend der Geräteverschlüsselung selbst und welche Komponenten, hardware- wie auch softwaretechnisch, zusammenspielen um die Verschlüsselung auf dem Mobilgerät durchzuführen.

2.1 Android Betriebssystem

AOSP (Android Open Source Project) nennt sich die Community und Plattform die aus Google und OHA Mitgliedern (Open Handset Alliance)[1] besteht und hauptverantwortlich für die Entwicklung und Verbesserung des Android Betriebssystems sind. Seit dem Release von Android 4.0 besitzt das Betriebssystem die Funktion eine Geräteverschlüsselung in Form einer vollen Festplattenverschlüsselung (FDE) durchzuführen. Dabei wird die /data Partition, auch genannt `userdata`, des Gerätes verschlüsselt. Diese volle Festplattenverschlüsselung tritt in Kraft wenn das Gerät ausgeschaltet wird. Siehe Abschnitt 2.4 2.4 für genauere Informationen. Ist das Gerät in Betrieb und hat man den Sperrbildschirm des Gerätes überwunden, funktionieren durch die Device Mapper Bibliothek dm-crypto 2.4.1 die Lese- und Schreiboperation auf dem Gerät transparent.

Die Verwendung von FDE hat gewisse Nachteile (Siehe Abschnitt 2.5), daher implementierte man zusätzlich ab der Version 7.0 von Android ein anderes Konzept der Geräteverschlüsselung, nämlich die dateibasierte Festplattenverschlüsselung, auch File Based Encryption (FBE) genannt. Daher kann man bei Geräten mit dieser Version von einer anderen Geräteverschlüsselung ausgehen, wenn diese auch wirklich verwendet wird. Aufgrund der Abwärtskompatibilität kann nicht immer ausschließlich auf Grund der Versionsnummer von Android ausgegangen werden, welche Geräteverschlüsselung jetzt wirklich

[1] Die Open Handset Alliance ist ein Konsortium zur Schaffung eines offenen Standards für Mobilgeräte. Mitgliedern des Konsortiums ist es nicht gestattet, Geräte herzustellen, die eine zu Android inkompatible Version der Software nutzen.

verwendet wird. Generell besitzen wenig erhältliche Smartphones die Eigenschaften, um die FBE zu betreiben. Siehe Abschnitt 2.2.1 für weitere Informationen zu den Anforderungen.

Auch hier zeigt sich der Bezug zu Linux sehr stark, da beide Arten von Linux Bibliotheken übernommen wurden. FDE basiert unter anderem auf dem Schema von dm-crypt[13] und FBE verwendet die native ext4 Dateisystem Verschlüsselung die in Linux ab der Kernelversion 4.1 verfügbar ist.[14]

2.1.1 Offenheit des Android Betriebssystems

Wie schon erwähnt verwaltet das AOSP den Quellcode von Android und Google ist der Haupteditor innerhalb des Projektes. Der Grundgedanke von Android ist ein offenes, frei verfügbares Betriebssystem für mobile Geräte, daher baut dieses Projekt auf verschiedenen Open Source Komponenten auf. Das inkludiert unzählige Bibliotheken, den Kernel von Linux, ein komplettes User Interface und viele weitere Komponenten, dennoch haben alle Teilkomponenten eines gemeinsam, nämlich eine von Open Source Initiative (OSI) geprüfte Lizenz. Der meiste Android Quellcode ist unter der Apache 2.0 Software Lizenz oder GNU Public License (GPL) lizenziert, dennoch sind manche Teile des Quellcodes nicht Open Source, wie die Digital Rights Management Komponenten, der Bootloader, diverse Firmware zu Peripheriegeräten, vorinstallierte Google Apps oder die Radioempfang Komponente. Zusätzlich merken viele Android Enthusiasten, die viel mit dem Quellcode arbeiten an, dass das Projekt nicht so frei und offen entwickelt wird, wie versprochen. Es existieren viele Anzeichen die andeuten, dass Google große Teile von Android im Geheimen entwickelt und so die Grundidee davon ad absurdum führt. Selbst die Smartphone Flagschiffreihen Nexus und Pixel enthalten Quellcode von proprietären Programmen. Diese Teile Closed-Source[2] zu belassen, hindert die Android Gemeinschaft Portierungen des Betriebssystems vorzunehmen und macht diese Bemühungen schwieriger.[15, S.7]

Das Android Betriebssystem setzt wie erwähnt auf einen modifizierten Linux Kernel auf, der hauptsächlich unter die GPL Lizenz fällt. Dieser Umstand fällt jedem Power-User und Software Entwickler schnell auf, da eines der praktischsten Tools zum Erkunden und Arbeiten mit Android die sogenannte Android Debug Bridge (adb) ist, die ein Teil des Android Software Development Kits ist. Wird dieses Tool gestartet und ein Gerät mit Android Betriebssystem ist mit dem Host verbunden, kann eine Remote Shell auf das Gerät, die einer Linuxshell ähnlich ist, verwendet werden. Diese Shell enthält nicht alle Befehle und Funktionen einer Bourne-again Shell, aber dennoch die wichtigsten und Android verwendet das gleiche Dateiberechtigungsschema wie Linux.

[2] Im Gegenteil zu Open Source Software, ist der Quellcode nicht frei verfügbar und die Software wird proprietär vertrieben

2.1.2 Fragmentierung

Durch die Offenheit und den Open Source Ansatz ergibt sich dadurch der Effekt von Fragmentierung innerhalb der Android Gemeinde. Dieser Effekt betrifft auf der einen Seite die Softwareplattform und deren Entwickler und auf der anderen Seite die Hardwarehersteller und Designer von Smartphones. Android auf dem einen Gerät ist nicht gleich dem Android auf einem anderen Gerät. Wenn man verschiedene Geräte unterschiedlicher Smartphone Hersteller nebeneinander legt und vergleicht, kann mit Leichtigkeit die Verschiedenheit festgestellt werden, da keine Gerätereihe einer anderen gleicht. Hersteller führen neue Hardwarelayouts, Sensoren, Kameras und weitere Features ein, wie es ihnen beliebt. Auch Funknetzbetreiber können mögliche Anpassungen fordern. Dadurch muss auch das Betriebssystem auf die einzelnen Geräte und deren Anforderungen der Hardware angepasst werden.

Dadurch ergibt sich, dass diese Anpassungen natürlich nicht alle von Google erledigt werden, sondern von den Herstellern selber, was die erwähnte Fragmentierung des Betriebssystems verursacht. Zusätzlich kommen noch die unterschiedlichen Smartphonereihen der einzelnen Hersteller als Effekt hinzu. Nicht jedes Smartphone besitzt alle technischen Anforderungen die neueste Stock Android Version und dessen neue Funktionen zu betreiben. Daher gibt es auch Probleme von Portierungen einer neuen Version auf ältere Smartphones, was dazu führt das viele Geräte die im Umlauf sind, das Ende ihres Updatezyklusses erreicht haben. Will man als Endkunde am letzten technischen Stand bleiben ist er mehr oder weniger gezwungen neuere Geräte mit möglichst langem Updatezyklus zu kaufen um dies zu garantieren.

Die Google Nexus und dessen Nachfolger Pixel Produktlinie ist in dieser Hinsicht entschieden anders, da auf diesen Plattformen Android entwickelt und getestet wird. Daher erhalten Besitzer dieser Smartphones einerseits monatlich ein Sicherheitsupdate für Android und andererseits ist es eher wahrscheinlich ein Upgrade auf eine neue Android Version zu bekommen. Als Beispiel sei das Google Nexus 5X erwähnt, das ursprünglich mit der Version 6.0 im Oktober 2015 veröffentlicht wurde, jetzt im Juli 2017 auf Android Version 7.1.2 läuft und im August 2017 das Update für Android 8 erhalten wird.[3] Im Vergleich ein Samsung Galaxy S6 wurde im April 2015 mit Android 5.1.1 veröffentlicht und steht im Juli 2017 bei Version 7.0 und ist auch nicht für ein Update auf Version 8.0 von Android vorgesehen.

Dieses kleine Beispiel soll das erwähnte Problem mit der Fragmentierung verdeutlichen da Benutzer die kein Nexus oder Pixel Gerät besitzen, oft länger auf ein Versionsupgrade und Sicherheitsupdates für ihr Gerät warten müssen und im schlimmsten Fall gar keines bekommen. Daher existieren am Markt sehr viele Geräte die eine alte und/oder ungepatchte Version von Android betreiben und somit Monate oder Jahre in Betrieb bleiben. Wie in der nachfolgenden Tabelle 2.1 zu erkennen ist, besteht der Großteil

[3] https://www.valuewalk.com/2017/06/phones-compatible-get-android-o-update/

der sich in Umlauf befindlichen Android Smartphones aus Geräten auf denen Android 5.0-5.1 (30.1 %) und Android 6.0 (31.8 %) läuft. Dies sind die offiziellen Zahlen vom Google Dashboard die monatlich in einer Periode von 7 Tagen erhoben und veröffentlicht werden.Der Anteil an Android 7.0 Geräten ist relativ gering und steigt auch nur mäßig, da der Unterschied zwischen der Erhebung im Juni 2017 und Juli 2017 bei allen Android 7 Versionen genau 2 % betrug (Steigerung von 9.5 % auf 11.5 %). [16, 17]

Codename	**Version**	**Verteilung**
Gingerbread	2.3.3 - 2.3.7	0,7 %
Ice Cream Sandwich	4.0.3 - 4.0.4	0,7 %
Jelly Bean	4.1.x	2.8 %
	4.2.x	4.1 %
	4.3	1.2 %
Kit Kat	4.4	17.1 %
Lollipop	5.0	7.8 %
	5.1	22.3 %
Marshmallow	6.0	31.8 %
Nougat	7.0	10.6 %
	7.1	0.9 %

Tabelle 2.1: Tabelle mit Android Versionen und deren Verteilung Stand 06. Juli 2017. [17]

Zusätzlich zum gezeigten Softwareaspekt der Fragmentierung, betrifft diese auch die Hardware. Verschiedene Hersteller bedienen, wie angedeutet, unterschiedliche Kundenschichten, je nachdem wie teuer das Smartphone am Ende sein soll. Daher reicht die Palette an Geräten von billigen Low-End Geräten bis zu teuren Flagschiffgeräten. Die unterschiedliche Hardware beeinflusst nicht nur den Preis oder das Aussehen des Gerätes sondern, wie erwähnt, die technischen Fähigkeiten und somit auch die Möglichkeiten des Betriebssystems. Beim Verbau der Hardwarekomponenten gibt es unterschiedliche Chiphersteller, die wiederum manche Standards für einen Typ Chip unterschiedlich implementieren. Beispielsweise wird das in einem späteren Kapitel behandelt, wo dies zu einer Sicherheitslücke geführt hat, die nur seitens der Software gepatcht werden konnte, jedoch nicht hardwaretechnisch 5.3.4. Diese Fragmentierung zu kontrollieren ist nicht nur im Interesse von Google sondern auch aller die mit Android Smartphones zu tun haben, angefangen vom Entwickler bis zum Endbenutzer. Daher werden sogenannte Android Compatibility Definition Documents (CDD) verwendet um die Fragmentierung einzugrenzen und allen Beteiligten ein klareres Bild über die Anforderungen und Funktionen der einzelnen Android Versionen

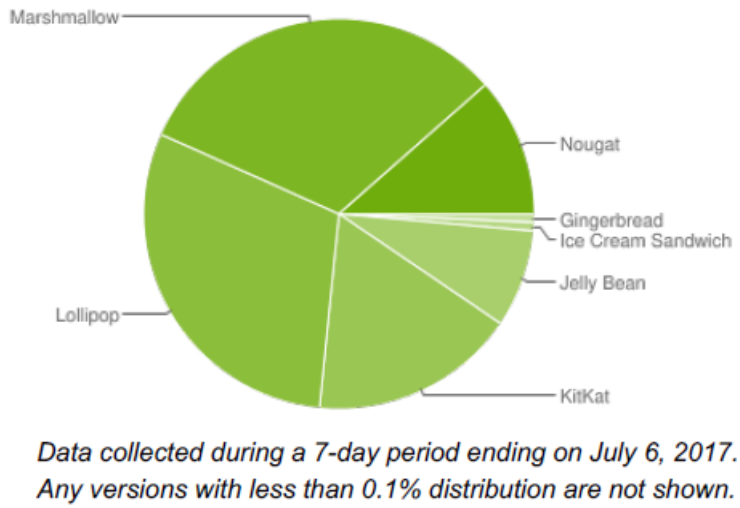

Abbildung 2.1: Tortendiagramm zur Tabelle 2.1.

zu geben.

2.2 Android Compatibility Definition Document (CDD)

Drittanbieter von Software und Hardware sind eine der Kernelemente für den Erfolg einer Smartphone Plattform wie Android, daher sind auch viele Parteien in das Projekt involviert und sicher auch einer der Gründe warum Android gerade einen Marktanteil von 86 % hat. [11] Ein Smartphone mit weniger Applikationen ist klarerweise nicht so attraktiv für Entwickler und Endbenutzer. Eine geeinte Plattform ist besser als ein fragmentierte, daher existiert auch dieser Kompromiss zwischen einerseits Offenheit und Diversifizierbarkeit und andererseits das Auftreten als einer Plattform und den daraus resultierenden Vorgaben die zu erfüllen sind.

Das Android Compatibility Definition Document (CDD) listet alle Anforderungen auf die erfüllt werden müssen (MUST Requirement) damit ein Device kompatibel zur entwickelten Android Version ist. Für jede Hauptversion von Android existiert ein solches Dokument und dessen Anforderungen, müssen erfüllt sein. Des Weiteren müssen auch alle Anforderungen, der darin referenzierten Dokumente erfüllt werden. Es repräsentiert damit den Policy Aspekt der Android Kompatibilität.

Es ist wichtig dass sich Hersteller und Drittentwickler an diese Richtlinien halten, da keine Testsuite wirklich in der Lage ist alles zu testen. Beispielsweise kann das Verhalten der OpenGL Grafik API getestet werden, ob sie sich richtig verhält, aber kein Softwaretest kann verifizieren ob die Grafik auch richtig am Bildschirm angezeigt wird. Allgemein können viele Hardwarefeatures unmöglich getestet,

werden wie Tastaturen, Wi-Fi Signale oder Bluetooth.

Das klare Ziel dieser Dokumente ist einen Überblick über die Anforderungen zu geben, referenziert als Art Überblicksdokument verschiedener Inhalte, um einen Aufschluss zu geben wie der Quellcode zu verwenden ist. Dabei besitzen sie jedoch keinen Anspruch allumfassend zu sein. Der Quellcode der jeweiligen Version bleibt die genauere Spezifizierung der Plattform und dessen APIs.

2.2.1 CDD Versions Vergleich

Wie bereits erwähnt sind die generellen Vorgaben die für die einzelnen Android Versionen zu erfüllen sind im CDD Sektion 9.9 der jeweiligen Version angeführt. [7][S.59]
Die Anforderungen werden im Folgenden zusammengefasst und die einzelnen Versionen unterschieden. Die Interpretation der einzelnen Kategorien (MUST, MUST NOT und SHOULD) erfolgt nach deren Deklaration im RFC 2119. [18]

MUST Anforderungen Android 5.0

1. Unterstütze FDE, falls ein Sperrbildschirm implementiert ist
2. Verschlüssele die /data Partition und die Partition der nicht entfernbaren internen SD Karte des Gerätes
3. Verwende AES mit der minimalen Schlüssellänge von 128-Bit
4. Verwende einen symmetrischen Block Cipher Mode, der für Speicherung konzipiert wurde (AES-CBC ESSIV oder AES-XTS)
5. Binde kryptografisch die Passwort Stretching Algorithmen an einen hardwareunterstützten Schlüsseltresor (TEE) falls ein solcher vorhanden ist

MUST NOT Anforderungen 5.0

1. Den Schlüssel zur Verschlüsselung unverschlüsselt auf den Datenträger schreiben
2. Den Schlüssel zur Verschlüsselung unter keinen Umständen vom Gerät versenden

SHOULD Anforderungen 5.0

1. Aktivierte Geräteverschlüsselung FDE zu jeder Zeit
2. AES verschlüsselt den DEK, falls dieser aktiv nicht gebraucht wird

- Verwende dabei ein gestretchtes Passwort vom Benutzer welches vom Sperrbildschirm kommt, mit einem langsamen Stretching Algorithmus. (scrypt oder PBKDF2)
- Verwende ein Standartpasswort (default_password) falls kein Passwort vom Benutzer angelegt worden ist

Die Spezifikation im CDD für Android 5 lässt Raum für unterschiedliche Interpretationen offen. FDE kann wie hier beschrieben, nur für Geräte angewendet werden, die einen Sperrbildschirm implementieren, was wiederum Sinn macht da ohne diesen die Daten des Smartphones für jeden mit physischen Zugriff auf das Gerät zugänglich wären. Benutzer erwarten einen Sperrbildschirm, daher ist es unwahrscheinlich das neue Geräte ohne diesen veröffentlicht werden, daher kann man eine Verschlüsselung mit FDE erwarten.

Die zweite MUST Anforderung ist die wichtigste und entscheidendste im Bezug auf die Geräteverschlüsselung, da diese Anforderung auch der Hauptunterschied zu Apple Produkten darstellt. Die bereits erwähnte /data Partition beinhaltet die installierten Anwendungen und privaten Dateien des Benutzers. Die interne simulierte SD Karte speichert wiederum meist Dateien des Benutzers da diese verwendet wird wenn das Gerät als USB Stick mit einem Host PC verbunden wird. Zusätzlich hat man mit adb Zugriff auf die SD Karte um Dateien zu verschieben. Daher werden nur private Daten oder Anwendungen von anderen Softwareanbietern und deren Daten verschlüsselt und nicht das Betriebssystem Android selber, was denn Unterschied zu iOS darstellt, da dort auch die proprietäre Betriebssystemkomponenten und die interne Architektur geschützt werden.

Die fünfte MUST Anforderung erlaubt es Android 5.0 leider das Gerät mit FDE zu verschlüsseln, ohne die wichtigste Verbesserung dieser Version, nämlich ohne einen hardwaregebundenen Schlüsseltresors. Ist dieser nicht vorhanden funktioniert die Verschlüsselung wie davor und manche simple Attacken wie ein normaler Offline Brute Force Angriff ist wieder möglich. Siehe Abschnitt 5.3.2.

Die erste SHOULD Anforderung ist der Grund warum es die Kritik der Presse an Google gab, sie würden das Versprechen der Out-of-the-Box Verschlüsselung wieder zurückziehen. 1.1

Es ist eher wahrscheinlich das Google versucht hat den Herstellern mehr Zeit zu geben dieses neue Feature zu implementieren. Dasselbe Szenario tritt später bei der File Based Encryption in Android 7 auch auf. Ein neues kryptografisches Feature auf einer mobilen Hardwareplattform zu implementieren und dabei deren Ressourcen optimal zu nutzen benötigt Zeit und es wird im CDD auch angemerkt, dass FDE in der Zukunft als MUST Anforderung angegeben wird.[7][S.59]

MUST Anforderungen Android 6.0

1. Unterstütze FDE, falls ein Sperrbildschirm implementiert ist
2. Verschlüssele die /data Partition und die Partition der nicht entfernbaren internen SD Karte des Gerätes
3. Aktivierte Geräteverschlüsselung FDE zu jeder Zeit um Out-of-the-Box Erfahrung zu gewährleisten
4. Verwende AES mit der minimalen Schlüssellänge von 128-Bit
5. Verwende einen symmetrischen Block Cipher Mode, der für Speicherung konzipiert wurde (AES-CBC ESSIV oder AES-XTS)
6. Binde kryptografisch die Passwort Stretching Algorithmen an einen hardwareunterstützten Schlüsseltresor (TEE) falls ein solcher vorhanden ist

MUST NOT Anforderungen 6.0

1. Den Schlüssel zur Verschlüsselung unverschlüsselt auf den Datenträger schreiben
2. Den Schlüssel zur Verschlüsselung unter keinen Umständen vom Gerät versenden

SHOULD Anforderungen 6.0

1. AES verschlüsselt den DEK, falls dieser aktiv nicht gebraucht wird
   - Verwende dabei ein gestretchtes Passwort vom Benutzer welches vom Sperrbildschirm kommt, mit einem langsamen Stretching Algorithmus. (scrypt oder PBKDF2)
   - Verwende ein Standartpasswort (default_password) falls kein Passwort vom Benutzer angelegt worden ist

MAY Anforderungen 6.0

1. Geräte, welche die Anforderungen nicht erfüllen, werden vom Update ausgeschlossen

Wie zu erkennen ist hat sich am CDD zu Android 6 nicht viel geändert. Eine Änderung stellt die dritte MUST Anforderung dar. Diese ist von SHOULD in CDD 5.0 auf MUST in CDD 6.0 geändert worden. Somit wurde das Versprechen definitiv in der Android Version 6.0 erfüllt. Interessanterweise gilt immer noch die jetzt sechste MUST Anforderung, daher falls das Gerät keinen hardwaregebundenen Schlüsseltresor besitzt fällt der Schutz von FDE massiv ab. Hier gibt Google die Verantwortung klar an die Hardwarehersteller ab, da nicht explizit über eine MUST Anforderung solch ein Schlüsseltresor

gefordert wird. Dazugekommen ist eine MAY Anforderung die besagt, falls ein Gerät eine frühere Version von Android ohne FDE besitzt und die Anforderungen nicht erfüllen kann, es von einem Update ausgeschlossen werden kann. [19][S.64-65]

MUST Anforderungen Android 7.1

1. Verschlüssele die /data Partition und die Partition der nicht entfernbaren internen SD Karte des Gerätes
2. Aktivierte Geräteverschlüsselung zu jeder Zeit um Out-of-the-Box Erfahrung zu gewährleisten
3. Implementierung der DirectBoot API, selbst wenn keine FBE unterstützt wird
4. FBE - Gerät muss in den Direct Boot Modus ohne Benutzereingabe booten und Applikationen Zugriff auf den DE Bereich geben
5. FBE - Zugriff auf den CE Bereich darf nur nach Eingabe des Benutzerpasswortes (PIN, Pattern oder Fingerabdruck) erfolgen
6. FBE - DE Bereich Schlüssel müssen kryptografisch auf den gerätespezifischen Root of Trust gebunden sein und Verified Boot muss unterstützt sein
7. FBE - Dateiverschlüsselung verwendet AES mit der minimalen Schlüssellänge von 256 Bit im XTS Modus
8. FBE - Die Verschlüsselung der Dateinamen verwendet AES mit der minimalen Schlüssellänge von 256 Bit im CBC-CTS Modus
9. FBE - AES 256 Bit XTS und AES 256 Bit CBC-CTS müssen implementiert sein
10. FBE - Schlüssel für die Bereiche CE und DE müssen kryptografisch auf den hardwaregebundenen Schlüsseltresor gebunden werden. CE Schlüssel müssen dabei von einer Benutzereingabe abgeleitet werden und falls keine existiert von einem Standartpasswort (default_password)
11. FBE - Schlüssel für CE und DE müssen einzigartig sein, daher CE und DE Schlüssel von unterschiedlichen Benutzern dürfen nicht gleich sein
12. FDE - Verwende AES mit der minimalen Schlüssellänge von 128-Bit
13. FDE - Verwende einen symmetrischen Block Cipher Mode, der für Speicherung konzipiert wurde (AES-CBC ESSIV oder AES-XTS)

14. AES verschlüsselt den DEK, falls dieser aktiv nicht gebraucht wird und benutze dabei ein gestretchtes Passwort vom Benutzer weclhes vom Sperrbildschirm kommt mit einem langsamen Stretching Algorithmus. (scrypt oder PBKDF2)

15. FDE - Binde kryptografisch die Passwort Stretching Algorithmen an einen hardwareunterstützten Schlüsseltresor (TEE) falls ein solcher vorhanden ist

MUST NOT Anforderungen 7.1

1. FBE - Gerät darf Zugriff auf den CE Bereich erlauben ohne das der Benutzer sein Benutzerpasswort eingegeben hat

2. FDE - Den Schlüssel zur Verschlüsselung unverschlüsselt auf den Datenträger schreiben

3. FDE - Den Schlüssel zur Verschlüsselung unter keinen Umständen vom Gerät versenden

SHOULD Anforderungen 7.1

1. Alle technischen Anforderungen erfüllen um FBE zu unterstützen

2. FBE - Applikationen, die vor dem Entsperren des Gerätes verfügbar sein sollen (Telefonbuch, Wecker, Messenger) Direct Boot fähig machen

3. Verwende ein Standardpasswort (default_password) falls kein Passwort vom Benutzer angelegt worden ist, um den DEK mit AES zu verschlüsseln

MAY Anforderungen 7.1

1. Geräte welche die Anforderungen nicht erfüllen werden vom Update ausgeschlossen

2. FBE - es können alternative Modi, Schlüssellängen und Chiffren für die Datei- und Dateinamensverschlüsselung implementiert werden

Die Spezifikation von Android 7.1 [20][S.81-82] zeigt die Anforderungen für die neue Geräteverschlüsselung FBE. Zwischen den Version 7.0 und 7.1 des CDD hat sich nur eine Anforderung geändert, nämlich die 14. MUST Anforderung. Diese war bei Version 7.0 noch unter den SHOULD Anforderungen und damit unverändert zu Version 6.0. Generell kann man erkennen das sich seit Android 6 im Bezug auf FDE nichts mehr geändert hat. Wieder gilt die Verantwortung der Gerätehersteller wenn es um den hardwaregebundenen Schlüsseltresor geht. Ohne diesen fällt FDE in Sachen Schutz gewaltig ab. Das Brisante an dieser Entwicklung ist, falls es Schwachstellen auf Seiten der Hardware gibt sind diese sehr schwer zu

beheben. Natürlich kann man Sicherheitsupdates via Software entwickeln und verteilen, dennoch bleibt die darunterliegende Hardware verwundbar und somit anfällig falls das Gerät auf eine anfällige Version von Android zurückgesetzt wird.

Google konzentriert sich dem Anschein nach auf die Verbesserung der Geräteverschlüsselung in Form von FBE. Die Anforderungen für FBE an die Gerätehersteller sind die selben wie die Anforderungen für die ext4 Dateiverschlüsselung unter Linux. AES-256 im XTS Modus für die Datei an sich und AES-256 im CBC-CTS Modus für die Dateinamen. Des weiteren muss ein neuer Modus Direct Boot unterstützt werden, der als einer der Hauptgründe für die Einführung von FBE ist. Direct Boot und die Einführung der CE und DE Areale soll den Benutzer die Möglichkeit geben sein Smartphone nach einem Reboot mit rudimentären Applikationen nutzen zu können ohne sein Gerät mit seinem Passwort zu entsperren. Dabei verlangt Google endlich die Verwendung eines hardwaregebundenen Schlüsseltresors für Speicherung der DE und CE Arealschlüssel als MUST Anforderung, falls FBE unterstützt werden soll, was FBE per Anforderung sicherer macht als FDE.

Google wiederholt aber die Vorgehensweise wie bei Android Version 5.0 und Full Disk Encryption. Die gesamte Implementierung von FBE fällt unter eine SHOULD Anforderung ohne Hinweis dies in absehbarer Zeit in eine MUST Anforderung abzuändern. Einzig die Implementierung der Direct Boot API fällt unter eine MUST Anforderung und genauer geht es dabei nur um den Broadcast der beiden Intents `LOCKED_BOOT_COMPLETED` und `ACTION_USER_UNLOCKED` um Applikationen die Direct Boot fähig wären, zu signalisieren, dass die DE und CE Bereiche verwendbar sind. Eine echte Anfrage auf diese Bereiche wird auf eine FDE unterstützende Weise umgeändert. Diese Maßnahme wird vermutlich deswegen getroffen um verschiedene Versionen der internen Applikationen wie das Telefonieren, Wecker oder Kalender zu verhindern. [20][S.81-82]

Daher sind einerseits die Möglichkeiten der FBE gegeben aber kaum ein Hersteller, außer Google selbst, benutzt sie. Das One Plus 5 des chinesischen Hardwareherstellers OnePlus welches am 20.Juni 2017 mit der Android Version 7.1.1 auf den Markt gekommen ist, besitzt als eines der wenigen Smartphones neben dem Google Pixel und Nexus 5X dieses Feature. Obwohl die älteren Vorgängermodelle OnePlus 3 und OnePlus 3T denselben Versionsstand von Android (7.1.1) wie das neue OnePlus 5 besitzen, implementieren diese Geräte als Geräteverschlüsselung FDE und nicht FBE. [4]

[4] `https://forums.oneplus.net/threads/secure-startup-option-missing-in-oneplus-5.588963/`

2.3 Password Stretching Funktionen

Bevor die eigentliche Geräteverschlüsselung behandelt wird, werden an dieser Stelle die Passwortstreckungsfunktionen (Password Stretching), die für Schlüsselerstellung benötigt werden, behandelt.

Passwörter die vorwiegend aus den ASCII Zeichen der mobilen Tastatur bestehen haben den Nachteil nicht den ganzen möglichen Zeichenraum auszunutzen um ein Passwort mit hoher Entropie zu erstellen. Daher werden Password Stretching Funktionen verwendet, um aus dem Passwort, welches der Benutzer eingibt einen kryptografischen Schlüssel zu erstellen. Für den Rest der Arbeit steht der Begriff Passwort für die Bildschirmsperrmethoden also PIN, Password, Zeichenmuster oder Fingerabdruck. Android verwendet hierfür die Password-Based Key Derivation Function 2 (PBKDF2) und scrypt während der Schlüsselerzeugung. Weiteres in Abschnitt 2.4.4 und 2.5.3.

2.3.1 Password Based Key Derivation Function 2

PBKDF2 ist ein Teil des Public-Key Cryptography Standards (PKCS) und wurde von RSA Laboratories designed und entwickelt.[21]. Das National Institute of Standards and Technology (NIST) empfiehlt in seinem Bericht Special Publication 800-132 die Verwendung der PBKDF2 Funktion zur Ableitung für Master Keys, die wiederum für eine Verschlüsselung verwendet werden. [22]
Die PBKDF2 Funktion benötigt dabei folgende vier Eingabewerte:

- Passwort - ein vom Benutzer eingegebener Input
- Salt - wird pro Benutzer zufällig als weiteres Unterscheidungsmerkmal generiert, erhöht somit die Entropie der Eingabe
- Count - Iterationszähler
- dkLen - Länge des abgeleiteten Schlüssels

Der Kern der PBKDF2 Funktion besteht aus einer Pseudo-Random Function (PRF). Für diese Funktion wird von den beiden erwähnten Dokumenten, RFC 2898 [21][S.8] und der NIST Special Publication 800-132 [22][S.11], ein Hash-based Message Authentication Code kurz HMAC definiert. Im genaueren wird dabei die SHA-1 Hashfunktion benutzt. Prinzipiell könnten andere kryptografische Funktionen ebenfalls verwendet werden, solange diese eine pseudo-zufälligen Ausgabe liefern.

Der Wert von Salt wird als weiteres Unterscheidungs- und Identifikationsmerkmal verwendet, um das Problem bei gleichen Passwörtern von unterschiedlichen Benutzern zu begegnen. Andernfalls könnte man die Benutzer nicht von einander unterscheiden, da der abgeleitete Schlüssel sonst ident wäre. Zusätzlich hat es den Sicherheitseffekt die Anzahl der möglichen Kombinationen für Schlüssel zu erhöhen.

Würde ein Angreifer alle möglichen Schlüssel vorberechnen wollen, muss er dies nicht nur für alle Passwörter machen, sondern auch zusätzlich jedes Passwort mit jeden möglichen Salt Wert vorberechnen, was die Speichergröße dieser vorberechneten Schlüsseltabellen immens erhöht.

Die Eingabevariable Count, also der Zähler ist jener Parameter der den Ableitungsprozess ausbremsen soll. Er definiert dabei wie oft die Pseudo-Zufalls Funktion durchgeführt werden soll um einen Block des gestreckten Schlüssels zu erzeugen. Diese Vorgehensweise würde theoretisch nicht nur einen Angreifer ausbremsen, sondern auch die eigentlichen Besitzer, welche sich am Gerät identifizieren wollen. Optimalerweise sollte man als normaler Benutzer von diesem Bremsmechanismus nichts mitbekommen, andererseits aber einen böswilligen Angreifer abhalten. Der Unterschied bei beiden Personengruppen liegt in der Anzahl der Versuche die sie tätigen, um sich am Gerät anzumelden. Bei einmaligen oder zweimaligen Eingeben des Passworts, merkt man diesen Bremseffekt nicht, da ein Angreifer meist viele Versuche tätigt, bevor er die richtige Eingabe identifiziert wird er der Bremseffekt bei jedem individuellen Eingabeversuch zu tragen kommen. In der NIST Empfehlung [22][S.10] werden für Standardapplikationen 1.000 und für kritische Applikationen 10 Millionen Iterationen spezifiziert. Da die Rechenleistung kontinuierlich ansteigt, muss dementsprechend auch die Anzahl der Iterationen erhöht werden, um den Bremseffekt über die Jahre konstant zu halten. Einer der Nachteile dieser PBKDF2 Funktion ist, dass ihr Fokus nur darauf liegt, stark rechenintensiv zu sein. Daher kann mit geringen Kosten ein Rechensystem erstellt werden, welche die Berechnungen parallelisiert.

2.3.2 scrypt

scrypt wurde von Colin Percival für die Firma Tarsnap entwickelt und erstmals im Mai 2009 auf der BSD-Konferenz als neue Key Derivation Funktion (KDF) präsentiert. Hierbei führt er ein neues Konzept ein, dass im Gegensatz zu PBKDF2, nicht nur gezielt die Berechnung rechenintensiv macht, sondern auch die Kosten der Hardware und den Verbrauch von Hardwareressourcen erhöht. Grundsätzlich kann man eine parallele Programmierung als Angriff nicht ausschließen, aber versuchen die Kosten, welche dieser Ansatz verursacht, zu erhöhen. scrypt nutzt hierzu den Umstand aus, dass Arbeitsspeicher verhältnismäßig teurer ist als reine CPU-Power.
Der Algorithmus von scrypt benötigt den folgenden Input:

- Passwort - ein vom Benutzer eingegebener Input
- Salt - wird pro Benutzer zufällig als weiteres Unterscheidungsmerkmal generiert, erhöht somit die Entropie der Eingabe
- N - CPU/Speicher Kostenparameter

- r - Blockgröße des Speichers
- p - Faktor der Parallelisierung
- dkLen - Länge des abgeleiteten Schlüssels

Zusammengefasst ist scrypt eine Kombination aus den folgenden Algorithmen:

- Salsa20/8 Core - eine Runden reduzierte Variante der Salsa20 Core Stromchiffre
- scryptBlockMix - Block-Mix-Algorithmus mit Salsa20/8 Core als Hash-Funktion, die wiederum als Hash-Funktion in scryptROMix dient
- scryptROMix - Algorithmus für die Berechnung von großen zufälligen Werten und das der Zugriff auf diese Werte zufällig geschieht, verwendet als Hashfunktion scryptBlockMix
- PBKDF2 - die bereits erwähnte PBKDF2 Funktion diesmal mit HMAC SHA256 als Pseudo-Zufallsfunktion

Um den abgeleiteten Schlüssel mit scrypt zu berechnen, sind einige Schritte und die oben erwähnten Funktionen zu kombinieren. Zuerst wird mit der PBKDF2 Funktion begonnen. Diese bekommt als Eingabeparameter den Counter auf 1 gesetzt und die Länge des Ausgabeparameters dkLen wird auf das Produkt von p * 128 * r gesetzt.

Der errechnete Output ist die Konkatenation der Blöcke 0 bis p - 1. Auf jeden Block wird der Algorithmus scryptROMix ausgeführt. Dadurch wird innerhalb von scryptROMix als Hashfunktion die Funktion scryptBlockMix aufgerufen, welche wiederum die reduzierte Stromchiffre Salsa20/8 als Hashfunktion verwendet.

Die Werte N, p und r sollen hierbei so gewählt werden um sich mit der Zeit den technischen Entwicklungen anzupassen. Dabei empfahl Colin Percival bei der Veröffentlichung für normale Anforderungen die Werte N=16384, r=8, p=1 und für kritische Anforderungen N=1048576, r=8, p=1.

2.4 Android Full Disk Encryption

Die Geräteverschlüsselung die in Android 4.4 eingeführt und bis Android 7.0 verwendet wird, nennt sich Full-Disk Encryption (FDE) und ist eine volle Festplattenverschlüsselung. Wie der Name Full-Disk Encryption vermuten lässt, sollte der ganze Festplattenspeicher des Android Smartphones verschlüsselt werden, was nicht der Fall ist, da nur die `/data` und `/sdcard` Partition verschlüsselt wird. Über die Anforderungen für die FDE wurde in Abschnitt 2.2.1 geschrieben. Aus diesen Anforderungen lässt sich

auch ein kleiner Einblick in die Funktionsweise von FDE gewinnen. Die nachfolgende Beschreibung der Funktionsweise der FDE ist an die Arbeit von Oliver Kunz [23][S.13-17] angelehnt, wo dieser die Vorgehensweise für Android in Version 5.0 beschrieben hat. Die Beschreibung wird nachfolgend auf Android 7.1 angepasst, dadurch ist der relevante Branche `android-7.1.1_r11`, welcher durchgesehen wird.

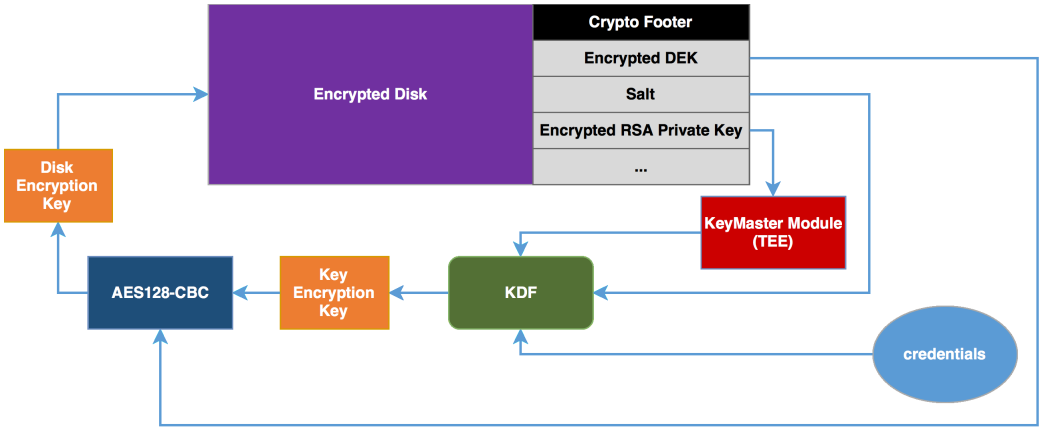

Abbildung 2.2: Übersicht der Komponenten von der FDE Verschlüsselung 2.4.[24]

2.4.1 Device-Mapper und dm-crypt Funktion

Device-Mapper (dm) ist ein Teil des Linux Kernels seit der Version 2.6. Dieses vielseitige Modul wird nicht nur in der FDE verwendet, sondern hat einen allgemeineren Fokus. Seine Kernfunktionalität erlaubt die Erzeugung virtueller blockorientierter Geräte, indem er deren Adressbereich auf andere blockorientierte Geräte abbildet. Der Device Mapper stellt einige Funktionen zur Verfügung, die von LVM benötigt werden und in früheren Linux-Versionen integraler Bestandteil von LVM waren. Darunter fallen die Erzeugung und Verwaltung der blockorientierten Geräte, Snapshots (inklusive Zurückschreiben der Änderungen ins Ursprungsgerät ("Merge")) sowie diverse RAID-Funktionen (insbesondere Striping (Level 0) und Mirroring (Level 1)). Dank der Herauslösung aus LVM können diese Funktionen nun auch mit anderen blockorientierten Geräten (z.B. Festplatten(partitionen) und loop devices) genutzt werden.

dm-crypt [5] ist eines dieser Geräte und hat die Aufgabe verschlüsselte Blöcke transparent auf die physischen Geräte abzubilden. Transparent ist hier so zu interpretieren, dass Lese-Operationen nach dem auslesen des physischen Blockes entschlüsselt werden und Schreib-Operationen verschlüsselt werden bevor man diese auf die Platte schreibt. Der Vorteil dieser Vorgehensweise ist, dass die oberen Schichten des Betriebssystems und dessen Applikationen sich nicht um die Ver- oder Entschlüsselung kümmern müssen. Dadurch ergibt sich wiederum der Nachteil, dass Teile des kryptografischen Schlüsselmateri-

[5] `https://gitlab.com/cryptsetup/cryptsetup/wikis/DMCrypt`

als solange wie das verschlüsselte Gerät gemountet ist, im Hauptspeicher gehalten werden müssen, was vor der Einführung des hardwaregebundenen Schlüsseltresors, die Verschlüsselung anfällig auf RAM Attacken, wie der Cold-Boot Attacke 5.3.5, gemacht hat.

Das Design von dm-crypt ist relativ flexibel in der Verwendung von unterschiedlichen kryptografischen Funktionen. Die im Linux Kernel verwendete Krypto API wurde als Basis für dm-crypt verwendet, daher können alle dort implementierten Funktionen potentiell benutzt werden.[6]

2.4.2 Schlüsselmanagement FDE

Die Full Disk Encryption von Android implementiert eine 2-Stufen Hierarchie wie in Abbildung 2.3 zu erkennen ist. Die erste Stufe ist der Disk Encryption Key (DEK), auch genannt Master Key, der verwendet wird die jeweiligen Partitionen der Festplatte zu verschlüsseln. Die zweite Stufe wird benötigt um den DEK zu schützen, daher wird dieser Schlüssel Key Encryption Key (KEK) genannt und verschlüsselt den Master Key (DEK).

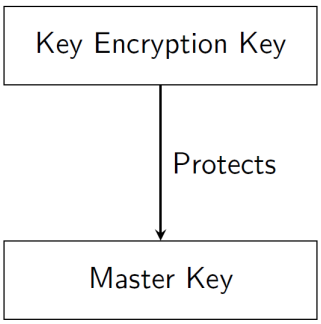

Abbildung 2.3: Übersicht der Schlüsselhierarchie.[23][S.14]

Das Erzeugen der Schlüssel wird von der Funktion `create_encrypted_random_key` [25][Zeile 1494] erledigt. Dabei liest es den Master Key und den Salt, welche beide 16 Byte (128-Bit) groß sind, von `/dev/urandom` eines unter Linux verwendeten Standardinterfaces welches Pseudo-Zufallszahlen generiert. Eines der interessanten Dinge, die in Version 7.1 noch immer gelten ist, dass diese eingelesenen Werte statisch bleiben. Der Master Key und der Salt bleiben statisch erhalten, solange die verschlüsselte Partition nicht gelöscht und danach neu verschlüsselt wird, was bei einigen Richtlinien innerhalb von IT Firmen möglicherweise nicht berücksichtigt wird. Um einen neuen Master Key und Salt zu erhalten muss ein Factory Reset durchgeführt werden, was zum Verlust der Daten führt.

Um die Sicherheit des Master Keys zu garantieren muss der KEK mindestens genauso kryptografisch stark sein. Das Generierung des 128-Bit großen KEK involviert ein Passwort und eine KDF Funktion.

[6] `http://www.chronox.de/crypto-API/crypto/index.html`

In Android 7.1 stehen dazu die bereits beschriebenen Funktionen PBKDF2 (Abschnitt 2.3.1) und scrypt (Abschnitt 2.3.2) zur Verfügung und zusätzlich seit Android 5.0 muss die KDF kryptografisch einen hardwaregebundenen Schlüsseltresor involvieren.

Wie in Abbildung 2.2 zu erkennen ist übernimmt die Funktion des hardwaregebundenen Schlüsseltresors ab Android 5.0 das Hardwaremodul KeyMaster das in einer TEE (Trusted Execution Environment) läuft um sich gegen Offline Brute Force Attacken zu schützen und um das System sicherer zu machen. Die Funktion `scrypt_keymaster` [25][Zeile 1287] implementiert diese Anforderung. Zuerst wird scrypt auf das Passwort angewandt was den Ausgabewert I_KEY ergibt. Danach wird diese Ausgabe dem KeyMaster Hardwaremodul mit der Aufforderung zum signieren übergeben. Das KeyMaster Hardwaremodul, welches in einer TEE läuft, liefert die SIGNATURE welche wiederum in einem erneuten Aufruf von scrypt verwendet wird. Die Ausgabe der zweiten scrypt Funktion ist der KEK welcher anschließend in der Funktion `encrypt_master_key` [25][Zeile 1328] verwendet wird um den Master Key mit AES-CBC zu entschlüsseln. Abbildung 2.4 veranschaulicht diesen Prozess noch einmal visuell.

Die einzelnen kryptografischen Werte Salt, verschlüsselter Master Key, Typ der KDF Funktion werden in der Datenstrukur des Crypto Footers [7] gespeichert. Zusätzlich wird in diesem Crypto Footer ein verschlüsselter RSA Privat Key gespeichert der vom KeyMaster Hardwaremodul geschützt wird.
Wenn von der Standardkonfiguration von FDE ausgegangen wird verwendet der Benutzer kein Passwort auf seinem Gerät. Für diesen Fall existiert das bereits erwähnte Standardpasswort. Dieses ist ein hexadezimal enkodiertes `DEFAULT_PASSWORD` und wird im Quellcode in Zeile 85 definiert. [25][Zeile 85]. Wenn der Benutzer später eine neues Passwort benutzen will, muss eine Schlüsselübersetzung durchgeführt werden. Dies bedeutet das eine Entschlüsselung des Ciphertextes mit dem aktuellen Schlüssel gemacht wird, danach aber die Verschlüsselung davon mit dem neuen Schlüssel. In Android 7.1 übernimmt das die Funktion `cryptfs_changepw` [25][Zeile 3320]. Die Funktion überprüft erst ob der Master Key korrekt entschlüsselt wurde und verschlüsselt dann diesen mit dem neuen KEK, der aus dem eingegeben Passwort des Benutzers abgeleitet wurde, und schreibt danach das Ergebnis zurück in den Crypto Footer.

2.4.3 Verschlüsselung bei initialen Bootvorgang

Wie bereits erwähnt wirkt bei Geräten wie dem Nexus 5X ab der Android Version 5.0 die Verschlüsselung von Beginn an, daher beim initialen Boot durchgeführt. Oliver Kunz hat in seiner Arbeit eine detaillierte Schilderung des Vorganges bei initialen Boot geliefert, die hier für Android 7.1 und FDE gemacht wird. [23][S.16-17] Die System Property `ro.crypto.state` wird auf unencrypted also unverschlüs-

[7] Speicherbereich in Android der wichtige Parameter für die Verschlüsselung speichert

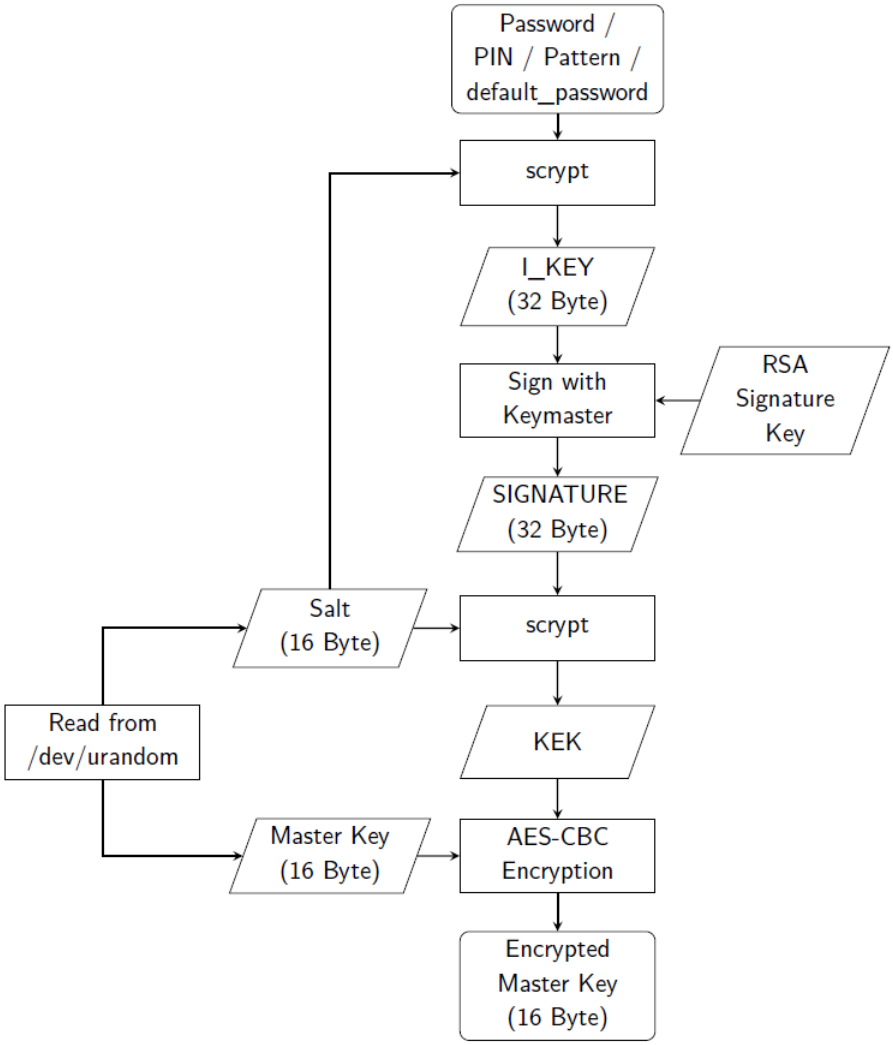

Abbildung 2.4: Übersicht des Schlüsselmanagements.[23][S.15]

selt gesetzt. Wenn das Gerät startet wird die `fstab` Datei ausgelesen, welche die Konfiguration für das Mounten der Partitionen enthält. Wenn innerhalb des Eintrags von userdata das Flag `forceencrypt` gesetzt ist, wird Android den Verschlüsselungsprozess beim ersten Booten starten. Die System Property `vold.decrypt` ist auf `trigger_post_fs_data` gesetzt. Das Android Startscript init.rc [26] beinhaltet eine Aktion für die erwähnte System Property, sodass der Verschlüsselungsservice gestartet werden kann. Bei Verwendung des Standardpasswortes wird die Funktion `cryptfs_enable_default` [25][Zeile 3314] ausgelöst, welche dann die Funktion `cryptfs_enable_internal` [25][Zeile 3316] mit folgenden Parametern aufruft:

- howarg - inplace
- crypt_type - CRYPT_TYPE_DEFAULT
- passwd - DEFAULT_PASSWORD
- allow_reboot_flag - false

Die Funktion `cryptfs_enable_internal` [25][Zeile 2937] koordiniert jetzt alle weiteren Schritte um die verschlüsselte Partition zu erstellen. Zunächst wird es die Hauptklassenservices von init.rc abdrehen, indem es die Property `vold.decrypt` auf `trigger_shutdown_framework` setzt [25][Zeile 3040] und danach alle Partitionen unmountet und die `/data` Partition in ein temporäres Dateisystem (tmpfs) einhängt.[25][Zeile 3063] Das erlaubt dem Framework in der Zeit der Verschlüsselung ohne die Userdaten zu laufen. [25][Zeile 3073] Zu Beginn des Verschlüsselungsprozesses wird der Crypto Footer mit der Funktion `cryptfs_init_crypt_mnt_ftr` [25][Zeile 2141] erstellt. Der Pfad nach dem `forceencrypt` flag in der Datei fstab gibt den Standort des Crypto Footers an. Nachdem der Crypto Footer initialisiert wurde, wird der Master Key wie in Abschnitt 2.4.2 beschrieben erzeugt. Nachdem der verschlüsselte Master Key erzeugt worden ist, wird er mit dem Aufruf der Funktion `decrypt_master_key` [25][Zeile 1474] wieder entschlüsselt.

Als Nächstes wird `create_crypto_blk_dev` [25][Zeile 1154] aufgerufen um ein virtuelles Block Device für dm-crypt zu erstellen. Die Konfiguration für das Device Mapping wird mit der Funktion `load_crypto_mapping_table` [25][Zeile 1083] geladen. Die Konfiguration ist statisch als String im Quellcode definiert `(aes-cbc-essiv:sha256)` [25][Zeile 1995] obwohl im CDD Dokument steht, dass der Benutzer andere Chiffren, Modi und IV Erzeugungseinstellungen verwenden kann, hat er durch diesen String nicht wirklich einen Einfluss.

Nachdem diese Schritte ausgeführt wurden, ruft die `cryptfs_enable_internal` als nächste Funktion `encrypt_groups` [25][Zeile 2368] auf, wo die unverschlüsselte Partition vom physischen Block

Device gelesen wird und von dm-crypt ins virtuelle Block Device geschrieben wird. Dadurch das dm-crypt kryptografisch transparent arbeitet, werden durch jede Schreib-Operation die Daten automatisch verschlüsselt.

Die finalen Schritte umfassen einerseits Aufräumarbeiten für das dm-crypt Mapping und andererseits wird danach der Bootvorgang fortgeführt. Diese Aufräumarbeit wird durch den Funktionsaufruf von `delete_crypto_blk_dev` [25][Zeile 1226] bewerkstelligt. Um den Bootvorgang fortzuführen wird der KEK vom Passwort, in diesem Fall DEFAULT_PASSWORD, abgeleitet um den Master Key zu entschlüsseln und das dm-crypt Device zu erstellen welches in `/data` gemountet wird.[23][S.17]

2.5 Android File Based Encryption

Die Geräteverschlüsselung, die in Android 7.0 eingeführt wird, nennt sich File Based Encryption (FBE). Dabei wird die Verschlüsselung nicht auf Block Ebene stattfinden sondern auf Dateiebene. Daher der Name dateibasierte Verschlüsselung. Wiederum werden nur die `/data` und `/sdcard` Partition verschlüsselt. Über die Anforderungen für die FBE wurde in Abschnitt 2.2.1 geschrieben. Aus diesen Anforderungen lässt sich auch ein kleiner Einblick in die Funktionsweise von FBE gewinnen. Wie erwähnt basiert die FBE auf der ext4 Dateiverschlüsselung für Linux, die seit Kernelversion 4.1 Teil davon ist. [8] Die Funktionalität der ext4 Dateiverschlüsselung wurde aber auf den Linux Kernel 3.10 backported und somit dem Android Projekt verfügbar gemacht.[27] FBE führt dabei das Feature des Direct Boot ein, welches verschlüsselten Geräten erlaubt direkt in den Sperrbildschirm zu booten und ihnen somit die Möglichkeit gibt Daten auszulesen oder Applikationen zu betreiben, ohne das der Benutzer sein Passwort zum entschlüsseln eingeben muss.

2.5.1 Mehrere Verschlüsselungsareale

Um das Feature des Direct Boot zu implementieren werden 2 neue Speicherorte eingeführt, die jeweils anders geschützt werden. Abbildung 2.5 stellt einen Überblick über die beiden Bereiche dar.

Credential Encrypted (CE)

Dieser Bereich steht den Applikationen und dem Gerät erst zur Verfügung wenn vom Benutzer das Passwort geliefert wird. Standardmäßig werden Daten in diesen Bereich gespeichert und nur explizit ausgewählte Applikationen und Daten werden in den anderen Bereich gespeichert.

Device Encrypted (DE)

[8] `https://www.phoronix.com/scan.php?page=news_item&px=EXT4-Changes-Linux-4.1`

Dieser Bereich steht dem Gerät und manchen Applikationen sofort nach dem Einschalten des Gerätes zur Verfügung. Applikationen wie das Telefon, der Wecker, Kalender oder diverse Messenger können ordnungsgemäß funktionieren ohne dass das Gerät entsperrt sein muss.

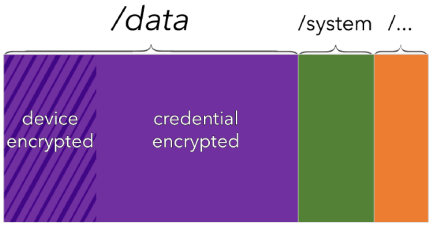

Abbildung 2.5: Darstellung Credential Encrypted und Device Encrypted Bereich.

Durch diese Aufteilung kann man für Applikationen und Benutzer eine Trennung aufgrund deren Profilen vornehmen, was einer Erhöhung der Sicherheit gleichkommt. Daher kann ein Besitzer eines FBE Smartphones beispielsweise mehrere Benutzer anlegen die unterschiedliche Applikationen im DE Bereich speichern.

2.5.2 Schlüsselmanagement FBE

Die File Based Encryption von Android implementiert ebenfalls eine 2-Stufen Hierarchie wie in Abbildung 2.3 zu erkennen ist. Daher existieren wieder ein DEK (Disk Encryption Key auch Master Key genannt) und ein KEK (Key Encryption Key) welcher die Aufgabe hat den DEK zu schützen.

Es wird an der gleichen Stelle wie FDE die Funktion `create_encrypted_random_key` [25][Zeile 1494] aufgerufen und diesmal 2 mal 64 Byte (512 Bit) von `/dev/urandom` eingelesen. DEKs, welche 512 Bit AES-XTS Schlüssel sind werden durch einen 256 Bit AES-GCM Schlüssel verschlüsselt, welcher im TEE gespeichert ist. Dieser Schlüssel ist bekannterweise der KEK (Key Encryption Key) welcher durch die KDF Funktion und dem Passwort des Benutzers abgeleitet wird. Der KEK ist dabei kryptografisch nicht so stark wie der DEK. Wiederum ist die KDF durch das KeyMaster Modul hardwaregebunden und somit von Offline Brute Force Attacken geschützt. Um den KEK im KeyMaster Modul, und somit in der TEE zu bekommen müssen folgende 3 Dinge erfüllt werden: [27]

- Auth Token - Wenn der Benutzer einen erfolgreichen Anmeldeversuch unternimmt muss vom Gatekeeper ein kryptografischer Authentifizierungstoken generiert werden.

- Stretched Credential - Das vom Benutzer eingegebene Passwort muss durch eine KDF (scrypt) gestretched werden.
- secdiscardable hash - Ein 512 Bit großer Hash einer 16 KB großen Datei für jeden einzelnen Benutzer. Diese Datei wird mit einigen anderen Informationen gespeichert, um die Schlüssel der einzelnen Benutzer abzuleiten.

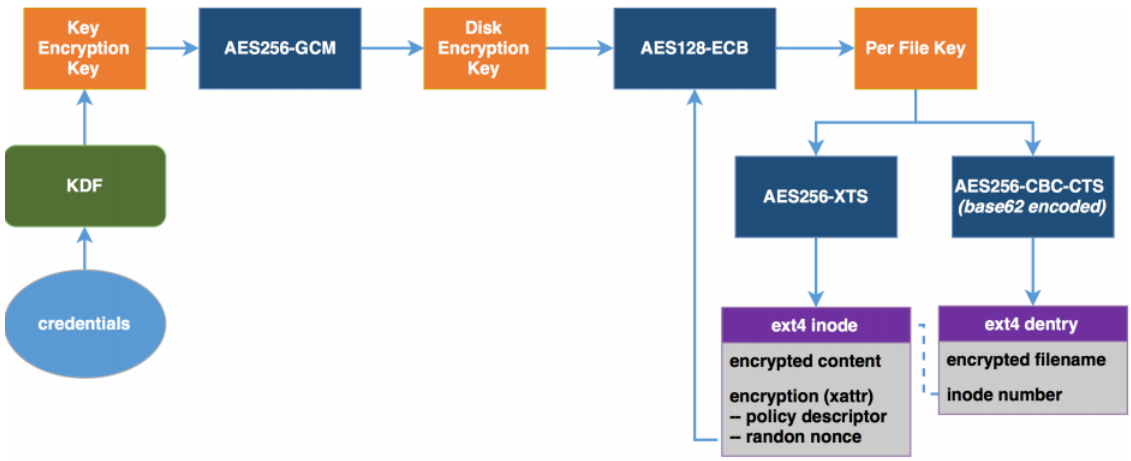

Abbildung 2.6: Überblick der File Based Encryption. [28]

Durch die Verwendung der ext4 Dateiverschlüsselung wird jede Datei mit einen anderen Schlüssel verschlüsselt. Die benötigten Schlüssel werden alle, unter anderem vom DEK, abgeleitet. Zusätzlich besitzt jede Datei ein Nonce die mit ihr verknüpft ist. Diese Nonce, die für jede Datei eine 16 Byte Sequenz darstellt, wird in der dentry [9] von der jeweiligen Datei mit gespeichert. Die einzelnen Schlüssel für die Dateien werden vom DEK und der Nonce der Datei abgeleitet, indem AES 128 Bit Electronic Code Block (ECB) verwendet wird.[28]

Die Namen der einzelnen Dateien werden mit AES 256 Bit CBC-CTS verschlüsselt. Was dabei rauskommt wird Base62 enkodiert. Dadurch wird der Ciphertext einerseits für Menschen lesbar, obwohl er keinen Sinn ergibt und andererseits wird sichergestellt, dass die Dateien Namen erhalten die in ext4 Dateisystem auch gültig sind. Der Inhalt der Dateien wird mit AES 256 Bit XTS verschlüsselt. Abbildung 2.5 soll das Konzept visuell verdeutlichen.

Etwas ähnliches wie ein Crypto Footer kann in Android 7.1 anhand vom Verzeichnis `/data/misc/vold/user_keys` [29][Zeile 67] gefunden werden. Dieses Verzeichnis enthält die zwei Unterordner `ce` und `de`. Diese Ordner enthalten je einen Unterordner für jeden

[9] Ein Objekt im ext4 Dateisystem welches die Inodes von Dateien und deren Dateinamen in Beziehung setzt

Benutzer auf dem Smartphone. Die Verzeichnisse sind nach den UserIDs benannt, daher hat Root die Nummer `0`, der Sekundäruser hat die Nummer `10` und alle weiteren erstellten Benutzer sequentiell aufsteigende Nummern. Schlüsselmaterial für die beiden Bereiche DE und CE sind in diesen Ordnern gespeichert.

```
user_keys
|-- ce
|    |-- 0
|        |-- current
|            |-- encrypted_key
|            |-- keymaster_key_blob
|            |-- salt
|            |-- secdiscardable
|            |-- stretching
|            --- version
--- de
    |-- 0
        |-- encrypted_key
        |-- keymaster_key_blob
        |-- secdiscardable
        |-- stretching
        --- version
```

Abbildung 2.7: Überblick Crypto Footer Android 7.1

Das Verzeichnis `ce` für jeden Benutzer beinhaltet das Material um Zugriff auf den Credential Encrypted Bereich zu erlangen. Dies beinhaltet, wie zu erkennen ist, unter anderem den Salt der für das Entschlüsseln des DEK gebraucht wird. Die Datei `stretching` beinhaltet die Parameter mit denen scrypt aufgerufen wird wenn das Passwort gestretched wird. [30] Der Inhalt der Datei hat folgende Form `scrypt N:r:p` welche noch aus Abschnitt 2.3.2 bekannt ist. Das was nach der KDF rauskommt, ist das bereits erwähnte Stretched Credential, welches für den Zugriff auf den DEK benötigt wird. Der zweite erwähnte Token ist der 512 Bit Hash der Datei `secdiscardable`. Der letzte Token wird vom Gatekeeper nach einen erfolgreichen Anmelden seitens des Benutzers geliefert. Ohne genauere Angaben gefunden zu haben, wird die Datei `encrypted_key` der Master Key sein und die Datei `keymaster_key_blob` das FBE Äquivalent des RSA Schlüssels und deren Funktionsweise, wie in Abschnitt 2.4.2 beschrieben, sein. Der Ordner `de` enthält ähnliche Dateien wie der `ce` Ordner, daher wird von einer gleichen Vorgangsweise ausgegangen. Die fehlende Salt Datei lässt sich leicht erklären, da für den Device Encrypted Bereich kein Passwort vom Benutzer gebraucht wird und somit auch nicht für den Zugriff auf den DEK.

3 Verwandte Arbeiten

Dieses Kapitel widmet sich der Betrachtung einiger verwandter Arbeiten, welche hier in Folge zusammengefasst werden. Dabei werden sowohl Arbeiten erwähnt, welche eine breite Basis für die Erstellung der hier vorliegenden Arbeit liefern, als auch auf Arbeiten hingewiesen, welche spezielle, in dieser Arbeit nicht näher behandelte Themen aufgreifen.

An erster Stelle sei die Arbeit [23] von Oliver Kunz erwähnt. Diese Arbeit stellt den Referenzpunkt dieser Arbeit dar, da in seiner Arbeit eine umfassende Untersuchung von Android 5.0 und der Full Disk Encryption beschrieben wird. Die allgemeine Struktur wurde in dieser Arbeit übernommen und auf Android 7.0 angepasst und die FDE neu und aktualisiert beschrieben. In seiner Arbeit werden die Funktionsweise von FDE und bekannte Angriffe gegen diese beschrieben. Unter den Angriffen befindet sich der Ansatz der Semi-Offline Brute Force Attacke, die bis dato einen neuen Ansatz des Offline Brute Force Angriffes darstellte.

Kurz nach der Veröffentlichung der Arbeit von Oliver Kunz, zeigte Gal Beniamini in seinem Blog, [24] dass der Android Schlüsseltresor nicht an die darunterliegende Hardware von Qualcomm Chips gebunden ist. Daher ist es möglich den Disk Encryption Key zu erhalten und damit wieder die Offline Brute Force Attacke durchzuführen, die zuvor durch die Hardwarebindung nicht mehr möglich war. Es wird die Funktionsweise der TEE, deren Architektur und die einzelnen Komponenten beschrieben und wie sie miteinander interagieren. Des weiteren wird der genaue Vorgang des Exploits durch einen Shellcode beschrieben und wie damit der Schlüssel extrahiert werden kann. Der Shellcode nutzt dabei eine Schwachstelle CVE-2016-2431 [31] aus, die Gal Beniamini selber entdeckt hat.

Halderman et al. zeigen in ihrer Arbeit [32] dass es möglich ist, den Verschlüsselungsschlüssel einer Geräteverschlüsselung, der im Hauptspeicher gehalten wird, zu extrahieren, nachdem das Gerät keine Stromzufuhr mehr hatte. Dabei wird beschrieben wie sich der Hauptspeicher physikalisch verhält nachdem der Strom weg ist, wie man den Flüchtigkeitseffekt durch Kühlen des Hauptspeichers verlangsamt und wie sie die Schlüssel im Arbeitsspeicher identifizieren.

Ein ähnliche Arbeit [33] stammt von Tilo Müller, Michael Spreitzenbarth und Felix Freiling, die sich mit dem gleichen Thema, nur für Android Smartphones beschäftigt haben. Sie beschreiben in ihrer Arbeit FROST, ein Recovery Image welches Tools und Programme besitzt um eine Variante der Cold-Boot Attacke auf Smartphones durchzuführen. Es wird dabei in der Arbeit die technischen Grundlagen beschrieben, wie das theoretischen Konzept von FROST aussieht, wie sie FROST praktisch umgesetzt haben und welche Ergebnisse sie beim Testen ihres Tools bekommen haben.

Das lang bekannte Konzept der Evil-Maid Attacke wird von Johann Götzfried und Tilo Müller in ihrer Arbeit [34] aufgegriffen und in einen Kontext für Android Smartphone neu umgesetzt. In dieser Arbeit beschreiben sie zuerst die technischen Grundlagen für FDE und den Hintergrund ihrer Arbeit. Danach zeigen sie ihre beiden Arten der Implementierung der Evil-Maid Attacke für Android und auch die Gegenmaßnahme zu einer Cold-Boot Attacke wie FROST, nämlich die ARMORED Implementierung. Diese Schutzmaßnahme hält die kryptografischen Schlüssel nicht wie sonst im Hauptspeicher, sondern in den Registern der CPU, was das Auslesen via Cold-Boot Attacke nicht mehr möglich macht.

Im Jahr 2016 stellten Thom Does und Mike Maarse in ihrer Arbeit [35] einen Weg vor, die Authentifizierung des Fingerabdrucksensors in Android 6 zu umgehen. Diese Arbeit stellt sich als Referenzwerk für den entwickelten Proof of Concept für Android Version 7.0 in Kapitel 6 dar. Die Autoren beschreiben in ihrer Arbeit zuerst die Komponenten des Fingerabdrucksensors, wie genau die Authentifizierung durch modifizieren des offenen Quellcodes von Google umgangen werden kann und welche Angriffsmöglichkeiten sich daraus ergeben.

4 Bedrohungsanalyse

In diesem Kapitel wird sich der Bedrohungsanalyse für die Geräteverschlüsselung angenommen, die bei der Erklärung der technischen Komponenten identifiziert wurden. Dazu wird ein Bedrohungsmodell (Threat Model) erstellt, für welches im nächsten Kapitel Angriffsszenarien überlegt werden und welche Angriffe dabei eingesetzt werden können.

4.1 Bedrohungsmodell

1. Schützenswerte Ziele

Ungeachtet des Namens den die Geräteverschlüsselungen tragen, wurde in den Abschnitten 2.2.1 2.4 2.5 erkannt, dass nicht die ganze Disk, sondern nur die Benutzerdaten Partitionen `/data` und `/sdcard` verschlüsselt sind. Daher sind die Daten, welche mittels Geräteverschlüsselung geschützt werden sollen, auf diesen Partitionen gespeichert. Es gibt keinen Schutz der Daten auf den anderen Partitionen auf dem Datenspeicher. Die Daten und Informatioen die auf `/data` und `/sdcard` gespeichert sind, sind hauptsächlich Dinge welche vom User erstellt und genutzt werden. Zum Beispiel, die installierten Applikationen oder die Dateien auf der intern simulierten SD Karte.

2. Ziel des Schutzes

In den Abschnitten 2.4.2 und 2.5.2 wurde identifiziert das für die Verschlüsselung der Daten ein CBC verwendet wird, welcher nur Datenvertraulichkeit (data confidentiality) gewährleistet. Dies stellt eine Möglichkeit dar, um Unbefugte zu hindern, irgendwelche unverschlüsselten Informationen zu bekommen.

3. Mögliche Angreifer

Die Geräteverschlüsselung von Android soll das Schutzziel Datenvertraulichkeit gegenüber einem Angreifer bieten, der physischen Zugriff auf das Smartphone hat. Ein solcher Angreifer könnte das Smartphone auseinanderbauen, einzelne Komponenten der Hardware analysieren,

das Gerät aus oder einschalten und es so lange in Verwendung haben, wie der Angreifer das Gerät benötigt. Solch einem Angreifer wird es immer möglich sein, eine Art von Brute Force Attacke gegen das Gerät durchzuführen, um den Schlüssel oder das Passwort zu bekommen.

4. Mögliche Angriffsszenarien gegen den Datenspeicher

Die Geräteverschlüsselung FDE hat primär das Ziel ruhende Daten der ganzen Festplatte zu schützen, dagegen hat die Geräteverschlüsselung FBE das Ziel Daten in der Bewegung und auch ruhend zu schützen. Wir differenzieren drei Beschaffenheiten der Daten:

- In Bewegung: Daten bewegen sich von einem Ort zum anderen
- In Verwendung: Daten werden in einer Operation genutzt
- In Ruhe: Daten sind auf einer Disk gespeichert

Da dm-crypt verwendet wird um die Benutzerpartition kryptografisch transparent zu halten (Abschnitt 2.4.1), befinden sich die Daten in Verwendung, solange diese Partition entschlüsselt und im Dateisystem befestigt ist. Daten in Ruhe, würde bedeuten, dass die verwendete Partition nicht gemountet ist, was allerdings nur dann zutrifft wenn das Gerät ausgeschaltet ist.

Bei FBE befinden sich die Daten nicht auf Ebene der Partition in diesen 3 Zuständen, sondern auf der Ebene der einzelnen Dateien. Wenn eine Datei bearbeitet wird, sind alle anderen im ruhenden Zustand verschlüsselt. Kommt ein Angreifer in den Besitz eines Passworts von einem der angelegten Benutzer am Gerät, hat er nur Zugriff auf alle Dateien dieses einen Benutzers. Bei FDE existiert im Gegensatz nur ein Schlüssel für die gesamten Benutzerdaten.

5. Angriffe vor denen nicht geschützt werden kann

Die Antworten auf die vorangegangene Frage zeigt die Limitierung von FDE und die quasi Verbesserung von FBE auf. Das Angriffsszenario welches nicht ausreichend geschützt ist, ist wenn ein Angreifer physischen Zugriff auf das Gerät hat, während es eingeschaltet ist. Hat der Benutzer unter FBE mehrere Benutzerkonten angelegt und für verschiedene Benutzerkonten unterschiedliche Passwörter, kann der Angreifer nur jeweils die Daten zu einem Benutzerkonto angreifen. Gelingt der Angriff beispielsweise auf dem Benutzerkonto Freizeit können die Daten des Benutzerkonto Arbeit nicht entschlüsselt werden.

Bei FDE hat der Angreifer nach erfolgreichem Angriff Zugriff auf die ganze `/data` Partition. Das letzte Szenario ist eine installierte bösartige Applikation, welche ihre Privilegien eskaliert

und aus ihrer Sandbox ausbricht. Dies ermöglicht den Zugriff auf die Benutzerdaten auf dem laufenden Gerät. Dieser Fall ist nicht ein Problem der Geräteverschlüsselung an sich, sondern eher ein Problem des Applikationssicherheitsmodells von Android.

5 Bedrohungslage

In diesem Kapitel wird die Bedrohungslage für die Geräteverschlüsselung behandelt, die aufgrund des Stand der Technik, bei Android Geräten gegeben ist und aufgrund der Bedrohungsanalyse relevant sind. Zuerst werden die Angriffsszenarien beschrieben, denen Smartphones üblicherweise in der Praxis ausgesetzt sind. Danach werden die generischen Gegenmaßnahmen aufgezeigt die Android implementiert, um verschiedene Angriffe zu verhindern oder zu erschweren. Zuletzt werden in diesen Kapitel die bekannten Angriffe auf die Geräteverschlüsselung von Android vorgestellt, deren Funktionsweise erklärt, welche Gegenmaßnahmen für den Angriff existieren, und welche Angriffsfläche der jeweilige Angriff besitzt. Dadurch das Android im generellen sehr Linux nahe ist, entsprechen die meisten Angriffe einem ähnlichen Angriff auf die Linux Implementierung der eingesetzten Technologien.

5.1 Angriffsszenarien

Im Kapitel 4 wurde ein Bedrohungsmodell für die Implementierung der Geräteverschlüsselung unter Android identifiziert. Basierend auf diesen Modell, und mit Einbeziehung der Praxis, definieren wir zwei Angriffsszenarien für das Smartphone. Dabei wird ein Szenario durch das Bedrohungsmodell erfüllt sein und das andere nicht. In beiden Szenarien ist der Angreifer im Besitz des Smartphones und hat so viel Zeit wie dieser benötigt, um seinen Angriff durchzuführen. In der Praxis hat der Angreifer meistens das Gerät entweder gefunden, gestohlen oder auf anderen Wege in seinen Besitz gebracht. Dabei gibt es aus praktischer Sicht zwei simple Fälle:

Gerät Ausgeschaltet

Das erste Angriffsszenario sieht vor, dass das Gerät ausgeschaltet ist. Daher bezieht sich dieses Angriffsszenario auf den vierten Punkt der Bedrohungsanalyse. Wenn das Smartphone ausgeschaltet ist und daher die Datenpartitionen nicht im Dateisystem gemountet sind, greift die Verschlüsselung von FDE, da diese genau für diesen Fall entwickelt wurde. Im Falle von FBE

ist das genau gleich. Daher ist es ein Angriffsszenario das vom Bedrohungsmodell erfüllt wird.

Gerät Eingeschaltet

Das praktische, wie auch theoretisch, genaue Gegenteil zum ersten Angriffsszenario ist ein eingeschaltetes Smartphone, welches Bedrohungen birgt die nicht vom Bedrohungsmodell erfüllt werden. Dieser Grund und die Tatsache aus der Praxis, dass keiner sein Smartphone bewusst ausschaltet, machen es zu einem sehr signifikanten Angriffsszenario. Die Daten eines laufenden Smartphone sind dadurch, nicht nur alleine durch die Geräteverschlüsselung FDE oder FBE geschützt. Daher bekommt der Sperrbildschirm Mechanismus eine bedeutende Rolle, da dieser im Schlüsselmanagementprozess von beiden Geräteverschlüsselungen eine Rolle spielt. Daher gibt es diese starke Verbindung von Angriffen gegen den Sperrbildschirm und der darunter arbeitenden Geräteverschlüsselung. Darüber hinaus kann die Geräteverschlüsselung noch so gut funktionieren und sicher sein, wenn der Sperrbildschirm leicht zu umgehen ist, wird dieser Schutz unterwandert.

5.2 Implementierte Gegenmaßnahmen

Alle Android Versionen haben generische und effektive Gegenmaßnahmen implementiert, welche einige der nachfolgend beschriebenen Angriffe abschwächen oder verhindern. Trotzdem hat man wenig Chancen auf Abwehr, wenn der Angreifer forensisch geschult ist und Speicherchips durch Löten aus dem Gerät herauslösen kann.

5.2.1 Gesperrter Bootloader

Smartphones können ähnlich wie ein Desktop PC in den Bootloader[1] gestartet werden. Dazu muss beim Starten des Smartphones eine Kombination von Tasten gedrückt werden. Bei dem Nexus 5X sind das die Power Taste und Lautstärke verringern Taste. Geräte die in den Bootloader Modus gestartet wurden, können von jedem Host PC, der mittels USB Kabel mit dem Smartphone verbunden ist, via dem Tool fastboot angesprochen werden. Dieses Tool ist wie adb ein Teil des Android Software Development Kit (SDK) und kann verwendet werden, um von einem spezifischen Image zu booten oder ein neues Image auf das Gerät zu flashen. [2]. Dies ist nur dann möglich, wenn der Bootloader entsperrt ist. Der Umstand das der Bootloader gesperrt sein

[1] Bootloader bezeichnet ein Programm in dem ausgewählt werden kann, welches Betriebssystem geladen werden soll

[2] flashen bezeichnet den Vorgang ein altes Image durch ein neues gänzlich zu ersetzen

sollte, wird nicht im aktuellen CDD [20] von Android Version 7.1 erwähnt. Es wird nur angegeben, dass die Funktion `PersistentDataBlockManager.getFlashLockState()` korrekterweise angeben soll, ob der Bootloader das flashen des Systemimages erlaubt oder nicht.[20][S.82]

Erfahrungsgemäß wird ein Smartphone mit gesperrten Bootloader ausgeliefert, da der durchschnittliche Benutzer die Funktionalitäten des Bootloader selten bis gar nicht brauchen wird. Es ist trotzdem vorgekommen, dass eine Gerätereihe (Samsung Galaxy S II) mit entsperrten Bootloader ausgeliefert wurde.[3]

Das Sperren oder Entsperren vom Bootloader kann mit dem erwähnten Tool fastboot erledigt werden. Dazu muss das Gerät mit einem Host PC der das Android SDK installiert hat verbunden werden, das Smartphone in den Bootloader gebootet werden, und mit dem Tool der Befehl /textttfastboot oem unlock ausgeführt werden. Das Smartphone wird eine Bestätigungsmeldung am Bildschirm anzeigen, welche bestätigt werden muss, da das Entsperren eine Löschung der /userdata Partition auslösen wird. Interessanterweise gibt es den Fall, das diese Löschung nicht richtig durchgeführt wird. Eine Sicherheitsanalyse der University of Cambridge hat gezeigt, dass diese Löschung aufgrund von falscher Implementierung und falscher Updatemechaniken seitens der Gerätehersteller fehlschlagen kann. [36].

Die daraus resultierende Konsequenz lautet, dass ein Angreifer, der ein Custom Image booten oder auf das Gerät flashen will, zuerst denn Bootloader entsperren muss. Wenn das für das anzugreifende Gerät der Fall ist, oder der Angreifer sicher sein kann, das die Löschoperation fehlschlagen wird, ist es dem Angreifer möglich, fortzufahren ohne zu riskieren die Daten zu verlieren. Trifft dies nicht zu, was meist der Fall ist, muss der Angreifer vorher die /userdata Partition kopieren. Siehe Abschnitt 6.3. Wie erwähnt, schützt das nicht vor Personen, die das Wissen und die Werkzeuge besitzen, um die Hardware beispielsweise mit einem JTAG Interface [4] oder dem Herauslöten der Chips direkt anzugreifen.

5.2.2 Deaktivierte Android Debug Bridge

Die Android Debug Bridge (adb) ist ein weiteres wichtiges Tool des Android Software Development Kits. Im Gegensatz zu fastboot, muss bei diesem Tool das Smartphone nicht in einem

[3] `https://forum.xda-developers.com/galaxy-s2/help/faq-gt-i9100-galaxy-s-ii-faq-thread-t1046748/`

[4] Joint Test Action Group (JTAG) eine Methodik zum Testen und Debuggen von integrierten Schaltungen auf Leiterplatten

speziellen Zustand gebootet werden, um mit dem Gerät zu interagieren. Dieses Tool ist eines der vielseitigsten und mächtigsten im Android SDK. Es erlaubt unter anderem, dem Benutzer das verschieben von Dateien vom Host PC auf das Gerät und vice versa. Die wohl wichtigste Funktion von adb ist die Ermöglichung, Befehle an die Shell des Gerätes zu senden oder eine Remote Shell auf das Gerät zu öffnen. Bei der Bedienung dieser Shell zeigt sich erneut die Ähnlichkeit von Android zu Linux, da die Befehle die gleichen sind wie bei einer bash. Der adb Daemon[5] startet die Remote Shell ohne Root Rechte, daher können keine Befehle, Funktionen ausgeführt oder Dateien und Ordnerstrukturen verwaltet werden, die Root Berechtigungen benötigen. Auf einem gerooteten Smartphone kann durch `su` jedoch eine Root Shell verwendet werden. Siehe Abschnitt 6.2.2 für weitere Informationen.

Auch ohne Root Berechtigungen ist die adb Shell mit der UID 2000, ein mächtiges Werkzeug, da die Dateien der `/userdata` Partition und der `/sdcard` Partition einfach verwaltet werden können. Android wird standardmäßig ohne laufenden adb Daemon gestartet und diesen Dienst zufälligerweise auf zu drehen ist sehr unwahrscheinlich, da der Benutzer sehr spezielle Aktionen ausführen muss, um das zu tun. Zuerst muss der Benutzer 2 Einstellungen in den Systemeinstellungen vornehmen und danach noch einen Bildschirmhinweis bestätigen, ob er das wirklich möchte.

1. Aktivieren der Entwickleroptionen - Der Benutzer muss 7 mal im Gerätemenü auf die Build Nummer seine Gerätes tippen, um den geheimen Menüpunkt Entwickleroptionen zu erhalten

2. Aktivieren des USB Debuggings - In den Entwickleroptionen muss die Checkbox USB Debugging aktiviert werden

Android hat ein zusätzliches Sicherheitsfeature ab der Version 4.2.2 eingebaut. Vor dieser Version konnte man nach der Durchführung der oben genannten Schritte jeden beliebigen Host mit installierten adb Tool mit dem Gerät verbinden. Um dies zu verhindern, wurde ein ähnliches Konzept wie bei SSH eingeführt. Bei der ersten Verbindung mit dem Host PC, nachdem der adb Daemon auf dem Gerät gestartet wurde, erscheint eine Benachrichtigung auf dem Bildschirm, die den RSA Key Fingerprint des Hosts anzeigt und es den Benutzer überlässt diesem Host zu vertrauen oder nicht und demnach die Verbindung zu akzeptieren oder nicht. Zusätzlich wird die Option gegeben, diese Informationen zu speichern, damit nicht beim erneuten

[5] Daemon lautet die Bezeichnung für Hintergrundprozesse die bestimmte Dienste zur Verfügung stellen

Verbinden oder dem Neustart des adb Daemons, die selbe Meldung noch einmal erscheint. Bei Smartphones die keinen Sperrbildschirm oder nur den Swipe Bildschirm konfiguriert haben, ist diese Host Authentifizierung sehr leicht zu umgehen, da ein Angreifer nur physischen Zugang zu dem Gerät benötigt und danach mit dem adb Tool machen kann was er will.

5.2.3 TEE

Seit Android 5.0 hat es den Support für hardwaregebundene Schlüsseltresore, die KeyMaster genannt werden, gegeben. Dies wird durch eine sogenannten TEE (Trusted Execution Environment) implementiert. Die TEE ist ein sicherer Bereich des Hauptprozessors, und wird dabei komplett vom Betriebssystem Android abgeschottet. Das KeyMaster Hardwaremodul ist dazu gedacht, den Schutz von kryptografischen Schlüssel zu gewährleisten, welche von Applikationen generiert und benötigt werden. Das KeyMaster Module kann kryptografische Schlüssel generieren und kryptografische Operationen auf diese ausführen, ohne dass das restliche Betriebssystem Zugriff darauf hat. Sobald die Schlüssel generiert wurden, werden sie verschlüsselt und wieder an Android zurückgesendet. Will Android beispielsweise eine Operation (Daten signieren) durchführen, welche die kryptografischen Schlüssel im TEE involviert, sendet Android den verschlüsselten Schlüssel zum KeyMaster Modul. Dieser entschlüsselt den Schlüssel, signiert die Daten, und schickt das Ergebnis zurück. Um den Key Encryption Key (KEK) resistent gegen Offline Brute Force Attacken zu machen, wurde ab Android 5.0 im Key Derivation Function Prozess (KDF Prozess) eine Signierung des Schlüssels mit einem Schlüssel aus dem KeyMaster Hardwaremodul vorgesehen. Das ist der RSA-2048 Private Key, welcher im Crypto Footer gespeichert wird.

Durch den Einsatz von TEE können Offline Brute Force Attacken abgewehrt werden, welche eine gefährliche Art des Brute Force Angriffes darstellen, da dort nach initialer Arbeit, dass physische Smartphone nicht mehr gebraucht wird.

Der interessante Aspekt ist jedoch, was passiert falls es genau in dieser Komponente hardwaretechnische Sicherheitslücken gibt, da diese einerseits tausende von Geräte betreffen würden und zwar softwaretechnisch gepatched werden können, aber die fehlerhafte Hardware trotzdem verbaut bleibt.[24] Siehe Abschnitt 5.3.4.

5.3 Bekannte Angriffe auf die Implementierung der Verschlüsselung

Nachfolgend werden die Beschreibungen der bekannten Angriffe auf die Implementierung der unterschiedlichen Geräteverschlüsselungen behandelt. Diese Auflistung ist nicht als vollständig zu betrachten.

5.3.1 Online Brute Force Angriff

Das Konzept einer Exhaustive Password Search Attack, auch bekannt als Brute Force Angriff, ist weder neu noch theoretisch recht kompliziert. Dabei wird versucht, das Passwort durch ausprobieren sämtlicher möglichen Kombinationen zu finden. Da im schlechtesten Fall dieser Vorgang manuell gemacht werden muss, wird generell versucht, diese Schritte zu automatisieren. Die simpelste Variante dabei ist, linear durch möglichen Passwortraum zu gehen und hoffen, dass das Passwort früher als später gefunden wird. Die Bezeichnung Online bezeichnet dabei die Eingabe der möglichen Passwörter am Gerät selber, als würde der Benutzer ein Passwort am Gerät eingeben. Es ist möglich diesen Angriff einmal bei der Passwortabfrage im Bootvorgang anzuwenden oder wenn das Smartphone normal im Betrieb ist. Dabei sollten beide Angriffsszenarien abgedeckt sein, da bei einer Abfrage im Bootvorgang, das Gerät vorher ausgeschaltet sein muss. Die ersten Versuche können noch manuell getätigt werden. Dabei werden schnell mehrere Dinge klar. Einerseits greift der bereits erwähnte Gatekeeper Dienst ein und verlangsamt ab der 5. falschen Eingabe (30 Sekunden Time Out) das weitere Vorgehen und andererseits ist es eine stumpfsinnige Arbeit per Hand die Kombinationen durchzuprobieren, da diese Methode für Menschen sehr fehleranfällig ist. Daher greift man klassischerweise auf einen automatisierten Ansatz zurück.

5.3.1.1 Angriffsanalyse

Wie beschrieben, können Android Smartphones mit dem adb Tool gesteuert werden. Daher kann mit Hilfe dieses Tools der ganze Angriffsprozess automatisiert werden, da somit Angriffsskripte geschrieben werden können, in welchen die Benutzereingaben simuliert werden. Mit dem Befehl `adb input` können die verschiedensten Eingaben simuliert werden. Dadurch kann ein automatisiertes Skript geschrieben werden, welches alle Möglichkeiten durchprobiert. Es gibt sehr viele Möglichkeiten, in welcher Form ein solches Angriffsskript aufgebaut werden kann.

Es wird hier vom theoretischen Fall ausgegangen, dass ein PIN mit vier Ziffern gesetzt wurde. Das schwierige Element bei einem automatisierten Skript ist, dass es kein Auge hat um den Bildschirm des Gerätes zu beobachten und festzustellen, ob ein Versuch richtig war oder falsch. Eine Lösung ist es den Kommandozeilenbefehl `dumpsys` bei jedem Versuch aufzurufen und das Ergebnis auf den String `mLockScreenShown` zu überprüfen. Ist dieser vorhanden, war der Versuch erfolglos und der Bildschirm bleibt gesperrt. Der nächste wichtige Befehl, der benötigt wird, lautet `sleep` um die Time Out Zeiten des Gatekeeper Dienstes im Angriffskript einplanen zu können. Somit besitzt man alle Bauteile, um das Skript laufen zu lassen.

5.3.1.2 Gegenmaßnahmen

Bei dem Versuch einen Online Brute Force Angriff im Bootmodus zu machen wird man schnell an die Grenzen dieses Vorhabens stoßen, da nach dem 29. Fehlversuch die Warnung vom Gerät angezeigt wird, dass nach dem 30. Fehlversuch eine Systemlöschung durchgeführt wird. Daher limitiert sich in diesem Szenario die Anzahl der Versuche auf 29 pro Einschaltung des Gerätes. Der Angriff, und damit das Angriffsskript, müssen auf diesen Umstand angepasst werden, um sinnvoll einen Angriff im Bootmodus zu tätigen.

Unter Android Version 5.0 bekommt man durch den Gatekeeper nur nach jedem 5. Fehlversuch ein Time Out von 30 Sekunden. Dadurch war es Oliver Kunz möglich in seiner Arbeit ein Online Brute Force Skript zu erstellen[23][S.28-29]. Mit diesem Skript konnte er einen 4 stelligen PIN in den Zeiten, welche man der Abbildung 5.1 entnehmen kann, erraten. Korrekterweise, muss man anmerken, dass man diese Zeiten verbessern könnte, indem man das durchgehen des PIN Raumes einer Strategie anpasst, beispielsweise zuerst die Kombinationen der Geburtsjahre von 1950 bis 2010 prüft oder besonders beliebte Kombinationen und Zahlenmuster.

PIN	Number of Timeouts	Duration of Attack	
1986	397	16 163 s	(∼4 h 30 min)
5000	1 250	40 826 s	(∼11 h 20 min)
9999	2 000	81 549 s	(∼22 h 40 min)

Abbildung 5.1: Online Brute Force Skript Zeiten für Android 5.0 [23][S.29]

Diese Zeit ist unter Android 6.0 nicht mehr möglich da, von Google Änderungen durchgeführt wurden, sodass der Gatekeeper nach dem 10. Fehlversuch jedes mal 30 Sekunden ein Time Out

verordnet, was die Zeit für den Worst Case von PIN 9999 auf theoretische 3,4 Tage befördert, wenn man inkrementell durch den PIN Zahlenraum geht. Siehe Abbildung 5.2 für die Time Out Logik des Gatekeepers von Android 6.0. Dennoch ist es in einer angemessenen Zeit machbar, falls der Brute Force Algorithmus optimiert wird.

```
245 uint32_t GateKeeper::ComputeRetryTimeout(const failure_record_t *record) {
246     if (record->failure_counter > 0 && record->failure_counter <= 10) {
247         if (record->failure_counter % 5 == 0) {
248             return 30000;
249         }
250     } else {
251         return 30000;
252     }
253     return 0;
254 }
```

Abbildung 5.2: Gatekeeper Logik Android 6.0 [37][Zeile 245]

Bei Android 7.0 wird allerdings ein etwas anderer Ansatz zum Einsatz gebracht. Siehe Abbildung 5.3 für die Time Out Logik von Android 7.0.
Hier kann man erkennen, dass nach dem 5. und 10. Fehlversuch ein Time Out von 30 Sekunden auftritt. Nach jedem weiteren Fehlversuch bis zum 30. Versuch bekommt man einen Time Out von 30 Sekunden. Zwischen dem 30. und 140. Fehlversuch steigert sich der Time Out mit exponentiellen Wachstum von 32 Sekunden bis zu 61440 Sekunden was 17 Stunden und 4 Minuten entspricht. Nach dem 140. Fehlversuch ist der Time Out für jeden weiteren Fehlversuch 1 Tag. Um das Worst Case Beispiel von 9999 zu verwenden, würde das 27 Jahre benötigen. Daher müsste bei Android 7.0 der Algorithmus stark optimiert werden, um eine Chance zu haben.

5.3.1.3 Angriffsfläche

Prinzipiell sind alle Versionen von Android und alle Angriffsszenarien anfällig gegen diesen Angriff, wobei zwischen Gerät Eingeschaltet und Gerät Ausgeschaltet ein großer Unterschied besteht. Die Deaktivierung von adb verhindert allerdings schon den Angriff. Wirklich effektiv ist der Angriff bei den Android Versionen vor 7.0 und wenn der Brute Force Algorithmus optimiert wurde.

5.3.2 Offline Brute Force Angriff

Dieser Abschnitt beschreibt die Offline Ansatz des Brute Force Angriffs, dessen Hauptunterschied darin liegt dass die Verzögerungsmechanismen des physischen Gerätes umgangen wer-

```
247 /*
248  * Calculates the timeout in milliseconds as a function of the failure
249  * counter 'x' as follows:
250  *
251  * [0. 5) -> 0
252  * 5 -> 30
253  * [6, 10) -> 0
254  * [11, 30) -> 30
255  * [30, 140) -> 30 * (2^((x - 30)/10))
256  * [140, inf) -> 1 day
257  *
258  */
259 uint32_t GateKeeper::ComputeRetryTimeout(const failure_record_t *record) {
260     static const int failure_timeout_ms = 30000;
261     if (record->failure_counter == 0) return 0;
262
263     if (record->failure_counter > 0 && record->failure_counter <= 10) {
264         if (record->failure_counter % 5 == 0) {
265             return failure_timeout_ms;
266         }  else {
267             return 0;
268         }
269     } else if (record->failure_counter < 30) {
270         return failure_timeout_ms;
271     } else if (record->failure_counter < 140) {
272         return failure_timeout_ms << ((record->failure_counter - 30) / 10);
273     }
274
275     return DAY_IN_MS;
276 }
```

Abbildung 5.3: Gatekeeper Logik Android 7.1 [38][Zeile 247]

den. Daher soll der Name Offline andeuten, dass der Angriff nicht auf dem Zielgerät ausgeführt wird, sondern auf einen separaten meist leistungstechnisch stärkeren Angriffssystem. Ziel bleibt weiterhin das Passwort zu finden, welches in der KEK Erstellung und den Sperrbildschirm verwendet wird. Der Angriff ist wohlgemerkt im Groben nur für Android Geräte unter Version 5.0 relevant, was zum Stand der Erstellung dieser Arbeit immerhin noch 26,6 % aller Geräte laut Google sind.2.1. Diese Versionen von Android binden den KEK nicht kryptografisch an das Gerät selber 2.4.2, daher kann ein Offline Brute Force Angriff durchgeführt werden. Für die Durchführung des Angriffs benötigt man das Android Gerät nur so lange wie es zum Imagen der Partitionen `userdata` und `metadata` benötigt.

1. die Metadaten Partition, die den Crypto Footer enthält
2. die Userdaten Partition

Verschiedene Ansätze wie das Imaging dieser Partitionen funktionieren kann wird in Abschnitt 6.3 beschrieben.

5.3.2.1 Angriffsanalyse

Die Linux Distribution Santoku Linux wird mit einem Proof of Concept Python Skript `bruteforce_stdcrypto.py` ausgeliefert an dessen Entwicklung über die Zeit etliche Autoren (Tom Cannon, Nikolay Elenkov und Oliver Kunz) beteiligt waren.[39] [23][S.32] Dieses Angriffsskript parst den übergebenen Crypto Footer der Metadaten Partition und läuft eine lineare Suche durch den möglichen PIN Eingaberaum. Dabei wird jeder Versuchskandidat durch scrypt mit den Parametern im Crypto Footer aufgerufen. Der abgeleitet KEK, der dabei ensteht, wird dann verwendet um den Master Key im Crypto Footer zu verschlüsseln, um danach mit diesem die ersten 1088 Bytes der Userdata Partition zu entschlüsseln. Wird dabei die Signatur `0xEF53` am Offset `0x1080` festgestellt, gilt das Passwort oder PIN als gefunden da diese Signatur die ext4 Magic Signature darstellt die im ersten ext4 Superblock gefunden werden kann.[6]

5.3.2.2 Gegenmaßnahmen

Google hat das Potential der Offline Brute Force Attacke entschieden mit der Einführung des hardwaregebundenen Schlüsseltresors geschwächt. Dadurch sind nicht mehr alle benötigten

[6] `https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout`

kryptografischen Komponenten im Crypto Footer enthalten und daher ist dieser Angriff wie beschrieben ab Android Version 5 mit dem vorgestellten Brute Force Skript von Santoku Linux obsolet. Dennoch zeigt der Ansatz von Oliver Kunz 5.3.3 auf, das nur die Einführung der Hardwarebindung nicht ausreichend ist.

5.3.2.3 Angriffsfläche

Die Angriffsfläche mit dem Santoku PoC Skript beläuft sich auf Geräte mit den Android Versionen unter 5.0. Dabei hängt es davon ab, ob der Angreifer in der Lage ist, die benötigten Partitionen des Gerätes zu bekommen. Für die Möglichkeiten die Partitionen bei einem eingeschalteten Gerät zu bekommen, siehe Abschnitt 6.3. Für das Angriffsszenario Gerät Ausgeschaltet kann immer noch das JTAG Interface verwendet oder der Chip rausgelötet werden.

5.3.3 Semi-Offline Brute Force Angriff

Oliver Kunz hat in seiner Arbeit einen funktionierenden Ansatz einer Semi-Offline Brute Force Attacke beschrieben, welche die Involvierung des TEE berücksichtigt.[23][S.33-36] Wie bereits erwähnt, änderte sich mit Einführung von Android 5.0 die Ableitung des KEK massiv, da jetzt das KeyMaster Hardwaremodul als hardwaregebundener Schlüsseltresor fungiert und seine Berechnungen in einer TEE durchführt und nicht mehr alle kryptografischen Komponenten auf der Festplatte vorhanden sind. Oliver Kunz adressiert mit seiner Idee der Semi-Offline Brute Force Attacke den Umstand, das im Prinzip das KeyMaster Modul nur zur Signierung in der KDF benötigt wird. Alle anderen kryptografischen Bestandteile, die man zur Entschlüsslung benötigt, kann man als Angreifer, der in Besitz des Gerätes ist, beschaffen. Durch den physikalischen Besitz kann der Angreifer das Smartphone weiter für seinen Angriff verwenden. Durch die Speicherung des öffentlichen verschlüsselten Privat RSA Exponenten und der RSA Modulus im Crypto Footer sind alle Sachen vorhanden, die für das Signieren durch das KeyMaster Modul benötigt werden. Die Hardwaregebundenheit könnte umgangen werden, falls man den verschlüsselten RSA Exponent entschlüsseln kann. Wenn der Schlüssel im KeyMaster Hardwaremodul statisch im Chip fixiert ist, kann das ausgenutzt werden. Daher nennt Kunz diesen Angriff Semi-Offline, da er das anzugreifende Gerät verwendet das KeyMaster Modul zum signieren zu verwenden.[23][S.33-36]

5.3.3.1 Angriffsanalyse

Bei der Erzeugung eines KEK, wird das KeyMaster Modul, und daher das physische Gerät, nur zur RSA Signierung verwendet. Die restlichen beiden Berechnungen mit scrypt können extern auf einem anderen leistungsstärkeren Host ausgeführt werden. Die Grundlage für den Angriff bildet die Datei `cryptfs.c` von AOSP selbst. Mit dem Verständnis dieser Quellcodedatei, war es Kunz möglich eine Proof of Concept Applikation mit dem Aufbautyp einer Client-Server Applikation zu entwerfen.[23][S.33-36]

1. Initialisiere das KeyMaster Modul mit dem Schlüsselmaterial des vorher geimageden Crypto Footer
2. Empfange den ersten intermediate Wert (`I_KEY`) nach der ersten scrypt Funktion
3. Signiere den `I_KEY` mit dem KeyMaster Modul am physischen Gerät
4. Schicke die Signatur weiter
5. Am externen Host, generiere Passwörter Kandidaten
6. Am externen Host, führe die zweite scrypt Operation mit der erhaltenen Signatur durch
7. Am externen Host, entschlüssele den Master Key aus dem Crypto Footer
8. Am externen Host, teste den entschlüsselten Master Key ob dieser richtig ist

Der Server in dieser Client-Server Applikation läuft auf dem Smartphone und horcht auf einen Port um Daten zu empfangen. Die Kommunikation wird in Abbild 5.4 dargestellt. Die erste Nachricht überträgt die Länge des Base64 enkodierten KeyMaster Key Blop aus dem Crypto Footer, gefolgt von der zweiten Nachricht, wo dieser Key Blop übertragen wird. Danach wartet der Server auf die Zwischenwerte, die er mit dem KeyMaster Hardwaremodul signieren und zurückschicken soll. Der Client führt danach die erwähnten Schritte 5. bis 8. durch, bis der Master Key gefunden wurde.[23][S.33-36]
Kunz Skript funktioniert dabei nur für PINs und sein Brute Force Ansatz bleibt das inkrementelle durchgehen des möglichen PIN Raumes. Keine Parallelisierung oder andere erweiterte Ansätze wurden verwendet, da dies nur einen PoC darstellen sollte. Es konnte im Zuge der Entwicklung dieser Applikation von Kunz bestätigt werden, dass der Schlüssel im KeyMaster Hardwaremodul, der den Private RSA Exponenten verschlüsselt, im Hardwaremodul fixiert und damit gleich bleibt. Daher ist er von einem Gerätewipe unbetroffen.[23][S.36]

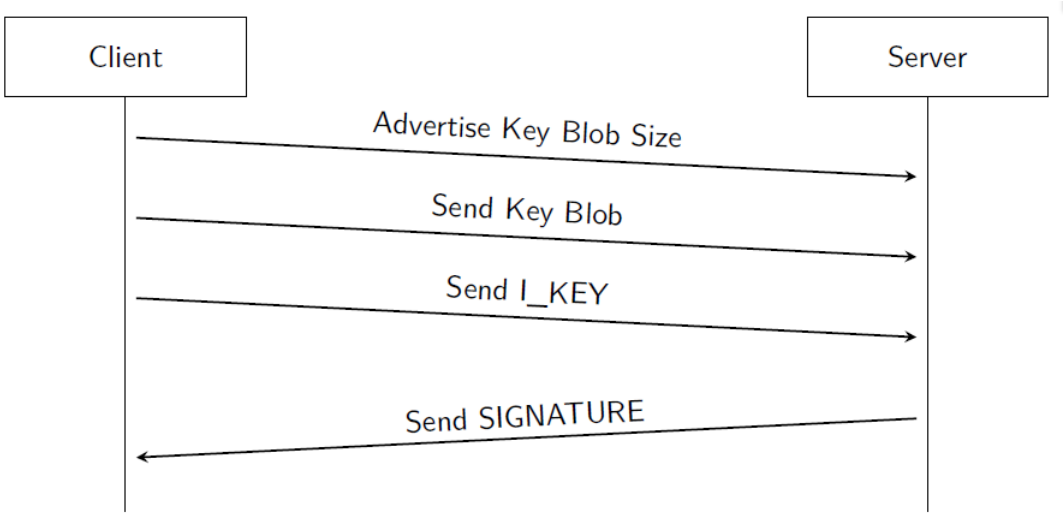

Abbildung 5.4: Client-Server Kommunikation. [23][S.34]

5.3.3.2 Gegenmaßnahmen

Die Semi-Offline Attacke besitzt die gleichen Limitierungen, wie die Offline Brute Force Attacke, daher kommt es darauf an ob der Angreifer in der Lage ist die benötigten Partitionen zu beschaffen. Siehe Abschnitt 6.3. Die Einführung des KeyMaster Hardwaremoduls reicht alleine nicht aus, um das Gerät von Offline Brute Force Attacken zu schützen. Kunz fordert als Gegenmaßnahme, das ein gesperrter Bootloader als MUST Anforderung in die CDD aufgenommen wird.[23][S.35] Das wurde aber, wie bereits in dieser Arbeit behandelt, bis Android Version 7.1 nicht gemacht.2.2.1 Die weiteren Änderungen, welche er vorschlägt, betreffen hauptsächlich die Handhabung des Schlüsselmaterials. Der Angriff war möglich, da der interne Schlüssel im KeyMaster Hardwaremodul, welcher zu Entschlüsselung des Private RSA Exponenten verwendet wird, statisch im Modul gespeichert ist. Es wird wie in Abschnitt 5.3.6 ein Mechanismus zur Schlüsseländerung gefordert, um einen Weg zu erhalten den internen Schlüssel im KeyMaster Hardwaremodul zu ändern.[23][S.35]

5.3.3.3 Angriffsfläche

Es hängt auch hier davon ab ob der Angreifer in der Lage ist, die benötigten Partitionen des Gerätes zu bekommen. Für die Möglichkeiten, die Partitionen bei einem eingeschalteten Gerät zu bekommen, siehe Abschnitt 6.3. Für das Angriffsszenario Gerät Ausgeschaltet kann immer noch das JTAG Interface verwendet oder der Chip rausgelötet werden. Google hat gegenüber

Kunz bestätigt, dass dieser Angriff auf Android 6.0 genauso funktioniert. [7]

5.3.4 TEE Schwachstelle

Wie bereits erwähnt wurde ab Android Version 5.0 das KeyMaster Hardwaremodul eingeführt. Siehe Abschnitte 5.2.3 und 2.4.2 für weitere Informationen. Gal Beniamini hat gezeigt, das der Inhalt des KeyMaster Hardwaremoduls, der ja Teil der TEE ist, nicht an die darunterliegende Hardware gebunden ist.[24] Für Geräte die einen Qualcomm Chip besitzen, existiert damit wieder die Möglichkeit Offline Brute Force Attacken durchzuführen. Qualcomm produziert die internen Chips für die meisten Smartphones der Welt, daher besitzen sie einen Marktanteil von 65%. [40] Durch das Ausnutzen von CVE-2015-6639 [41] und CVE-2016-2431 [31] können die Qualcomm KeyMaster Modul Schlüssel extrahiert werden.

5.3.4.1 Angriffsanalyse

Die Implementierung der TEE von Qualcomm nennt sich Qualcomm Secure Execution Environment. QSEE erlaubt die Ausführung von sogenannten Trustlets (kleine Applikationen) in einem gesicherten Bereich auf dem Hauptprozessor. Trustlets erlauben es damit dem Android Betriebssystem, dass als nicht sicher gilt, eine sichere Anwendung auf dem Prozessor durchzuführen. Eines dieser Trustlets ist die KeyMaster Applikation, welche in der restlichen Arbeit als KeyMaster Hardwaremodul bezeichnet wurde. Dieses Trustlet ist zuständig für die Entschlüsselung des Private RSA Exponenten mit dem signiert. Der Befehl `sign_data` wurde von Gal Beniamini reverse engineered.[24] [8] Er zeigte, das der interne statische Schlüssel im KeyMaster Hardwaremodul der den Private RSA Exponenten verschlüsselt, nicht an die Hardware gebunden ist, sondern der TrustZone zur Verfügung steht. Die TrustZone ist ein weiterer Bereich innerhalb der TEE, die diese, und die unterschiedlichen Trustlets monitored. Für diese TrustZone gibt es eine Schwachstelle CVE-2016-2431 die es erlaubt, Code im TrustZoneKernel auszuführen. Daher wurde ein kleiner Shellcode von Gal Beniamini geschrieben, der den Schlüssel des KeyMaster Moduls aus der TrustZone ausliest und in die unsichere Welt des Android Betriebssystems zurückgibt. Dadurch das dieser Schlüssel extrahiert werden kann, können Offline Brute Force Attacken wieder durchgeführt werden.[24]

[7] `https://www.youtube.com/watch?v=QpCWS5dM7eY`

[8] Reverse Engineering bezeichnetden Vorgang ein bestehendes System zu analysieren und sukzessive durch Untersuchung der einzelnen Bestandteile und Strukturen das ursprüngliche System als 1-1-Kopie nachzubauen

5.3.4.2 Gegenmaßnahmen

Die Schwachstelle CVE-2015-6639 wurde im Security Bulletin von Jänner 2016 behoben. Buildnummer LMY49F oder Android Version 6.0 mit dem Security Patch Level vom Jänner 2016 sind nicht mehr davon betroffen. Schwachstelle CVE-2016-2431 wurde im Mai 2016 behoben. Durch die Einführung von FBE in Android Version 7 wird ein gänzlich anderes Verschlüsselungskonzept verwendet.

5.3.4.3 Angriffsfläche

Anfällig gegen diesen Angriff sind alle Geräte mit der Android Version 6.0 vor dem Sicherheitspatch vom Jänner 2016 und das Gerät muss als TEE einen anfälligen Qualcomm Chip verbaut haben. Es besteht die Möglichkeit, ein Smartphone auf eine anfällige Version zurückzusetzen um den Angriff doch durchzuführen. Mithilfe der Partitionsimages der Userdaten, einem Smartphone wie dem Nexus 5X, das auf Android 6.0 Stand Oktober 2015 zurückgesetzt werden kann und dem passenden Angriffsskript, kann die Userpartition entschlüsselt werden. Das Zurücksetzten auf eine anfällige Version funktioniert ab Android Version 7.0 mit aktivierten FBE nicht mehr, jedoch bei der Verwendung von FDE unter Android 7.0 ist es nach Meinung des Autors wahrscheinlich möglich. Bezüglich den Angriffsszenarien gelten die gleichen Bedingungen wie bei der Semi-Offline Attacke. Siehe Abschnitt 5.3.3.3

5.3.5 Cold-Boot Angriffe

Um kryptografische Transparenz zu erreichen muss der Verschlüsselungsschlüssel im Hauptspeicher gehalten werden. Der Hauptspeicher ist definiert als flüchtiges Speichermedium, daher wird allgemein angenommen, dass nachdem der Strom abgedreht worden ist, der Inhalt des Hauptspeichers für immer und sofort verloren ist. In der Praxis stellt sich diese Behauptung als falsch heraus wie Halderman und Kollegen in ihrer Arbeit[32] mit der folgenden Abbildung 5.5 visualisiert haben.
Zuerst wird das Bild im Originalzustand gezeigt, danach 30 Sekunden nachdem der Strom weg war, 60 Sekunden und das letzte Bild zeigt den Zustand des Bildes nach 5 Minuten. Die Dauer in denen die Daten noch rekonstruierbar bleiben, kann mit herunterkühlen des Speichermoduls erhöht werden. Das vergrößert das Zeitfenster von Angriffen auf flüchtige Speicher. In diesem Zeitfenster muss es dem Angreifer gelingen, möglichst schnell ein Abbild des Hauptspeichers zu generieren. Dabei reichen die Ansätze von PXE Netzwerkboot bis zum Booten von externen

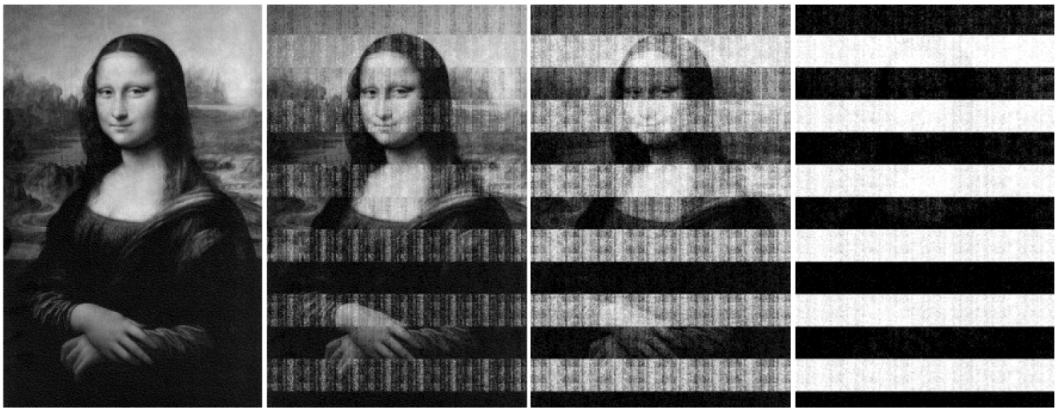

Abbildung 5.5: Visualisierung des Datenverlustes nach Ausschalten des Gerätes [32][S.50]

.

Festplatten. In ihrer Arbeit demonstrieren die Autoren einen Angriff auf AES 128 verschlüsselte Festplatten. Diese wurden mit unterschiedlichen Softwarelösungen verschlüsselt, unter anderem die beschriebene dm-crypt von Linux die auch in Android verwendet wird. Mit ihrem Programm `keyfind` waren sie in der Lage den 128 Bit AES Schlüssel aus dem Hauptspeicherabbild zu extrahieren.[32][S.55]

5.3.5.1 Angriffsanalyse

In ihrer Arbeit[33] zeigen T. Mueller, M. Spreitzenbarth und F. Freiling, dass das Konzept der Cold-Boot Attacke auch für Android angewandt werden kann. Sie entwickelten ein Proof of Concept Tool für Android Version 4, dass auf einem Galaxy Nexus (GT-9250) installiert wurde. FROST (Forensic Recovery of Scrambled Telephones) ist ein Recovery Image das auf dem Gerät installiert wird, wenn physischer Zugriff besteht. Um den Angriff durchzuführen wird das Gerät auf bis zu 5 Grad Celsius runtergekühlt. Danach wird es neugestartet und das Recovery Image FROST mit `fastboot` installiert. Dazu muss der Bootloader des Gerätes entsperrt sein. Danach ist FROST in der Lage nicht nur den AES 128 Bit Schlüssel zu extrahieren, sondern alles was im DRAM vorgefunden wird. FROST kann auch nach dem entsperren des Bootloaders installiert werden und dabei den AES Schlüssel extrahieren, was aber nicht mehr sinnvoll ist, da das entsperren des Bootloaders eine Systemlöschung zur Folge hat.

5.3.5.2 Gegenmaßnahmen

Die Gegenmaßnahme zu dieser Attacke lautet das Schlüsselmaterial nicht im Hauptspeicher zu halten. ARMORED [34][S.92-96] ist ein Ansatz die Verschlüsselungsschlüssel in den CPU Registern zu halten und nicht im Hauptspeicher. Die Autoren testeten erfolgreich ihren Ansatz gegen FROST, daher war es nicht möglich mit FROST den AES Schlüssel auszulesen.[34][S.92-96] Die Limitierung von ARMORED liegt aber klar an der Größe der CPU Register. ARMORED kann nur 128 Bit große Schlüssel aufnehmen, was bedeutet das die Schlüssel, die für FBE verwendet werden, nicht geschützt werden können.

5.3.5.3 Angriffsfläche

Ein Angreifer benötigt einen entsperrten Bootloader, um den Angriff durchzuführen und der Verschlüsselungsschlüssel DEK muss im Hauptspeicher vorhanden sein. Android Versionen die FDE verwenden, dürften nach Meinung des Autors immer noch anfällig auf diese Attacke sein. Durch die Einführung von FBE wurde im CDD definiert, dass die Schlüssel für CE und DE kryptografisch an die TEE gebunden werden müssen.2.2.1 Daher geht der Autor dieser Arbeit davon aus, dass wenn Schlüssel durch FROST auf Geräten mit FBE gefunden werden können, diese nur einzelne spezifische Dateischlüssel sind. Nur eines der beiden Angriffsszenarien wird von der ColdBoot-Attacke abgedeckt, da das Schlüsselmaterial vorher im Hauptspeicher sein muss.

5.3.6 Statisches Default Passwort und Master Key

Die Implementierung seit Android Version 5.0 verwendet ein bekanntes Standardpasswort und einem statischen, aber geheimen Master Key. Oliver Kunz hat in seiner Arbeit für Android Version 5.0 in diesen Bereich potentielle Bedrohungen identifiziert.[23][S.36-38] Zuerst wird jeder einzelne Aspekt individuell betrachtet und dann die potentiellen Bedrohungen anhand ihrer Kombinationen dargestellt.

5.3.6.1 Statisches Default Passwort

Die KEK Generation, die in den Abschnitten 2.4.2 und 2.5.2 diskutiert werden, erfordert ein Passwort. Dabei wird auch erläutert, dass in drei Situationen das im Source Code definierte Standard Passwort `DEFAULT_PASSWORD` verwendet wird.

- Verschlüsselung beim ersten hochfahren
- Bildschirmsperrmethode Wischen oder wenn Keine verwendet wird
- Passwort während des Hochfahrens des Gerätes nicht abfragen

In den ersten beiden Situationen, besteht durch die Geräteverschlüsselung kein Schutz. Jeder der physischen Zugriff zum Smartphone hat, besitzt die volle Kontrolle. Es ist weder beim Hochfahren, noch beim Entsperren des Bildschirms oder im Fall dass die Bildschirmsperre aktiviert ist, ein Passwort erforderlich. Das bedeutet, dass ein Angreifer, in unseren zwei definierten Angriffsszenarios, mithilfe eines Skriptes welches eine Abänderung der Semi-Offline Brute Force Attacke von Oliver Kunz ist [23][S.33-36], in der Lage ist den entschlüsselten Master Key zu bekommen, da er alle benötigten Teile besitzt.

5.3.6.2 Statischer Master Key

Es wurden bisher keine Methoden bezüglich einer Änderung des Master Keys aufgezeigt, da diese funktionell nicht implementiert sind. Der Master Key wird dabei einmal eingelesen von `/dev/urandom` und bleibt dann statisch auf dem Gerät im Crypto Footer, bis das Smartphone gelöscht wird. Änderungen eines Schlüssels sind ein wichtiger Teil des Lebenszyklus eines Schlüssels. Benutzern wird daher oft empfohlen ihre Passwörter und Keys regelmäßig aus Sicherheitsgründen zu ändern. Unternehmen definieren in den meisten Fällen eine Lebenszeit für das Schlüsselmaterial ihrer Mitarbeiter und Server in ihren Sicherheitsrichtlinien. Die Abwesenheit einer solcher Hauptfunktionalität im Lebenszyklus des Master Keys auf Android Geräten schränkt die Möglichkeiten deren Benutzer ein. Wurde der Master Key eines Gerätes kompromittiert, bleibt nur das Sichern der Dateien übrig und eine Systemlöschung muss durchgeführt werden, um wieder neues Schlüsselmaterial aus `/dev/urandom` zu lesen. Danach müssen die gesicherten Daten wieder auf das Gerät gespielt werden. Dieselbe Prozedur müsste eigentlich bei einer strikten Unternehmensrichtlinie genauso durchgeführt werden, es sei denn, es wurde in der Richtlinie eine Ausnahme für Smartphones beschrieben. Bei einer Länge von 128 Bit für den Master Key, ist es unwahrscheinlich, dass dieser durch Kryptoanalyse wiederhergestellt werden kann, nachdem er einmal gelöscht wurde.

5.3.6.3 Angriffsanalyse

Derweil wurde jeder Aspekt individuell betrachtet und dabei wurden schon potentielle Bedrohungen festgestellt. Durch Kombination der einzelnen Aspekte kann ein noch bedrohlicheres Szenario erstellt werden. Wenn ein neues Gerät gekauft und danach in Betrieb genommen wird, besitzt es nach dem initialen Boot als Passwort das Standardpasswort. In diesem fiktiven Beispiel gehen wir von dem wahrscheinlichsten Fall aus, dass ein Gerät gekauft wird, dass nicht in der Lage ist FBE zu benutzen. Der Benutzer setzt derweil noch keinen Sperrbildschirm ein. Ein Angreifer, der in dieser Phase, physischen Zugang zum Gerät bekommt, kann die beiden benötigten Partitionen `userdata` und `metadata`(Crypto Footer) imagen, das abgeänderte Semi-Offline Brute Force Skript verwenden und somit den Master Key entschlüsseln.

Falls danach der Benutzer ein Passwort konfiguriert, ist das für den Angreifer egal. Mit den bisher getroffenen Vorkehrungen benötigt der Angreifer bei einem erneuten Angriff nur die `userdata` Partition noch einmal. Mit dem zuvor erlangten Crypto Footer und dem entschlüsselten Master Key, kann der Angreifer die zweite, diesmal mit mehr Daten als direkt nach dem initialen Boot versehene, `userdata` Partition entschlüsseln und mounten.

5.3.6.4 Gegenmaßnahmen

Um das Maß der Bedrohung für die beschriebenen Aspekte zu minimieren, müssten grundlegende Änderungen am System durchgeführt werden. Der vermutliche Grund hinter dem Standardpasswort, besteht aus 2 Argumenten. Einerseits wird das Gerät beim initialen Starten verschlüsselt ohne das der Benutzer etwas tun muss und andererseits ermöglicht es die Optionen Ẅischenünd K̈einebeim Sperrbildschirm. Eine Lösung wäre den Benutzer zu einer Eingabe eines Passwortes zu zwingen, dass das Standardpasswort `DEFAULT_PASSWORD` ersetzt bevor das Smartphone dass erste Mal startet. Dies würde aber für den Angriff mit dem Semi-Offline Brute Force Skript keine Änderung bedeuten.[23][S.38] Aus Sicht der Sicherheit wäre eine bessere Lösung den Benutzer den Sperrbildschirm mit einem Passwort einzurichten bevor der Master Key generiert wird. Diese Lösung wird wahrscheinlich wegen der immensen Änderungsarbeit am Quellcode von Google nicht umgesetzt. Weiters wäre der Benutzer gezwungen sich ein sehr wichtiges aber eher selten gebrauchtes Passwort zu merken, was nicht jeder will.[23][S.38] Bezüglich des statischen Master Key, würde eine Schlüsseländerungsfunktion eine geeignete Maßnahme zur Verbesserung der Sicherheit ausreichen. Das Problem an diesen Konzept ist, dass für eine Änderung des Master Keys in dm-crypt ein neues virtuelles Device erstellt werden

muss, alle Daten eingelesen und entschlüsselt werden müssen und auf ein neues Gerät geschrieben werden müssen und dadurch mit dem neuen Master Key verschlüsselt werden. Daher stellt das eine hohe Prozessor und doppelte Speicherbelastung während der Schlüsseländerung dar, die nicht jedes Smartphone erfüllen kann.

5.3.6.5 Angriffsfläche

Prinzipiell sind alle Versionen ab Android 5.1 von diesen Umständen betroffen, da ab dieser Version das Standardpasswort existiert. Es sind auch beide Angriffsszenarien von diesen Umständen betroffen.

5.3.7 Evil-Maid Angriff

Das Konzept der Evil-Maid Attacke beruht auf dem namensgebenden bösen Zimmermädchen, welches die verschlüsselten Geräte der Hotelbesucher angreift. Während der Abwesenheit der Gäste gelangt ein Angreifer in den Raum und bekommt dadurch physischen Zugriff auf die verschlüsselten Geräte. Bei einem Gerät mit Geräteverschlüsselung muss dennoch ein kleiner Teil unverschlüsselt bleiben, damit das System booten kann. Bei einem Notebook wäre des seine Boot Partition, bei einem Android Smartphone alle anderen Partitionen außer der verschlüsselten Userdaten Partition. Durch den physischen Zugriff auf die nicht verschlüsselten Datenbereiche der Geräte ist es dem Angreifer möglich Keylogger oder andere Malware zu installieren, bevor er den Raum verlässt. Der Besitzer welcher anschließend später sein Gerät mit dem Passwort entschlüsselt, weiß nichts von dem Keylogger, der jetzt seine Tastenanschläge auf den unverschlüsselten Datenbereich mitschreibt, oder per Netzwerk an den Angreifer versendet. Falls das böse Zimmermädchen danach das wieder leere Zimmer betritt, kann sie das Passwort vom unverschlüsselten Teil der Festplatte erhalten. [42]

5.3.7.1 Angriffsanalyse

Bei einem Android Smartphone unterscheidet sich die beschriebene Attacke in ein paar Punkten. Es ist um einiges komplexer und aufwändiger ein vorbereitetes Custom Image auf das mobile Gerät zu spielen als das Installieren von Malware auf einem Notebook von einem USB Stick aus. Der Angriff benötigt einen entsperrten Bootloader und eine angepasste Systempartition welche die Keylogging Funktionalität beinhaltet. In der Arbeit von Johannes Götzfried und Tilo Mueller[34] wird ein solches EvilDroid, mit den geforderten Eigenschaften beschrieben.

Die Autoren beschreiben außerdem einen weiteren Angriff, der für einen gesperrten Bootloader gilt. Bei diesen Angriff werden vorher genug Informationen über das Opfer gesammelt, um ein identes Smartphone im Raum des Opfers zu platzieren und es mit dem Originalgerät auszutauschen. Auf diesem Duplikat läuft EvilDroid, welches das Passwort mitloggt und an den Angreifer sendet der im Besitz des Originals ist. EvilDroid simuliert dabei in einer Applikation den Boot Vorgang, um als eine valide Android Version zu erscheinen. Gibt das Opfer das Passwort auf dem Duplikat ein ist der Angriff geglückt. Die Evil-Maid Attacken besitzen alle einen Kernfaktor, nämlich dass das Opfer sein Gerät in einem Raum lässt auf den der Angreifer Zugriff hat. Dass mag bei Notebooks und Desktop PC der Fall sein, bei mobilen Geräten wie einem Smartphone wird die Wahrscheinlichkeit geringer, dennoch gibt es Szenarien wo dies durchaus eintrifft.

5.3.7.2 Gegenmaßnahmen

Die erste Version des Evil-Maid Angriffs kann erfolgreich vom gesperrten Bootloader abgewehrt werden. Der zweite Ansatz funktioniert dadurch, dass der Eingabebildschirm für das Passwort beim Booten jedes Mal gleich aussieht, daher ist es schwer festzustellen, ob gerade wirklich Android oder EvilDroid gebootet wird, sofern die richtige Version des Betriebssystems verwendet wird. Das Feature des Verified Boot kann dadurch umgangen werden, da bei Verletzung von Verified Boot nur eine Warnung beim Starten des Gerätes aufscheint, aber fortgefahren werden kann. Wenn das Opfer diese Meldung nicht sieht, kann es keine Unterscheidung treffen.

5.3.7.3 Angriffsfläche

Evil-Maid Angriffe betreffen alle Versionen von Android. Für die zwei definierten Angriffsszenarien sind beide Arten der beschriebenen Varianten von Evil-Maid möglich. Das Deaktivieren von adb hat keinen Einfluss auf die Evil-Maid Attacke, jedoch des Sperren des Bootloaders.

6 Sicherheitsüberprüfung

In diesen Kapitel soll es um die konkrete Sicherheitsüberprüfung und damit das Erstellen eines Proof of Concept für einen konkreten Angriff auf die Geräteverschlüsselung einer relativ neuen Android Version gehen. Dazu wird zuerst die Methodik des Vorhabens beschrieben, das Setup der verwendeten Geräte, wie diese präpariert wurden und welche Varianten es zum Erstellen eines Partitionsimages gibt und danach das Erstellen des eigentlichen Proof of Concepts für einen Angriff auf die Geräteverschlüsselung.

6.1 Methodik

Das Testen des Proof of Concepts wurde auf zwei LG Nexus 5X Codename `bullhead` durchgeführt. Die Gerätereihe von Nexus gilt dem reinen Android am nähesten. Auf diesen Geräten läuft deswegen Plain Android, welches sehr wenig Quellcode von Drittherstellern besitzt. Falls Schwachstellen auf solch einem Gerät gefunden werden, kann davon ausgegangen werden, dass mehrere Hersteller betroffen sind, da diese den Plain Android Quellcode für ihre Geräte umsetzen. Die beiden Geräte wurden einerseits auf Android 7.1.1 r11 Build Nummer N4F26I Jänner 2017 und Android 7.1.2 r18 N2G47Z Juli 2017 gebracht um relativ aktuelle Versionsnummern von Android zu verwenden. Auf beiden Geräten wurde die FBE explizit in den Entwickleroptionen des Gerätes aktiviert. Auf den Geräten läuft der Linux Kernel 3.10.73. Auf beiden Geräten wird der Bootloader entsperrt. Zusätzlich werden die Geräte gerootet und adb aktiviert. Näheres unter Abschnitt 6.2. Als Host PC wird eine Kali Linux 2017.1 VM verwendet. In dieser VM wird der Quellcode aus dem Google Repository geladen und der Proof of Concept entwickelt. Zusätzlich werden die Aufgaben wie das Entsperren des Bootloaders, rooten der Geräte, das allgemeine Arbeiten mit den Geräten via adb, das zurücksetzen und flashen von Android Abbilder und das experimentieren mit den Geräten über diese VM erledigt.

6.2 Setup der Android Smartphones

6.2.1 Entsperren des Bootloaders

Das Entsperren des Bootloaders wurde in Abschnitt 5.2.1 schon beschrieben. Zwecks der Vollständigkeit hier noch einmal die Kurzfassung. Das Gerät muss in den Bootloader gebracht werden. Dies geschieht durch das Drücken der bestimmten Tastenkombination beim Starten oder das Gerät wird über den Befehl `adb reboot bootloader` dazu gebracht. Danach muss der Befehl `fastboot oem unlock` ausgeführt werden und die Systemlöschung am Gerät selber bestätigt werden. Danach ist der Bootloader entsperrt und das Gerät somit bereit gerootet zu werden.

6.2.2 Rooten des Gerätes

Um das Gerät zu rooten muss vorher der Bootloader entsperrt werden. Danach wird ein Custom Flash Image benötigt. Es wurde das Image `twrp-3.1.1-0-bullhead.img` von TeamWin [1] verwendet. Um Root Rechte auf dem Gerät zu erlangen benötigt man noch die App SuperSU [2]. Verwendet wurde die letzte Version 2.82 von SuperSU. Um den Rootvorgang auf dem Gerät durchzuführen muss die heruntergeladene SuperSU.zip Datei auf das Smartphone übertragen werden. Mit dem Befehl `adb push SuperSU.zip /sdcard/Download/` bringen wir die Zip Datei auf das Gerät.

Danach wird der Befehl `adb reboot bootloader` ausgeführt um wieder im Bootloader zu landen. Danach wird mit dem Tool fastboot das Image von TeamWin auf das Gerät mit dem Befehl `fastboot flash recovery twrp-3.1.1-0-bullhead.img` geflashed. Wurde der Befehl erfolgreich ausgeführt, wählt man auf dem Gerät durch Drücken der Volume Up Taste im Bootloader den Recovery Mode aus. Somit startet sich das Recovery Image von TeamWin. Mit diesen Image kann jetzt die SuperSU.zip auf der SD Karte ausgewählt und installiert werden. Danach wird das Smartphone normal gestartet und die App SuperSU so konfiguriert, dass alle Anfragen auf Root Zugriff zugelassen werden, damit nicht jedes mal wenn in der adb Shell der Befehl `su` aufgerufen wird, der Root Zugriff am Bildschirm des Gerätes bestätigt werden muss.

[1] `https://eu.dl.twrp.me/bullhead/`
[2] `http://www.supersu.com/download`

6.2.3 Android Quellcode Repository

Der Download vom Android Quellcode wird durch das Programm repo übernommen. [3] Nachdem repo installiert wurde, wird ein Ordner erstellt, in den dann der Quellcode heruntergeladen wird. Durch den Parameter `-b <Bezeichnung-des-Branches>` kann der Branch angegeben werden um den Android Quellcode einzugrenzen. Die Bezeichnung des Branches passt auf eine bestimmte Build Nummer einer Android Version für ein Google Gerät. Auf der Seite `https://source.android.com/source/build-numbers` kann nachgesehen werden, wie die Bezeichnung des genauen Branches für das Gerät lautet. Für die beiden Geräte waren das Branch `android-7.1.1_r10` und Branch `android-7.1.2_r18`.

Diese beiden Repositories wurden jeweils mit dem Kommandozeilenbefehl `repo init -u https://android.googlesource.com/platform/manifest -b <BdB>` in ihren Ordnern initialisiert und mit dem Befehl `repo sync` synchronisiert und dadurch heruntergeladen. Danach können die Quelldateien in den Repos analysiert und verändert werden. Um Teile des Quellcodes zu kompilieren muss zuerst im Verzeichnis des Repositories mit dem Befehl `source build/envsetup.sh` das envsetup ausgeführt werden um Zugriff auf viele Helferfunktionen zu bekommen. Diese Funktionen helfen mit dem Repository zu arbeiten. Einer dieser Funtionen lautet `choosecombo`, welche hilft die genaue Version der Android Version zu bestimmen, die kompiliert werden soll. Mit dem Befehl `echo -e "\naosp_bullhead\n\n"|choosecombo` wird die Release Version und das Zielprodukt `aosp_bullhead` bestimmt. Mit dem Befehl `mmma <Name-der-Komponente` kann gezielt eine Komponente von diesem Branch kompiliert werden. Für die Binary des Fingerprint Daemons ist das der Befehl `mmma` system/core/fingerprintd/. Das kompilierte Ergebnis wird im lokalen Repository im Ordern `out` des jeweiligen Repositories gespeichert.

6.3 Partition Imaging

Für die Erstellung des Proof of Concepts, der sich mit der Umgehung des Fingerabdrucksensors beschäftigt wurde das Erstellen von Abbildern der einzelnen Partitionen nicht benötigt, dennoch wird es nachfolgend aus zwei Gründen beschrieben. Einerseits, aufgrund der Vollständigkeit und andererseits, um die in Kapitel 5 gezeigten Angriffe besser nach vollziehen zu können. Die nachfolgenden Techniken wurden in der Recherchephase der Erstellung dieser

[3] `https://source.android.com/source/downloading`

Arbeit verwendet, um wie erwähnt die Angriffe nachvollziehen zu können.

Die Partitionen von Android Geräten zu imagen, daher ein Abbild davon erstellen, kann unterschiedlich erreicht werden, dennoch existieren Limitierungen:

- Das Image der Benutzerdaten kann nicht direkt am Smartphone selber erstellt werden und dort gespeichert werden, da das ja nur auf der `userdata` Partition möglich ist.
- adb ist nicht am Zielgerät aktiviert oder ein Sperrbildschirm schützt dadurch die Aktivierung von adb
- Das Zielgerät besitzt einen gesperrten Bootloader
- Das Entsperren vom Bootloader löst eine Löschung der Daten aus

6.3.1 Imaging ohne Sperrbildschirm

Wenn das Zielgerät keinen Sperrbildschirm besitzt, kann ein Angreifer den Zugriff via adb selber auf dem Gerät aktivieren wie in Abschnitt 5.2.2 beschrieben. Sollte adb bereits aktiviert sein, kann der Angreifer dadurch am Zielgerät die aufscheinende Host Authentifizierung bestätigen. Besitzt der Angreifer ein Gerät, dass zwar einerseits abd aktiviert hat, aber die Host Authentifizierung aufgrund eines Sperrbildschirmes nicht aktivieren kann, muss er auf einen Host PC ausweichen der diese besitzt.

6.3.1.1 Host PC

1. Zuerst muss eine Kommandozeile geöffnet werden und damit adb gestartet werden. Danach mit dem Befehl `adb devices` fortfahren. Dieser zeigt eine Liste aller mit dem Host verbundenen und erkannten Android Geräte an. Dieser Befehl löst auch die Host Authentifizierung am Smartphone aus, welche bestätigt werden muss.
2. Starten eines Portlisteners am Smartphone auf einen Port der verwendet werden soll und konfiguriere eine Reverse Forwarding Verbindung auf einen Port mit dem Host PC. `adb reverse tcp:DEVICE_PORT tcp:HOST_PORT`
3. Starten eines Netzwerk-Listeners netcat der auf den Host Port horcht und dabei einen Zielordner für das Partitionsimage bestimmt.
   `nc -l HOST_PORT | dd of=<path-to-destination>`

4. Eine weiter Kommandozeile wird geöffnet und in dieser mit `adb shell` eine Kommandozeile auf dem Smartphone geöffnet.

   a) Zuerst kontrollieren ob das Forwarding geht.
   `netstat -an | grep -iw DEVICE_PORT`

   b) Danach die userdata und metadata Partitionen auf dem Smartphone lokalisieren mithilfe von `ls` und `find`

   c) Das Partitionieren starten und gleichzeitig über netcat zum Host PC schicken.
   `dd if=<path-to-partition> | nc 127.0.0.1 DEVICE_PORT`

Für die weitere Partitionen einfach die Schritte ab Punkt 3 wiederholen.

6.3.1.2 Smartphone

Die Entwickleroptionen und adb müssen aktiviert sein und das Gerät via USB mit dem Host PC verbunden.

1. Auf dem Android Gerät muss die Host Authentifizierung mit OK bestätigt werden, wenn diese ausgelöst wird.

6.3.2 Imaging mit Sperrbildschirm

Die Vorgehensweise gestaltet sich ähnlich wie schon beschrieben. Der Unterschied liegt im konfigurierten Sperrbildschirm am Android Gerät der das Bestätigen der adb-Host Verbindung verhindert. Falls der Bootloader jedoch nicht gesperrt ist, kann ein Custom Recovery Image geladen werden, um damit die Arbeit durchzuführen.

6.3.2.1 Smartphone

Das Gerät muss via USB mit dem Host PC verbunden sein.

1. Reboote das Android Gerät in den Bootloader.

2. Wenn der Bootloader Bildschirm erscheint wechsle zurück auf den Host.

6.3.2.2 Host PC

Der Angreifer kann ein Custom Recovery Image verwenden oder selber erstellen. Meist wird das Image von TeamWin verwendet. Wie in Abschnitt 5.2.1 beschrieben gibt es hier keine Gegenmaßnahmen um dies zu verhindern falls der Bootloader entsperrt ist.

1. Öffne eine Kommandozeile und boote das Smartphone mit dem Custom Recovery Image durch den Befehl
   `fastboot boot <path-to-recovery.img>`
2. Danach kann mit der Kommandozeile adb gestartet werden. Danach mit dem Befehl `adb devices` fortfahren. Dieser zeigt eine Liste aller mit dem Host verbundenen und erkannten Android Geräte an. Dieser Befehl löst auch die Host Authentifizierung am Smartphone aus, welche bestätigt werden muss.
3. Starten eines Portlisteners am Smartphone auf einen Port der verwendet werden soll und konfiguriere eine Reverse Forwarding Verbindung auf einen Port mit dem Host PC.
   `adb reverse tcp:DEVICE_PORT tcp:HOST_PORT`
4. Starten eines Netzwerk-Listeners netcat der auf den Host Port horcht und dabei einen Zielordner für das Partitionsimage bestimmt.
   `nc -l HOST_PORT | dd of=<path-to-destination>`
5. Eine weiter Kommandozeile wird geöffnet und in dieser mit `adb shell` eine Kommandozeile auf dem Smartphone geöffnet.
   a) Zuerst kontrollieren ob das Forwarding geht.
      `netstat -an | grep -iw DEVICE_PORT`
   b) Danach die userdata und metadata Partitionen auf dem Smartphone lokalisieren mithilfe von `ls` und `find`
   c) Das Partitionieren starten und gleichzeitig über netcat zum Host PC schicken.
      `dd if=<path-to-partition> | nc 127.0.0.1 DEVICE_PORT`

Für die weitere Partitionen einfach die Schritte ab Punkt 3 wiederholen.

6.3.3 Imaging via Hardware-Level

Falls diese beide generischen Gegenmaßnahmen Abschnitt 5.2.1 und 5.2.2 aktiv sind, kann nur durch eine äußert komplexe Vorgehensweise die Partitionen geimaged werden. Dabei kann ver-

sucht werden den Inhalt des Flash Memory ohne Hilfe des Betriebssystems auszulesen. Wie schon erwähnt kann das durch Zuhilfenahme des JTAG Interfaces erreicht werden. Diese Vorgehensweise setzt immenses Vorwissen in den Gebieten Elektronik und Chipsatzbau voraus und steht damit in keinem Vergleich zu den bisher gezeigten Methoden. Wenn dieser Ansatz immer noch nicht zielführend war, besteht die letzte Möglichkeit darin, den Chip aus dem Gerät herauszulöten und danach mit spezieller Hardware wie Ming the Merciless auszulesen. [4]

6.4 Umgehen des Fingerabdrucksensors

Der Fokus der Sicherheitsüberprüfung dieser Arbeit galt dem Erstellen eines Proof-of-Concept für einen Angriff auf die Authentifizierung mit dem Fingerabdrucksensor, um die Geräteverschlüsselung von Android Version 7.1.1 und Android Version 7.1.2 zu umgehen. Thom Does und Mike Maarse beschreiben in ihren Paper einen Weg für Android Version 6.0 um den Fingerabdrucksensor zu umgehen.[35]. Dies bildet die Basis für meinen Proof-of-Concept um zu prüfen, ob die neu eingeführte Geräteverschlüsslung FBE unter Android Version 7.0 anfällig auf diesen Angriff ist.

6.4.1 Software Komponenten

Das Fingerabdruck Authentifizierungssystem besteht aus mehreren Komponenten, die von unterschiedlichen Entwicklern kommen. Abbildung 6.1 stellt alle Komponenten und deren Verhältnisse visuell dar. Die rote Komponente repräsentiert die Third Party Entwickler und deren geschriebene Apps für das Smartphone. Die grüne Komponente ist die Fingerprint Hardware Abstraction Layer, welche als Interface zur Hardware fungiert. Diese Komponente ist herstellerabhängig und daher bei unterschiedlichen Geräteherstellern unterschiedlich implementiert, da sie für die direkte Kommunikation mit dem Sensor zuständig ist. Die gelben Komponenten stellen die Teile des AOSP Quellcodes dar, welche Hardware- und Herstellerunabhängig programmiert wurden.
Der Fingerprint Daemon `fingerprintd` ist die Komponente, welche für meinen Proof-of-Concept am relevantesten ist. Dieser läuft als eigener Prozess am Android Gerät und kommuniziert mit der `Fingerprint HAL`. Somit ist er die Betriebssystem Komponente, welche am nähesten zur Hardware ist. Dieser Dienst macht Aufrufe durch die `Fingerprint HAL`, um

[4] `http://blog.mpecsinc.ca/2011/02/encryption-via-bitlocker-or-other-with.html`

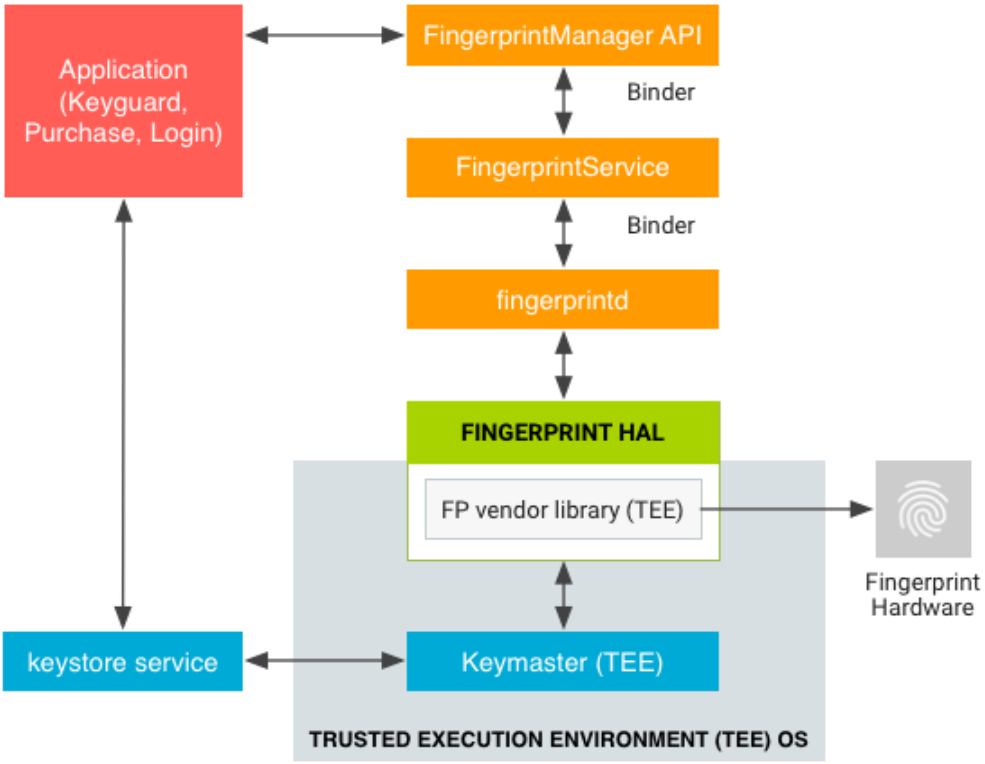

Abbildung 6.1: Fingerabdruck Authentifizierungssystem [43]

.

Fingerabdrücke zu enrollen[5], zu löschen und um sie zu authentifizieren. Die enrollten Fingerabdrücke werden als Sicherheitsmaßnahme in der TEE gespeichert.

6.4.2 Angriffsanalyse

Jedesmal wenn der Fingerabdrucksensor des Gerätes einen Fingerabdruck erkennt, sendet das TEE die dazu berechnete ID zum Fingerprint Daemon. Jeder Fingerabdruck, der am Gerät enrolled ist, besitzt in der TEE eine dazugehörende einzigartige Fingerprint ID. Diese IDs sind zufällig generiert. Fingerabdrücke können jede ID erhalten, außer der speziellen ID 0. Die ID 0 ist von der Hardware reserviert um mitzuteilen, dass der Fingerabdruck vom Sensor nicht gelesen werden konnte. Daher kann vermutet werden, dass jeder Wert der nicht 0 ist, als erfolgreicher Authentifizierungsversuch gewertet wird. Es wurde von den Autoren Thom Does und Mike Maarse in der Komponente `FingerprintService` eine Überprüfung auf die Fingerprint ID gefunden, welche von `fingerprintd` gesendet werden.[35][S.8] Lautet die gesendete ID 0, entscheidet die Überprüfung das der Fingerabdruck nicht eingelesen werden konnte und die

[5]Enrollen bezeichnet den Vorgang, ein biometrisches Merkmal in einem Computersystem zu speichern

Authentifizierung schlägt fehl. Falls die ID etwas anderes als 0 ist, entscheidet die Überprüfung dass der Fingerabdruck erkannt worden ist und die Authentifizierung gelingt. Diese Entscheidung wird an die Instanz von `FingerprintManager` weitergeleitet, die den Authentifizierungsprozess initiiert hat. Die Anwendung, welche `FingerprintManager` implementiert hat, wird nun etwas ausführen das vom Authentifizierungsresultat abhängig war. Zum Beispiel entsperrt sich der Sperrbildschirm bei einem positiven Resultat oder zeigt eine Fehlermeldung bei einem negativen an. Mit diesem Wissen sieht der Proof-of-Concept wie folgt aus: Die Überprüfung in `FingerprintService` kann ausgenutzt werden indem von `fingerprintd` immer eine positive ID gesendet wird. Dadurch entscheidet `FingerprintService` das der Versuch gültig war und leitet das positive Ergebnis an die Zielapplikation weiter. Der Quellcode von `fingerprintd` muss an der richtigen Stelle geändert werden und die modifizierte Binärdatei von `fingerprintd` an die richtige Stelle auf dem Smartphone kopiert werden, um den Sperrbildschirm zu umgehen.

FingerprintDaemonProxy.cpp + (~/Desktop/Android_Nexus5/system/core/fingerprintd) - VIM

File Edit Tools Syntax Buffers Window Help

```
    closeHal();
}

void FingerprintDaemonProxy::hal_notify_callback(const fingerprint_msg_t *msg) {
    FingerprintDaemonProxy* instance = FingerprintDaemonProxy::getInstance();
    const sp<IFingerprintDaemonCallback> callback = instance->mCallback;
    if (callback == NULL) {
        ALOGE("Invalid callback object");
        return;
    }
    const int64_t device = (int64_t) instance->mDevice;
    switch (msg->type) {
        case FINGERPRINT_ERROR:
            ALOGD("onError(%d)", msg->data.error);
            callback->onError(device, msg->data.error);
            break;
        case FINGERPRINT_ACQUIRED:
            ALOGD("onAcquired(%d)", msg->data.acquired.acquired_info);
            callback->onAcquired(device, msg->data.acquired.acquired_info);
            break;
        case FINGERPRINT_AUTHENTICATED:
            ALOGD("onAuthenticated(fid=%d, gid=%d)",
                    msg->data.authenticated.finger.fid,
                    msg->data.authenticated.finger.gid);
            if (msg->data.authenticated.finger.fid != 0) {
                const uint8_t* hat = reinterpret_cast<const uint8_t *>(&msg->data.authenticated.hat);
                instance->notifyKeystore(hat, sizeof(msg->data.authenticated.hat));
            }
            callback->onAuthenticated(device,
                    408, //msg->data.authenticated.finger.fid,
                    msg->data.authenticated.finger.gid);
            break;
        case FINGERPRINT_TEMPLATE_ENROLLING:
            ALOGD("onEnrollResult(fid=%d, gid=%d, rem=%d)",
                    msg->data.enroll.finger.fid,
                    msg->data.enroll.finger.gid,
                    msg->data.enroll.samples_remaining);
            callback->onEnrollResult(device,
-- INSERT --                                                    70,63         17%
```

Abbildung 6.2: Quellcodeänderung in der Datei FingerprintDaemonProxy.cpp [44]

Wie in Abbildung 6.2 zu erkennen ist, wurde die Änderung im Quellcode in der Funktion `hal_notify_callback` Zeile 70 durchgeführt. Nachdem die modifizierte Binärdatei erstellt worden ist, siehe Abschnitt 6.2.3 für genauere Informationen, musste diese Datei nach `/system/bin/fingerprintd` kopiert und gestartet werden, um die Schwachstelle ausnutzen zu können. Das Kopieren der modifzierten Binärdatei erfolgte mit einem Zwischenschritt auf die SD Karte und nachdem die `/system` Partition schreib und lesbar gemountet wurde, kann die Datei in den Zielordner kopiert werden.

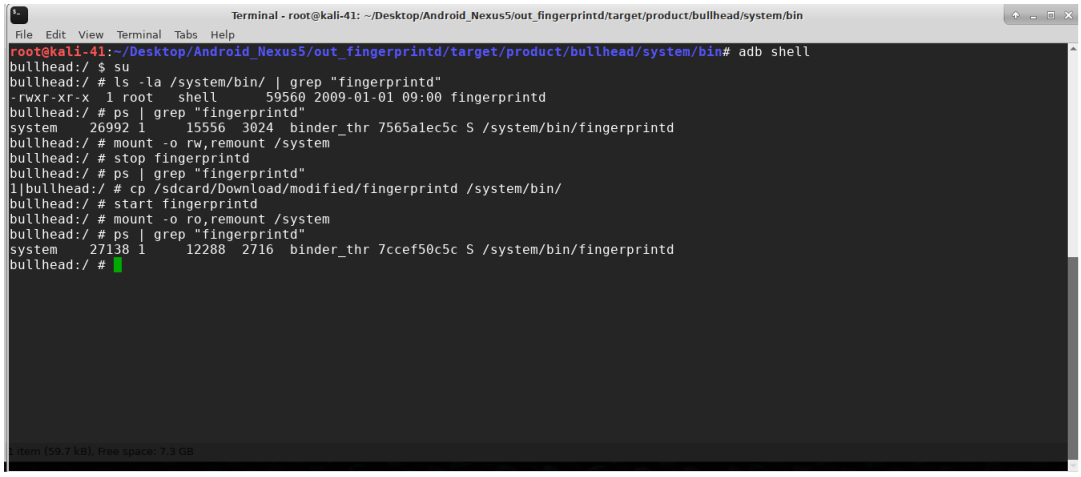

Abbildung 6.3: Kopieren der Binärdatei und Überprüfen des Prozesses

Auf der Abbildung 6.3 wird das Kopieren von der SD Karte in den Zielordner gezeigt. Zusätzlich musste der ursprüngliche Prozess beendet werden, um die Datei ersetzend zu kopieren. Daher kann man die zusätzlichen Überprüfungen mit dem Befehl `ps` sehen. Nachdem die modifizierte Binärdatei als Prozess lief, konnte sich jeder auf den Testgeräten mit seinem Finger anmelden, obwohl dieser Finger auf dem Gerät nicht enrolled war, da immer von der modifizierten Binärdatei die Fingerprint ID 408 zurückgegeben wird.

6.4.3 Gegenmaßnahmen

Als Gegenmaßnahmen schlägt der Autor dieser Arbeit vor, systemnahe kritische Binaries oder die `/system` Partition zusätzlich zu verschlüsseln. Das würde den unverschlüsselten potentiellen Angriffsbereich verkleinern. Wenn die `/system` Partition in der gleichen Weise verschlüsselt wird, wie der Device Enrypted Bereich auf der `/data` Partition, wären dessen Dateien und somit die kritischen Systemkomponenten im ruhenden Zustand geschützt. Das Verschlüsseln würde zwar einen Angriff nicht verhindern, aber die Ausführung eines Angriffes zumindest

erschweren.

6.4.4 Angriffsfläche

Die Experimente wurde von den Autoren und mir nur auf einem Nexus 5X bei unterschiedlichen Android Versionen (6.0, 7.1.1 und 7.1.2) durchgeführt, daher kann nicht auf andere Hardwarehersteller und Fingerabdrucksensoren geschlossen werden. Dennoch kann angenommen werden, dass mehrere Smartphonehersteller durch diese Schwachstelle betroffen sind. Um die Quellcodedateien auf dem Gerät zu ändern wird root Zugang benötigt (entsperrter Bootloader) und adb muss aktiviert sein. Um die Fingerabdruckauthentifizierung zu verwenden muss immer zusätzlich ein Passwort (PIN, Passwort oder Muster) durch den Benutzer gesetzt werden, welches beim Booten des Gerätes eingegeben werden muss. Daher eignet sich der Angriff nicht bei einem ausgeschalteten Gerät. Der Angriff wurde auf beiden Arten der Geräteverschlüsselung FDE und FBE durchgeführt, dabei wurde ein Gerät nach dem erfolgreichen Angriff auf File Based Encryption zurückgesetzt und die FBE nicht in den Entwickleroptionen aktiviert, damit das Gerät FDE verwendet. Bei beiden Arten konnte der Angriff erfolgreich durchgeführt werden. Auf Android Version 7.1.2 konnte der Angriff nicht zu 100 Prozent durchgeführt werden. Der Angriff konnte zwar bestätigt werden, allerdings ließ sich der Fingerprint Daemon nicht im laufenden Betrieb des Gerätes wieder starten. Erst nach einem Reboot des Gerätes, wo entgegen der Anforderung, das Passwort eingeben werden muss, wurde die modifizierte Binärdatei vom Betriebssystem ausgeführt und es konnte sich wieder jeder mit seinem Finger anmelden. Diesey Verhalten wurde durch das Update auf Android Version 7.1.2 verursacht.[6] Nach diesen Update kam es bei unzähligen Benutzern zu Abstürzen des Fingerabdrucksensors, der sich danach nicht mehr selbst startete. Das gleiche Problem trat auch bei der Entwicklung des Proof-of-Concepts auf. Nachdem der Prozess gestoppt, wurde um die Binärdatei zu kopieren, konnte sie danach nicht mehr zum Laufen gebracht werden, weder die modifizierte Binärdatei noch die originale Binärdatei. Die einzige Möglichkeit den Sensor wieder zum Laufen zu bekommen, war das Smartphone neu zu starten. Bei Android Version 7.1.1 startete sich der Prozess nach dem Kopieren der modifizierten Binärdatei von selber und der Angriff konnte im laufenden Betrieb durchgeführt werden.

[6] `https://productforums.google.com/forum/#!topic/nexus/vLbK8_TF7vE`

6.5 Übersichtstabelle der Angriffe

Eine potentiell funktionierende Attacke für ein Android Smartphone und dessen Version auszuwählen, hat sich nach der Recherche und während dem Schreiben dieser Arbeit, als sehr langwierig und zeitaufwendig dargestellt. Daher wird in diesem Abschnitt eine Angriffsübersicht, der in dieser Arbeit beschriebenen Attacken, als Tabelle zur visuellen Veranschaulichung zusammengefasst. Ein ✓in der Spalte Implementierte Gegenmaßnahmen bedeutet, diese Gegenmaßnahme wirkt erfolgreich gegen diesen Angriff. Ein ✓in der Spalte Angriffsszenarien,Android oder Geräteverschlüsselung zeigt die betroffenen Arten und Versionen der jeweiligen Spalte an. Ein **?** zeigt ein möglicherweises zutreffen an und ein **-** ein nicht zutreffen.

Attacken auf die Implementierung	Implementierte Gegenmaßnahmen			Angriffs-szenarien		Android			Geräte verschlüsselung	
	Gesperrter Bootloader	adb deaktiviert	TEE	Ausgeschalten	Eingeschalten	5.0	6.0	7.0	FDE	FBE
Online Brute Force Angriff	-	✓	-	✓	✓	✓	✓	✓	✓	✓
Offline Brute Force Angriff	✓	✓	✓	✓	✓	-	-	-	✓	-
Semi-Offline Brute Force Angriff	✓	✓	-	✓	✓	✓	✓	✓	✓	?
TEE Schwachstelle	✓	✓	-	✓	✓	✓	✓	✓	✓	-
Cold-Boot Angriffe	✓	-	-		✓	✓	✓	✓	✓	?
Statisches Standardpasswort und Master Key	✓	✓	-	✓	✓	✓	✓	✓	✓	✓
Evil-Maid Angriffe	✓	-	-	✓	✓	✓	✓	✓	✓	✓
Umgehen des Fingerabdrucksensors	✓	✓	-	-	✓	✓	✓	✓	✓	✓

Tabelle 6.1: Angriffsübersicht dieser Arbeit

7 Fazit und Ausblick

In dieser Arbeit wurde die aktuelle Sicherheitslage im Bereich der Geräteverschlüsselung von Android analysiert. Hierzu wurde in Kapitel 2 der aktuelle Stand der Technik betrachtet und die eingesetzten Technologien erklärt, die für die Verwendung der unterschiedlichen Arten der Geräteverschlüsselung benötigt werden. Abschnitt 2.2.1 vergleicht die Anforderungen der unterschiedlichen Android Versionen und zeigt darüber einige sehr interessante Erkenntnisse über die einzelnen Android Versionen auf. Die Implementierung des hardwaregebundene Schlüsseltresor ist bis Android 7.0 optional, daher kann der Smartphonehersteller entscheiden dieses Feature zu erfüllen oder nicht. Eine der wichtigsten Erkenntnisse ist, dass bei Versionsnummer 7.0 von Android, aufgrund der Versionsnummer nicht auf die Art der Geräteverschlüsselung geschlossen werden kann ohne nähere Informationen des Smartphones zu besitzen. Jede Version von Android 7.0 aufwärts kann beide Arten besitzen und mindestens die alte FDE. Die Implementierung der FBE ist klar eine SHOULD Anforderung, mit keinem Vermerk, dass sich das in nächster Zeit ändern wird. Daher wird nur ein kleiner Anteil von den 11.5 % der Android 7.0 Geräte (Siehe Tabelle 2.1 eine File Based Encryption verwenden. Das restliche Kapitel beschäftigt sich mit der Funktionsweise der beiden Geräteverschlüsselungen.

In Kapitel 4 wurde ein Bedrohungsmodell anhand der Funktionsweise der beiden Geräteverschlüsselungen und den gewonnenen technischen Erkenntnissen von Kapitel 2 erstellt, welches in Kapitel 5 auf zwei konkrete Angriffsszenarien ausgelegt wurde.

Kapitel 5 baut auf den vorher erarbeiteten Ergebnissen auf und definiert aufgrund dieser Erkenntnisse zwei konkrete Angriffsszenarien. Zusätzlich werden die generischen Schutzmaßnahmen von Android erklärt, welche verwendet werden um Angriffe zu verhindern oder zu erschweren. Der Rest des Kapitels widmet sich der Erklärung und Einschätzung der bekannten Angriffe auf die Android Geräteverschlüsselungen. Dabei wurde jeder einzelne Angriff erklärt, beschrieben welche Gegenmaßnahmen für den Angriff existieren und definiert welche Angriffsfläche der jeweilige Angriff besitzt.

Im letzten Kapitel 6 wurde schlussendlich ein Proof-of-Concept zur Umgehung der Geräteverschlüsselung in der Android Version 7.0 implementiert. Dabei wurde die Methodik und das Setup der Smartphones am Beginn des Kapitels erklärt. Abschnitt 6.3 wurde zum besseren Verständnis der Arbeit und der Vollständigkeit beschrieben. Im restlichen Kapitel wird der Proof-of-Concept zur Umgehung des Fingerabdrucksensors in Android Version 7.0 mit beiden Geräteverschlüsselungsarten beschrieben, wie er implementiert wurde und gezeigt, dass dadurch die Geräteverschlüsselung umgangen werden kann. Zu Schluss wurde eine Übersicht der behandelten Angriffe erstellt, um einen schnelleren Überblick zu erhalten.6.1

Aufgrund der Erkenntnisse, welche im Laufe dieser Arbeit erworben wurden, kann die in Abschnitt 1.2 gestellte Forschungsfrage, *Inwiefern lässt sich die Geräteverschlüsselung in Android 6.0 bis 7.1 umgehen?*", und deren Unterfragen, *Welche Schlüsseltechnologien werden zur Umsetzung der Geräteverschlüsselung in Android 6.0 bis 7.1 verwendet?* und *Welche bereits bekannten Angriffe existieren für die Geräteverschlüsselung in Android 6.0 bis 7.1?* wie folgt beantwortet werden:

Es ist eindeutig möglich die Geräteverschlüsselungsarten von Android 6.0 bis 7.1 mit den verschiedensten Ansätzen und Angriffen zu umgehen. Dabei muss jedes anzugreifende Gerät relativ genau bekannt sein, um eine dafür geeignete Attacke auszuwählen und durchzuführen wie in Abschnitt 6.5 geschildert wird. Die jeweils verwendeten Schlüsseltechnologien der Geräteverschlüsselungsarten werden in Kapitel 2 beschrieben um damit die erste Unterfrage zu beantworten. Das Kapitel 5 beschreibt die bereits bekannten Angriffe und beantwortet damit die zweite Unterfrage. In Kapitel 6 wurde ein Proof-of-Concept auf Basis eines bereits bekannten Angriffs implementiert, um zu zeigen, dass unter Android 7.0 beide Geräteverschlüsselungsarten umgangen werden können. Wie erwähnt wurde, ist der Umstand, dass auf einen Plain Android nahem Smartphone diese Schwachstellen festgestellt und ausgenutzt werden können, sehr schwerwiegend, da davon ausgegangen werden kann, dass andere Smartphonehersteller, die sich an den Quellcode von AOSP halten, damit die selben Schwachstellen in ihren Geräten implementieren. Ein weiterer Punkt, der nicht nur für Smartphones sondern allgemein für die Welt der IT gilt, ist das prinzipiell der Software nicht vertraut wird, der darunterliegenden Hardware dagegen schon. Die in Abschnitt 5.3.4 aufgezeigte Problematik daran ist, dass eine fehlerhafte Hardware schwierig bis gar nicht zu patchen ist. Gerade bei Android Smartphones, welche das quelloffene Betriebssystem nicht Mitverschlüsseln, sondern nur die Benutzerdaten, hat das den Effekt dass ein Angreifer durch Abbilden der wertvollen verschlüsselten Benut-

zerdatenpartition auf ein Gerät ausweichen kann, welches eine Schwachstelle in der Hardware besitzt. Dabei kann er auf ein Gerät mit einem ungepatchtes Betriebssystem ausweichen und somit die Schwachstelle in der Hardware wieder ausnutzen.

Es bleibt abzuwarten, in welche Richtung sich die Geräteverschlüsselung unter Android Geräten entwickelt, da aus jetzigem Stand, sich dasselbe Szenario wie bei der Einführung von FDE wiederholt. Die Hersteller werden derweil nicht verpflichtet die File Based Encryption zu implementieren, was für eine flächendeckende Einführung dieser, im Grunde sichereren Verschlüsselung als FDE, nicht von Vorteil ist. Im Gegenteil, bis die verschiedensten Smartphonehersteller das Feature wirklich implementieren, vergeht sicher noch einige Zeit. Zeit in der Android Smartphones bedauerlicherweise nicht so sicher sind, wie sie es sein könnten.

Abbildungsverzeichnis

2.1 Tortendiagramm zur Tabelle 2.1. 9
2.2 Übersicht der Komponenten von der FDE Verschlüsselung 2.4.[24] 19
2.3 Übersicht der Schlüsselhierarchie.[23][S.14] 20
2.4 Übersicht des Schlüsselmanagements.[23][S.15] 22
2.5 Darstellung Credential Encrypted und Device Encrypted Bereich. 25
2.6 Überblick der File Based Encryption. [28] 26
2.7 Überblick Crypto Footer Android 7.1 27

5.1 Online Brute Force Skript Zeiten für Android 5.0 [23][S.29] 39
5.2 Gatekeeper Logik Android 6.0 [37][Zeile 245] 40
5.3 Gatekeeper Logik Android 7.1 [38][Zeile 247] 41
5.4 Client-Server Kommunikation. [23][S.34] 45
5.5 Visualisierung des Datenverlustes nach Ausschalten des Gerätes [32][S.50] . . 48

6.1 Fingerabdruck Authentifizierungssystem [43] 61
6.2 Quellcodeänderung in der Datei FingerprintDaemonProxy.cpp [44] 62
6.3 Kopieren der Binärdatei und Überprüfen des Prozesses 63

Tabellenverzeichnis

2.1 Tabelle mit Android Versionen und deren Verteilung Stand 06. Juli 2017. [17] . 8

6.1 Angriffsübersicht dieser Arbeit 66

Literaturverzeichnis

[1] "The NSA Files," https://www.theguardian.com/world/interactive/2013/nov/01/snowden-nsa-files-surveillance-revelations-decoded#section/1, accessed: 03. Juni 2017.

[2] "Why did Lavabit shut down," https://www.theguardian.com/commentisfree/2014/may/20/why-did-lavabit-shut-down-snowden-email, accessed: 02. Juni 2017.

[3] "Google FDE by default," https://www.washingtonpost.com/news/the-switch/wp/2014/09/18/newest-androids-will-join-iphones-in-offering-default-encryption-blocking-police/, accessed: 02. Juni 2017.

[4] "Going Dark: Are Technology, Privacy, and Public Safety on a Collision Course?" https://www.fbi.gov/news/speeches/going-dark-are-technology-privacy-and-public-safety-on-a-collision-course, accessed: 03. Juni 2017.

[5] "Google quietly backs away from encrypting new Lollipop devices by default [Updated]," https://arstechnica.com/gadgets/2015/03/google-quietly-backs-away-from-encrypting-new-lollipop-devices-by-default, accessed: 03. Juni 2017.

[6] Google, "Security Enhancements in Android 5.0," https://source.android.com/security/enhancements/enhancements50, accessed: 03. Juni 2017.

[7] ——, "Android 5.0 Compatibility Definition Document," https://source.android.com/compatibility/5.0/android-5.0-cdd.pdf, Januar 2015, accessed: 02. Juni 2017.

[8] ——, "Android 5.1 Compatibility Definition Document," https://source.android.com/compatibility/5.1/android-5.1-cdd.pdf, July 2015, accessed: 02. Juni 2017.

[9] Apple, "A Message to Our Customers," https://www.apple.com/customer-letter/, February 2016, accessed: 03. Juni 2017.

[10] A. Floemer, "FBI hat iPhone 5c geknackt, zieht Klage gegen Apple zurück," http://t3n.de/news/edward-snowden-fbi-apple-iphone-5c-update-687534/, March 2016, accessed: 03. Juni 2017.

[11] R. Molla, "Closing the books on Microsoft's Windows Phone," https://www.recode.net/2017/7/17/15984222/microsoft-windows-phone-mobile-operating-system-android-iphone-ios, July 2017, accessed: 20. Juli 2017.

[12] Google, "Content License," https://source.android.com/source/licenses, March 2017, accessed: 03. Juni 2017.

[13] ——, "Full-Disk Encryption - Android Open Source Project," https://source.android.com/security/encryption/full-disk, accessed: 09. Juni 2017.

[14] ——, "File Based Encryption - Android Open Source Project," https://source.android.com/security/encryption/file-based, accessed: 02. Juni 2017.

[15] J. Drake, Z.Lanier, C. Mulliner, P. Fora, S. Ridley, and G. Wicherski, "Android hacker's handbook," Wiley, Standard ISBN: 978-1-118-60864-7, 04 2014.

[16] Google, "Dashboard," https://developer.android.com/about/dashboards/index.html#Platform, Juni 2017, accessed: 12. Juni 2017.

[17] ——, "Dashboard," https://developer.android.com/about/dashboards/index.html#Platform, Juli 2017, accessed: 09. Juli 2017.

[18] S. Bradner, "Key words for use in RFCs to Indicate Requirement Levels," RFC 2119 (Proposed Standard), Mar. 1997. [Online]. Available: https://www.ietf.org/rfc/rfc2119.txt

[19] Google, "Android 6.0 Compatibility Definition Document," https://source.android.com/compatibility/6.0/android-6.0-cdd.pdf, Oktober 2015, accessed: 02. Juni 2017.

[20] ——, "Android 7.1 Compatibility Definition Document," https://source.android.com/compatibility/7.1/android-7.1-cdd.pdf, Februar 2017, accessed: 02. Juni 2017.

[21] B. Kalinski, "PKCS 5: Password-Based Cryptography Specification Version 2.0," RFC 2898 (Proposed Standard), Sep. 2000. [Online]. Available: https://www.ietf.org/rfc/rfc2898.txt

[22] M. S. Turan, E. Barker, W. Burr, and L. Chen, "Recommendation for Password-Based Key Derivation Part 1: Storage Applications," SP 800-132, Dec. 2010. [Online]. Available: http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-132.pdf

[23] O. Kunz, "Android full-disk encryption: a security assessment," https://www.royalholloway.ac.uk/isg/documents/pdf/technicalreports/2016/rhul-isg-2016-8-oliver-kunz.pdf, Royal Holloway University of London, Tech. Rep. 1, April 2016, accessed: 15. März 2017.

[24] G. Beniamini, "Extracting Qualcomm's KeyMaster Keys - Breaking Android Full Disk Encryption," https://bits-please.blogspot.co.at/2016/06/extracting-qualcomms-keymaster-keys.html, März 2016, accessed: 05. März 2017.

[25] AOSP, "cryptfs.c - Source Code File," https://android.googlesource.com/platform/system/vold/+/android-7.1.1_r11/cryptfs.c, Jänner 2017, accessed: 11. Juni 2017.

[26] ——, "init.rc - Source Code File," https://android.googlesource.com/platform/system/core/+/android-7.1.1_r11/rootdir/init.rc, Jänner 2017, accessed: 11. Juni 2017.

[27] Google, "File-Based Encryption," https://source.android.com/security/encryption/file-based, July 2017, accessed: 10. Juli 2017.

[28] Quarkslab, "A glimpse of ext4 filesystem-level encryption," https://blog.quarkslab.com/a-glimpse-of-ext4-filesystem-level-encryption.html, August 2015, accessed: 13. Juli 2017.

[29] AOSP, "Ext4Crypt.cpp - Source Code File," https://android.googlesource.com/platform/system/vold/+/android-7.1.1_r11/Ext4Crypt.cpp, Jänner 2017, accessed: 12. Juni 2017.

[30] ——, "ScryptParameters.cpp - Source Code File," https://android.googlesource.com/platform/system/vold/+/android-7.1.1_r11/ScryptParameters.cpp, Jänner 2017, accessed: 12. Juni 2017.

[31] N. I. of Standards and Technology, "CVE-2016-2431 Detail," NIST, 2016, https://nvd.nist.gov/vuln/detail/CVE-2016-2431, September 2016, accessed: 19. Juni 2017.

[32] A. Halderman, S. Schoen, N. Heninger, W. Clarkson, and W. Paul, "Lest We Remember: Cold Boot Attacks on Encryption Keys," Communications of the ACM - Security in the Browser, Volume 52 Issue 5, May 2009 https://www.usenix.org/legacy/event/sec08/tech/full_papers/halderman/halderman.pdf, Mai 2009, accessed: 08. Juni 2017.

[33] T. Mueller, M. Spreitzenbarth, and F. Freiling, "FROST - Forensic Recovery of Scrambled Telephones," International Conference on Applied Cryptography and Network Security https://www1.informatik.uni-erlangen.de/filepool/projects/frost/frost.pdf, Oktober 2012, accessed: 05. Juni 2017.

[34] J. Goetzfried and T. Mueller, "Analysing Android's Full Disk Encryption Feature," JoWUA Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications, 2014, http://isyou.info/jowua/papers/jowua-v5n1-4.pdf, März 2014, accessed: 11. Juni 2017.

[35] T. Does and M. Maarse, "Subverting Android 6.0 Fingerprint Authentication," https://homepages.staff.os3.nl/~delaat/rp/2015-2016/p30/report.pdf, University of Amsterdam, Tech. Rep. 1, Februar 2016, accessed: 19. Juni 2017.

[36] R. A. Laurent Simon, "Security Analysis of Android Factory Resets," University of Cambridge, 2015, https://www.cl.cam.ac.uk/~rja14/Papers/fr_most15.pdf, Mai 2015, accessed: 09. Juni 2017.

[37] AOSP, "gatekeeper.cpp - Source Code File," https://android.googlesource.com/platform/system/gatekeeper/+/android-cts-6.0_r22/gatekeeper.cpp, Jänner 2017, accessed: 06. Juni 2017.

[38] ——, "gatekeeper.cpp - Source Code File," https://android.googlesource.com/platform/system/gatekeeper/+/android-7.1.1_r11/gatekeeper.cpp, Jänner 2017, accessed: 06. Juni 2017.

[39] T. Cannon, N. Elenkov, and O. Kunz, "bruteforce-stdcrypto.py - Source Code File," Santoku Linux Distribution, 2015, https://github.com/mlet/Santoku-Linux/tree/master/tools/android/android_bruteforce_stdcrypto, Juli 2015, accessed: 28. Juni 2017.

[40] C. Cimpanu, "QuadRooter Android Security Bugs Affect over 900 Million Devices," http://news.softpedia.com/news/quadrooter-android-security-bugs-affect-over-900-million-devices-507052.shtml, August 2016, accessed: 17. Juni 2017.

[41] N. I. of Standards and Technology, "CVE-2015-6639 Detail," NIST, 2016, https://nvd.nist.gov/vuln/detail/CVE-2015-6639, Juni 2015, accessed: 19. Juni 2017.

[42] A. Tereshkin, "Evil Maid goes after TrueCrypt!" SIN '10 Proceedings of the 3rd international conference on Security of information and networks, 2010, http://theinvisiblethings.blogspot.co.at/2009/10/evil-maid-goes-after-truecrypt.html, Oktober 2009, accessed: 11. Juni 2017.

[43] Google, "Fingerprint HAL," https://source.android.com/security/authentication/fingerprint-hal, März 2017, accessed: 02. Juni 2017.

[44] ——, "FingerprintDaemonProxy.cpp - Source Code File," https://android.googlesource.com/platform/system/core/+/android-7.1.1_r11/fingerprintd/FingerprintDaemonProxy.cpp, Jänner 2017, accessed: 06. Juni 2017.