Information Security

Automatische Erkennung manipulierter App-Installationen unter Android

Diplomarbeit

zur Erlangung des akademischen Grades

Diplom-Ingenieur

eingereicht von

Thomas Pokorny, BSc

1710619830

im Rahmen des

Studienganges Information Security an der Fachhochschule St. Pölten

Betreuung

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

Mitwirkung:

St. Pölten, 12. Juni 2019 _____ _____

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

*

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*

Abstrakt

Die Anzahl an Cyber-Bedrohungen im digitalen Zeitalter ist eine stetig steigende und dadurch ernstzunehmende Situation, welche von der Gesellschaft wahrgenommen werden muss. Heutzutage sind Smartphones im täglichen Leben allgegenwärtig, da sie unseren Alltag erleichtern. Dies haben jedoch auch Kriminelle erkannt, wodurch Smartphones ein potenzielles Ziel geworden sind. Daher müssen Gegenmaßnahmen für solch gezielte Angriffe entwickelt werden, um rechtzeitig Manipulationen erkennen zu können, um dadurch die Sicherheit von Daten zu gewährleisten. In dieser Arbeit wurde ein Lösungsansatz entwickelt, welcher als Proof-of-Concept realisiert wurde, um das Erkennen von manipulierten Apps durchführen zu können. Dabei wurde der Fokus auf die Verwendung von On-Board Tools gelegt, welche das Smartphone, sowie das eingesetzte Betriebssystem für die Analysen mitbringt, gelegt.
Resultierend zeigte sich, dass der Lösungsansatz es erlaubt, Daten ohne speziellen Benutzerberechtigungen zu sichern, sowie die Analyse von Apps, mittels definierter Identifikationsmerkmalen automatisiert auf Manipulationen zu prüfen. Schlussendlich wurde gezeigt, dass die Möglichkeit für die Prüfung auf Manipulationen, keine Modifizierungen am System notwendig sind, sowie die Durchführung ohne kommerziellen Tools möglich ist. Dieser Lösungsansatz bringt Gewissheit, ob Apps am Smartphone manipuliert wurden bzw. ob es Veränderungen zwischen zwei oder mehreren Systemzuständen gegeben hat.

Abstract

The number of cyber threats in the digital age is an ever-increasing and therefore serious situation, which must be taken into account by those of society. These days, smartphones are ubiquitous in everyday life because they make our daily lives easier. However, criminals have also recognized this, making the smartphone a potential target. Therefore, countermeasures for such targeted attacks must be developed in order to detect manipulations in time and thus ensure the security of the data. In this thesis a solution was developed which was realized as proof-of-concept in order to be able to detect manipulated apps. The focus was on the use of on-board tools, which the smartphone as well as the operating system used for the analyses bring along. As a result, it was shown that the proposed approach makes it possible to secure data without special permissions and to automatically compare the analysis of apps for manipulations using defined identification features.

Finally, it was shown that the possibility of checking for manipulations, no modifications on the system are necessary, as well as the implementation without commercial tools is possible. This approach provides certainty as to whether apps on the smartphone have been manipulated or whether there have been changes between two or more system states.

Inhaltsverzeichnis

1. Einleitung

Smartphones sind in heutiger Zeit nicht mehr wegzudenken. Sie sind als ständiger Begleiter immer und überall dabei. Smartphones erleichtern den täglichen Alltag, wenn es darum geht Dinge zu kaufen, ein E-Mail zu schreiben, oder mit anderen zu kommunizieren via Messenger Services. Für all diese Anwendungsfälle gibt es mittlerweile eine enorme Anzahl an Applikationen (Apps), welche diese Funktionialität bieten. Aufgrund der schnellen Entwicklung von Apps, steigt auch das Risiko, Opfer von Schadsoftware bzw. manipulierter Software zu werden. Dies zu erkennen ist oftmals nur mit sehr hohem Aufwand möglich, bzw. erst dann wenn es schon zu spät ist und sämtliche persönliche oder berufliche Daten gestohlen wurden. Infolgedessen soll in dieser Diplomarbeit ein Lösungsansatz gezeigt werden, welche Möglichkeiten es gibt, um manipulierte Smartphone-Apps zu erkennen.

Im Bereich der digitalen Forensik ist ein enormer Bedarf an der Auswertung von mobilen Endgeräten zu vermerken. Entwicklungen von speziellen Tools, welche gezielt für die Datensicherung bzw. für die Aufbereitung von solchen Geräten herangezogen werden, befinden sich unter ständiger Weiterentwicklung. Jedoch befinden sich diese immer im Rückstand, wenn es um die Kompatibilität von Geräten geht, da die Entwicklung von Endgeräten in der Regel schneller voran geht als die Implementierung neuer Technologien in forensischen Tools. Dabei sind einige Herausforderungen die es zu beachten gibt, wie beispielsweise der Einsatz unterschiedlicher proprietärer Betriebssysteme, Embedded Filesysteme, Applikationen, sowie Peripherie. In optimalen Fällen werden diese Komponenten unterstützt, jedoch besteht auch die Möglichkeit, dass keine Kompatibilität besteht und der Zugriff auf das Endgerät und deren Daten nicht möglich ist.

Grundsätzlich hängt die Auswahl einer forensischen Software vom angestrebten Ziel ab. Abhängig von der eingesetzten Technologie eines Smartphones, fällt die Entscheidung auf unterschiedliche forensische Produkte. Die Evaluierung von verfügbaren Produkten unterschiedlicher Hersteller macht durchaus Sinn, um ein gewünschtes Ergebnis zu erreichen. Dies kann durch die Erstellung einer Matrix, bestehend aus Produkten und deren Funktionalität, erhoben werden.[1]

1.1. Problemstellung und Forschungsfrage

In der digitalen Forensik, speziell im Bereich der mobilen Forensik, gibt es eine Vielzahl an Produkten bzw. Softwarelösungen, um entsprechende Datensicherungen oder auch Beweismittel zu sichern. Im kriminalpolizeilichen, sowie behördlichem Bereich haben sich jedoch nur eine handvoll an Herstellern durchgesetzt. Um eine gezielte Auswertung eines Smartphones machen zu können, ist daher die entsprechende Software zu wählen, da alle Produkte ihre Stärken und Schwächen in unterschiedlicher Art und Weise aufweisen. Daher kann eine forensiche Auswertung einige Herausforderungen mit sich bringen, da oftmals mehrere Ansätze gewählt werden müssen, um das gewünschte Ergebnis zu erreichen und der tatsächliche Aufwand viel höher ist als womöglich notwendig wäre, da eine Datensicherung in der Regel vielmehr beinhaltet, jedoch nur ein spezifischer Teil benötigt wird.
Eine rasche Auswertung über den Zustand von Apps, welche sämtliche Informationen über die Echtheit bzw. Manipulation liefert, ist in heutiger Zeit, aufgrund der hohen Anzahl an Sicherheitsrisiken oftmals notwendig. Speziell in Ländern in denen die Nutzung des Internets streng reguliert ist und von staatlicher Seite ein großer Einfluss herrscht, sind Manipulationen von Apps nicht auszuschließen.
Aufgrund dieser Ausgangssituation werden in dieser Diplomarbeit folgende Forschungsfragen behandelt:

- **Welche Möglichkeit gibt es Manipulationen auf Applikationsebene zu erkennen?**
  Um die Beantwortung dieser Forschungsfrage durchführen zu können, lassen sich folgende Unterfragen ableiten:
  - **Ist die Manipulation von Apps zur Laufzeit möglich?**
    Die Nutzung von mehreren Apps zur selben Zeit ist für Smartphones die Regel, daher ist die Verbarbeitung von Daten permanent im Gange. Manipulationen von Apps durch Dritte könnte einen Datenverlust für Besitzer von Smartphones bedeuten.
  - **Wie könnte die Umsetzung als Softwarelösung aussehen, um mit einfachen Systemmitteln automatisiert Systemzustände vergleichen zu können?**
    Moderne Smartphones vefügen über stendig steigende Speicherressourcen und dadurch wächst auch die Zahl an Apps, die ein Gerät zur Verfügung stellen kann. Da der Aufwand für die Untersuchung jeder App manuell einen sehr hohen Aufwand bedeuten würde, ist die Entwicklung einer Softwarelösung als Proof-of-Concept notwendig.

1.2. Google's Android im Fokus

Aufgrund der Verbreitung von Android am Smartphone-Markt, wurde das Augenmerk auf dieses System gelegt. Google hält einen weltweiten Marktanteil mit dem mobilen Betriebssystem bei 87,7 Prozent. Dem gegenüber steht Apple mit dem Betriebssystem IOS und einem Marktanteil von 12,1 Prozent. Der Rest, nämlich 0,2 Prozent, teilte sich Microsoft und Samsung mit deren entwickeltem Betriebssystem Tizen, stand 2017.[2] Android ist eine der heute am weitesten verbreiteten Smartphone-Plattformen. Smartphones ersetzen die traditionellen Mobiltelefone als Taschencomputer und Multimedia-Gerät in einem. Diese persönlichen Geräte bieten einen Einblick in das Leben des Besitzers. Ein Kalender mit den Daten des Benutzers oder Benutzerin, ein Telefonbuch mit einer Liste von Kontakten, Social-Media-Konten und Bankanwendungen sind nur eine kleine Teilmenge aller Informationen, die auf einem typischen Smartphone zu finden sind.[3] Das Erkennen von manipulierten Smartphone-Apps hat nicht nur bei Android Relevanz, sondern auch bei allen anderen Betriebssystemen, jedoch würde dies den Rahmen dieser Arbeit sprengen, sollten alle Systeme berücksichtigt werden. Nichtsdestotrotz funktionieren ähnliche Konzepte auch auf anderen Systemen.

1.3. Abgrenzung der Arbeit

Diese Diplomarbeit konzentriert sich nicht auf die Evaluierung einzelner Produkte verschiedenster Hersteller, sondern fokussiert sich auf die Überprüfung der Integrität von Smartphone-Apps, bzw. der Validierung auf Echtheit einer Applikation am Smartphone. Darunter versteht man die Überprüfung auf Veränderung bzw. Manipulation von Daten. Dies soll anhand eines möglichen Lösungsansatzes, mittels eines Proof-of-Concept dargestellt werden. Dafür wird eine Software entwickelt, welche festgehaltene Systemzustände (Sicherung von Apps) entsprechend verarbeitet und etwaige Manipulationen erkennt und diese entsprechend ausgibt.

1.4. Eingesetzte Hardwareressourcen

Für diese Diplomarbeit stand ein Samsung Galaxy S5 Neo mit der Version 6.0.1 zur Verfügung, um sämtliche Tests durchführen zu können. Des Weiteren wurde für die Entwicklung des Proof-of-Concepts Python3 als Programmiersprache gewählt, auf Basis der Linux Distribution Ubuntu 18.04 LTS. Da Python plattformunabhänig funktioniern, kann die Implementierung des entwickelten Lösungsansatzes auch auf anderen Betriebssystemen eingesetzt werden.

1.5. Aufbau dieser Arbeit

Kapitel zwei befasst sich mit Lösungsansätzen aus anderen Forschungsarbeiten, die in diesem Bereich ähnliche Probleme lösen bzw. die Sicht aus einem anderen Blickwinkel zu dem gleichen Problem - die Erkennung von Manipulationen am Smartphone - beschäftigten.
In Kapitel drei werden grundlegende Informationen über Android erläutert. Dies beinhaltet alle notwendigen Teile, welche aus forensischer Sicht bzw. für die Erstellung des Proof-of-Concept notwendig sind. Des Weiteren wird auf den Entwicklungsprozess von Apps, sowie der Installationsprozess am Smartphone eingegangen.
Kapitel vier erklärt die wichtigesten Abläufe in der digitalen Forensik. Es wird auf den grundlegenden Prozessablauf von Beginn der Datensicherung, bis hin zur Auswertung und Analyse von gesammelten Daten eingegangen.
Kapitel fünf behandelt den praxisorientierten Teil. Es wird das Konzept des Proof-of-Concept vorgestellt, sowie die Erstellung eines Backups mittels der Android Debug Bridge (ADB) durchgeführt. Das Kapitel schließt mit der Durchführung eines möglichen Szenario, um zu zeigen wie in gelebter Praxis der Vergleich von zwei Systemzuständen aussehen könnte.
Kapitel sechs beschreibt zusammenfassend einen Rückblick auf diese Arbeit und den möglichen Problemen, die bei der Entwicklung solch einer Software auftreten können, sowie ein Fazit darüber, welchen Mehrwert bzw. Vorteil man durch diesen Lösungsansatz hat und welche Alternativen es gibt.

2. Related Work

Nicht nur in dieser Arbeit wird ein Lösungsansatz erarbeitet um die Problemstellung und Forschungsfrage zu beantworten, sondern auch in anderen wissenschaftlichen Arbeiten werden ähnliche Konzepte entwickelt und hinterfragt, um Manipulationen in Apps identifizieren zu können. Dabei werden ähnliche Blickwinkel auf das Problem gelegt, jedoch liegt der eigentlichen Fokus immer darin, Malware[1] zu erkennen und entsprechende präventive Maßnahmen zu erarbeiten, um dem Risiko von Manipulationen und Datendiebstahl entgegenzuwirken. Im folgenden werden Forschungsansätze bzw. Lösungen anderer Arbeiten aufgezeigt, um einen Überblick deren zu bekommen, sowie mögliche Vergleiche zwischen anderen wissenschaftlichen Arbeiten zu diskutieren.

2.1. DroidOLytics : Robust Feature Signature for Repackaged Android apps on Official and Third party android markets

In dieser wissenschaftlichen Arbeit wird das Problem zur Erkennung von manipulierten Android-Apps mittels speziellen Signaturen von bekannten Malware-Typen analysiert. Im Detail bedeutet dies, unbekannte - bis Dato unentdeckte - Varianten von bereits bekannten Malware-Typen zu identifizieren.
Dieser Ansatz sucht nach statistischen Merkmalen im Programmcode, welche auf Manipulationen in einer App hinweisen. Um die Ähnlichkeit von unbekannten Programmcode mit bekannten bösartigen Programmcode zu erkennen, werden sogenannte Ähnlichkeitssignaturen verwendet, um eine neuartige Variante eines bereits bekannten Malware-Typs zu identifizieren.
Da die Kompilierung (engl. repackaging) einer App ohne Zerlegung (engl. disassembling) der APK-Datei möglich ist, kann ohne Verlust inhaltlicher bzw. funktionalen Eigenschaften eine App mit Malware neu erstellt werden. Dies bedeutet, dass ein Entwickler oder Entwicklerin von Schadsoftware, mit minimalen Aufwand, fremden Programmcode einer App zufügen und dadurch Manipulationen am Smartphone verursachen kann.
Der geplante Lösungsansatz ist zusätzlich gegen Verschleierungstechniken (engl. obfuscation) resistent.

[1] Malicious Software z. Dt. Schadsoftware

Dieses Verfahren basiert auf einem entwickelten Algorithmus, welcher auf Ähnlichkeiten von Funktionen im Programmcode trainiert ist. Legitime Apps weisen mit hoher Wahrscheinlichkeit Ähnlichkeiten auf, aufgrund der Verwendung von gleichen Programmcode, da oftmals nur durch die Versionierung maginal etwas verändert wird. Dafür wurde der Algorithmus mit 1260 Beispieldaten (engl. Samples) bzw. Ähnlichkeitssignaturen von 57 verschiedenen Maleware-Typen trainiert. Diese Signaturen wurden gegen Apps aus dem Google Play Store, sowie Drittanbietern, getestet. Bei diesem Test wurde auf Abweichungen im Programmcode und Repackaging bekannter Malware geachtet. Effektiv wurden 6779 Dateien, abstammend von Google Play, sowie 672 Dateien von Drittanbieter-Apps analysiert. Der Test zeigte, dass 40 Dateien von Google Play als verdächtig identifziert wurden und von Drittanbietern 21 betroffen sind. Daraus resultiert eine Genauigkeit von 99,4 Prozent bei Google Play und 96,87 Prozent bei Drittanbieter-Apps.
Das Ergebnis dieser Forschungsarbeit zeigt, dass der entwickelte Lösungsansatz robust gegen unbekannten Schadcode ist, welcher mit verschiedensten Verschleierungstechniken bearbeitet wurde und zusätzlich auch nicht von bekannten Anti-Viren Herstellern erkannt wird. Verdächtiger Programmcode welcher vom Ansatz erkannt wurde, wurde manuell analysiert, um Ähnlichkeiten mit bestehenden Malware-Signaturen zu überprüfen.[4]

2.2. Detecting Illegaly-copied Apps on Android Devices

Hierbei wird die Problematik manipulierter Apps über einen ähnlichen Ansatz wie in dieser Diplomarbeit behandelt. Jedoch unterscheidet sich dieser davon, dass ein Lösungsansatz entwickelt wurde, um illegal kopierte Apps zu erkennen und präventiv ein Mechanismus entwickelt wurde, um solchen Apps die Ausführung zu unterbinden.
Hervor geht, dass diese Forschungsarbeit durchgeführt wurde, da durch das Kopieren von Apps möglicherweise Urheberrechte von Entwickler und Entwicklerinnen verletzt werden und man diese schützen möchte. Des Weiteren bewirkt der Einsatz solcher kopierten Apps, dass Schadsoftware eingeschleust werden könnte. Der geplante Detektor bestimmt ob eine App illegal kopiert wurde und blockiert diese anschließend wenn notwendig, mittels einer Signatur basierenden Technik. Zusätzlich wird sämtlicher interner, als auch externer Speicher auf illegale Apps durchsucht. Die Umsetzung dieses Detektors wird mit einem unsichtbaren Manager - also einem Service im Hintergrund laufend - gesteuert, um den Benutzer oder Benutzerin daran zu hindern diesen Mechanismus zu löschen bzw. zu stoppen.
Das Resultat dieser Arbeit zeigt, dass installierte, illegal kopierte Apps, innerhalb einer Sekunde erkannt werden, sowie Apps, welche am internen/externen Speicher liegen, innerhalb von zwei Sekunden entdeckt werden. Abhängig von der Anzahl an installierten Apps, sowie am Speicher befindlichen, erhöhen

sich die benötigten Ressourcen. Dies bedeutet, dass schlussendlich die Performance der Android Plattform leidet, je mehr Apps bzw. Speicher genutzt wird. Um einen effizienteren Lösungsansatz erfolgreich umsetzen zu können, benötigt es in diesem Bereich weitere Studien, um einen performanteren Suchalgorithmus zu entwickeln.[5]

2.3. Tamper Detection Scheme Using Signature Segregation on Android Platform

Da Android-Apps nicht sicher gegen die Neukompilierung (repackaging) geschützt sind, wurde als Gegenmaßnahme in dieser Arbeit ein APK-Tester (engl. Attester) enwickelt. Dieser soll in der Lage sein manipulierte Apps zu erkennen, mit der Grundidee, eine sichere Laufzeitumgebung zu gewährleisten. Um solch einer Attacke entgegenzuwirken, wurden für die Implementierung dieses APK-Tester das AOSP - *Android Open Source Project* - modifziert. Des Weiteren wurde dafür ein *Extractor, Integrity Verifier, Decryptor* und *Network Module* entwickelt, sowie ein eigener *Signal Handler* und *Warning Message* um den Benutzer oder Benutzerin auf mögliche Manipulationen hinzuweisen.
Jedoch hat dieser Lösungsansatz Limitierungen, da dieser APK-Tester grundsätzlich nur gegen den APK-Packetnamen prüft. Sollte ein Angreifer oder Angreiferin den Namen des Pakets ändern und fügt Schadcode hinzu, so würde dies nicht erkannt werden. Eine weitere Limitierung ist, dass der APK-Tester nur vom Hersteller angewendet werden kann, da der Programmcode der Android-Platform modifiziert werden muss.[6]

2.4. An Analysis of Whatsapp Forensics in Android Smartphones

Bei dieser wissenschaftlichen Arbeit, wurde der Fokus auf die Durchführung forensischer Analysen des Messengers Whatsapp, sowie weiteren ähnlichen Apps gelegt. Dabei wurde erkannt, dass Nachrichten, welche mittels Whatsapp versendet bzw. empfangen wurden, in einer SQLite Datenbank gespeichert werden. Whatsapp verwendet dazu einen 192-bit langen Schlüssel um diese Datenbank zu verschlüsseln, jedoch ist das eingesetzte AES-Verfahren nicht sicher. Grund dafür ist, dass der Schlüssel mittels einem Python-Skript ausgelesen werden kann. Dadurch können automatisiert Nachrichten aus der Datenbank exportiert werden.
Um eine Manipulation der Datenbank erkennen zu können, muss ein Prozess definiert werden, wie diese Informationen verglichen werden können.[7]

2.5. Android Forensics Techniques

Um spezifische forensische Techniken, für die Prüfung mobiler Geräte mit Android, beschäftigt man sich in dieser Arbeit. Auf den Prozess für die Sicherung und Handhabung der Daten wird hier genau eingegangen, sowie verschiedene kommerzielle Hersteller in dem Bereich evaluiert. Strategien für die Umgehung des Pass-Code, als auch Techniken um Root-Berechtigungen zu erlangen werden beschrieben.[8]
Diese Techniken können hilfreich sein, wenn es darum geht Daten zu sichern, sowie die Erkennung von Manipulationen am Smartphone.

2.6. Android forensics: Automated data collection and reporting from a mobile device

In dieser Forschung wird ein Prototyp eines Systems vorgestellt, welches für das System-Monitoring in Enterprise Umgebungen vorgesehen ist. Dieses System benötigt weder Root-Berechtigungen oder muss anderweitig an der Systemarchitektur manipuliert werden. Dabei wurde in dieser Arbeit eine neuartige Strategie für die Implementierung verschiedener Komponenten in Android vorgestellt, welche nützlich für die Überwachung des Systems sind. Zugleich dient diese Forschungsarbeit als Leitfaden für den Zugriff auf Datensets über die Standard-API von Android.[9]

2.7. Android forensics: Simplifying cell phone examinations

Einem Überblick über eine Vielzahl an forensischen Tools wurde hierbei nachgegangen. Der Fokus wurde dabei auf das Sichern von Daten am Smartphone gelegt. Die Evaluierung der verschiedensten Tools zeigte deren Vor- und Nachteile bei der Gewinnung von Informationen. Dabei wurden unterschiedliche Datenmengen erzielt.[10] Die Erkenntnis, dass unterschiedliche Ergebnisse bei der Datensicherung erzielt wurden, zeigt dass es nicht nur einen Lösungsansatz gibt, um Smartphones forensisch analysieren zu können.

2.8. Android Malware Forensics: Reconstruction of Malicious Events

Manipulationen am System kann durch Malware erfolgen. Ein Ansatz diese zu erkennen, ist die Analyse von ausgeführten Events, welche von der Schadsoftware ausgelöst werden. Diese Herangehensweise wird in dieser Arbeit durchgeführt. Im Detail wurden hierbei Verfahren gezeigt, wie das Verhalten von Malware analysiert werden kann. Dies wurde anhand einer realen Malwareanalyse durchgeführt. Das Resultat ergab, dass ein Lösungsansatz entwickelt wurde, welcher die angeforderten Benutzerrechte einer APK-Datei ermittelt und mit den tatsächlichen vergebenen abgleicht. Malware fordert oftmals mehr Berechtigungen an als eine App benötigt.[11] Um einen besseren Schutz vor Malware erreichen zu können, müssen weitere Mechanismen implementiert werden, welche per Laufzeit angeforderte Berechtigungen prüft.

2.9. A study of user data integrity during acquisition of Android devices

In dieser Arbeit wird eine schon bekannte Methode zur Erfassung von Android-Geräten unter Verwendung des Recovery-Mode eingesetzt. Diese wertet Variablen des Android Recovery-Mode aus, die die Datenintegrität zum Zeitpunkt der Datenerfassung potenziell beeinträchtigen. Darauf basierend, wurde eine Analyse durchgeführt und ein Tool zur Datenerfassung entwickelt, welches die Integrität der erfassten Daten sicherstellt. In Folge dessen, wurde mittels Fallstudie demonstriert, welche Fähigkeit das Tool zur Wahrung der Datenintegrität aufweist.
Resultierend aus der Arbeit muss erwähnt werden, dass sämtliche Tests auf Geräten von Samsung durchgeführt wurden. Weitere Tests sollten auf Geräten anderer Hersteller durchgeführt werden. Der Fokus hierbei lag auf der Sicherstellung der Daten und sollte für zukünftige Arbeiten zusätzlich die Analyse mit einbeziehen.[12]

2.10. Toward a general collection methodology for Android devices

Der Hauptfokus dieser wissenschaftlichen Arbeit ist ein allgemeiner Prozess zur Datenerfassung auf Android-Geräten, welcher an mehreren spezifischen Geräten durchgeführt wird. Als Resultat wurde eine allgemeine Methode zur digitalen forensischen Sammlung auf Android-Geräten demonstriert. Durch eine spezielle Bootmethode, die die Verwendung von benutzerdefinierten Recovery-Bootimages ermöglicht, können Daten auf Android-Geräten gesammelt werden. Die Verwendung des Recovery-Bootimages bietet ein konsistentes, wiederholbares Verfahren zum Sammeln von Daten. Um einen Überblick über sämtliche Geräte zu bekommen, welche dieses Verfahren unterstützen, sollte für weitere Forschungsarbeiten dies evaluiert werden.[13]

2.11. New acquisition method based on firmware update protocols for Android smartphones

Hierbei wird eine neue Methode für die Erfassung von Daten des gesamten Flash-Speichers entwickelt. Basierend am Update-Protokoll der Firmware, wurden Analysen am Smartphone angestellt, welche es erlaubten eine Möglichkeit zu entwickeln, um die Datenintegrität nicht zu gefährden. Daraus resultierend ergab sich die Erkenntnis, dass dieser Lösungsansatz gegenüber anderen Ansätzen überlegen, im Bezug auf die Integritätsgarantie, Zeitersparnis bei der Durchführung der Sicherung, sowie bei der Erstellung von *physical dumps* mit aktiven Sperr-Bildschirm ist.[14] Für zukünftige Forschungsarbeiten mit diesem Lösungsansatz muss möglicherweise das Update-Protokoll neu analysiert werden, da es zu späteren Zeitpunkten nicht mehr möglich ist dies so anzuwenden, aufgrund von möglichen Änderungen des Update-Mechanismus bei neuen Android-Versionen.

2.12. Automated forensic analysis of mobile applications on Android devices

Ein voll automatisiertes Tool, für forensische Analysen auf Android-Systemen wurde in dieser Forschungsarbeit entwickelt. Dieses Tool führt statische Analysen von APK-Dateien durch und generiert Grafiken, welche Flussdiagramme des Programmcodes darstellen. Des Weiteren werden sämtliche Abhängigkeiten zwischen den Funktionen angezeigt. Ein weiteres Feature dieses Tools, liegt in der Erkennung von sensiblen Informationen und deren Speicherort. Getestet wurde das entwickelte Tool mit 100

zufällig gewählten Apps. Resultierend zeigte sich, dass die Auswertung aller Tools über 64 Stunden dauerte, sowie sensible Informationen (GPS-Daten) am lokalen Speicher entdeckt wurden.[15]
Für den Abgleich von Manipulationen bietet dieses Tool eine gute Grundlage, um Daten auszuwerten und diese für weitere Forschungsarbeiten zu verwenden.

2.13. Network and device forensic analysis of Android social-messaging applications

In dieser Arbeit liegt die Forschung im Bereich der Datensammlung mittels forensischen Analysen der gespeicherten Daten am Smartphone, sowie des Netzwerkverkehrs mit Fokus auf Instant-Messenger-Apps. Diese Forschungsarbeit zeigt welche Features solche Messenger anbieten, um Informationen zurückverfolgen zu können. Zusätzlich, welche Merkmale Instant-Messaging-Apps an Spuren hinterlassen, die es ermöglichen, verdächtige Daten zu rekonstruieren oder zumindest teilweise zu rekonstruieren, und ob die forensische Netzwerkanalyse die Rekonstruktion von Daten ermöglicht. Das Resultat zeigte, dass in den meisten Fällen es möglich war, Daten zu rekonstruieren wie beispielsweise Passwörter, Screenshots, Bilder, Videos, Audio Dateien, sowie gesendete Nachrichten und Profilbilder uvm..[16]
Dieser Lösungsansatz bietet eine gute Grundlage, um gezielt nach Informationen am Smartphone zu suchen, welche auch für Angreifer und Angreiferinnen sehr interessant sind. Dies zeigt auch, dass diese Daten einen erhöhten Schutzbedarf benötigten.

3. Android in der Theorie

In diesem Kapitel werden jene Komponenten bzw. Eigenschaften erläutert, welche das Betriebssystem Android auszeichnen, die für das Ausführen einer App notwendig sind und daher ein Rolle in dieser Arbeit darstellen. Darunter wird auch der Prozessablauf von der Erstellung einer App, einschließlich der Möglichkeiten eine App im Google Play-Store zu veröffentlichen, sowie mögliche Alternativen verstanden. Des Weiteren werden auch die grundlegenden Komponenten des Betriebssystems behandelt, um ein Verständnis der Architekur zu bekommen.

3.1. Android Architektur

Die Architektur von Android ist nahezu wie alle anderen Plattformen aufgebaut. Wie ein Stapel (engl. Stack) mit unterschiedlichen Schichten (engl. Layers), welche aufeinander aufbauen. Wobei eine Schicht der jeweils darüber liegenden Schicht Ressourcen zur Verfügung stellt. Ganz unten befindet sich ein Linux Kernel. Dieser ist verantwortlich für die Bereitstellung sämtlicher Ressourcen, die für das Betriebssystem gefordert werden, wie z.B. Hardwaretreiber die spezifisch für das jeweilige Smartphone vom Hardwarehersteller sind. Über dem Kernel befinden sich die nativen Bibliotheken (engl. Libraries). Diese sind als Code-Module zu betrachten, welche in nativen Maschinencode kompiliert sind und für bestimmte Services zur Verfügung stehen, um Apps und andere Programme lauffähig zu machen. Darin inkludiert ist beispielsweise *Surface Manager*[1]*, Web Kit*[2] *und SQLite*[3]. Diese Libraries werden von Prozessen des Kernels ausgeführt. Jede ausgeführte Applikation bedient sich einer eigenen Android Laufzeitumgebungs-Instanz (engl. Runtime enviroment), der sogenannten *Dalvik Virtual Machine*.[17] Die Dalivk VM ist jedoch nur bis zur Version 4.x Hauptbestandteil von Android und wurde mit der Version 5.x durch die *Android Runtime (ART)* abgelöst.

[1] Verantwortlich für die grafische Darstellung am Gerät

[2] Web-rendering für den Standardbrowser

[3] Standard Datenbanktechnologie für Android Plattformen

3.1.1. Dalvik Virtual Machine

Die *Dalvik VM* ist eine virtuelle Ausführungsumgebung, welche bis zur Android Version 4.x zum Einsatz kommt. Diese Art der virtuellen Umgebung ist für Java-programmierte Applikationen entwickelt worden. Jedoch muss Java-Code in das *DEX*[4]*-Format* konvertiert werden, um diesen zur Ausführung bringen zu können. Kompiliert wird mit einem *Just-in-Time (JiT)* Übersetzer (engl. Compiler). Dies bedeutet, dass jedes mal vor der Ausführung, der Bytecode in Maschinencode interpretiert wird. Dieser Vorgang wird jedes mal beim Ausführen der Applikation ausgeführt, wodurch die Performance des Systems leidet. Für jeden Prozess bzw. Applikation wird bei der Ausführung eine eigene virtuelle Instanz erzeugt.[17] Eine interessante Eigenschaft der *Dalvik VM* ist, dass mithilfe des Tools *dx* Java-Binärdateien (*.class Dateien) in das DEX-Format umgewandelt werden können und dadurch optimiert werden, um auf Android-basierten Geräten mit eingesetzter *Dalvik VM* verwendet werden können.[18]

3.1.2. Android Runtime - ART

Mit der Ablöse der *Dalvik VM* wurde mit der Version 5 die *Andorid Runtime (ART)* implementiert und ist bis heute in der aktuellen Version 8 Bestandteil. Applikationen welche für die *Dalvik VM* entwickelt wurden sind jedoch weiterhin kompatibel. Bei ART kommt das *Ahead Of Time (AOT)* Verfahren zum Einsatz. Dieses Verfahren wirkt sich positiv auf die Performance des Geräts aus, da hierbei die App bei der Installation nur einmal kompiliert wird und in weiterer Folge bei jeder Ausführung der App sofort von der CPU ausgeführt werden kann.

3.2. Systemzugriff via ADB-Schnittstelle

Android bietet prinzipiell in allen Versionen eine eigene Schnittstelle, um mit externen Geräten kommunizieren zu können. Die *Android Debug Bridge (ADB)* ist eine Kommandozeilen basierende Schnittstelle, welche eine Vielzahl an Möglichkeiten zur Verfügung stellt, wie z.B. Installation von Apps, Debugging, Backup, sowie den Zugriff auf das darunter befindliche Dateisystem. Die Schnittstelle benötigt drei Komponenten für die Kommunikation zwischen einem Android-Gerät und einem externen Rechner.[19]

- Client Software – Diese wird benötigt um einzelne ADB-Befehle vom externen Rechner über die Kommandozeile abzusetzen.
- ADB Daemon – Der ADB Daemon-Dienst ist ein Prozess der im Hintergrund eines Systems läuft. Dieser führt ebenfalls ADB-Befehle aus.

[4] Dalvik Executable

- Server Software – Die Serverkomponente managed die Kommunikation zwischen dem Client und dem Daemon-Prozess. Auch dieser ist als Hintergrundprozess am externen Rechner zu finden.

Auf diese Art und Weise können Daten vom Gerät exportiert werden. Eine weitere Möglichkeit Daten von dem Gerät zu exportieren, kann durch manuelle Untersuchung des Gerätespeichers erfolgen. Dies bietet den Vorteil, dass keine speziellen Tools benötigt werden. In der Praxis ist jedoch diese Art der Datengewinnung eher mit Nachteilen zu betrachten, da nur Daten gesichert werden können welche für den Benutzer oder Benutzerin ersichtlich sind und nicht jene Daten, welche eventuell vom System als gelöscht markiert sind.

3.3. Eingesetzte Übertragungsprotokolle

Media Transfer Protocol - Für die Kommunikation zwischen Android und einem PC steht das Media Transfer Protocol (MTP) zur Verfügung. Dieses wurde im Jahr 2004 von Microsoft vorgestellt und ist als Weiterentwicklung des Picture Transfer Protocols auf allen neueren Android Versionen implementiert. Grundsätzlich wurde es entwickelt um weitere Gerätetypen, neben MP3-Playern und Digitalkameras, kompatibel zu machen. Die Funktionsweise dieses Protokolls verhällt sich nicht so wie der USB-Massenspeicher, sondern zeigt nur die zur Verfügung gestellten Dateien an. Somit bleibt das darunter befindliche Dateisystem unberührt und auch kein direkter Zugriff ist möglich. Wenn ein Gerät am PC angeschlossen wird schickt dieser eine Anfrage an das mobile Gerät und dieses reagiert mit einer Liste von freigegebenen Verzeichnissen bzw. sonstigen Dateien. Dargestellt wird das Mobilgerät bespielsweise im Windows Explorer als eigenständiges Gerät, welches wie gewohnt durchsucht werden kann. Linux benötigt dafür die *libmtp* Library, welche grundsätzlich in allen modernen Linux Distributionen enthalten ist. Ledeglich Apple's Betriebssystem Mac OS X unterstützt das Protokoll nicht. Apple's iPod, iPhone, sowie Ipad nutzen ein proprietäres Übertragungsprotokoll.[20]

Picture Transfer Protocol - Das PTP ist die Vorgängerversion des moderneren Media Transfer Protocols. MTP basiert auf PTP, jedoch bietet dieses mehr Features bzw. Erweiterungen. Das Picture Transfer Protocol wird hauptsächlich für Digitalkameras eingesetzt. Dies bedeutet dass jede Software, welche das Übertragen von Bildern von Digitalkameras unterstützt, auch die Übertragung über ein Android-Gerät akzeptiert, sofern der Modus auf PTP gestellt ist. Dieses Protokoll wird von Apple's Mac OS X unterstützt und kann somit über eine USB-Verbindung Bilder zwischen den beiden Systemen übertragen.[20]

USB-Mass-Storage - Bei älteren Android-Versionen ist man möglicherweise gezwungen eine Verbindung über den USB-Mass-Storage herzustellen, sofern MTP und PTP nicht zur Verfügung stehen. Sofern das Gerät eine extra SD-Karte als Speichermedium besitzt, kann diese entfernt werden und direkt am PC ausgewertet werden.[20]

3.4. App-Stores und deren Sicherheit

Für Smartphones haben sich Google‘s Betriebssystem Android, sowie das von Apple entwickelte IOS weltweit durchgesetzt. Jedoch unterscheiden sich diese Systeme wesentlich bei der Bereitstellung von Apps für das jeweilige System. Bei IOS ist die Installation von Apps nur über den offiziellen „App Store“ möglich und nur unter speziellen Umständen über andere Quellen beziehbar. Man spricht in diesem Fall von Jailbraken[5]. Wogegen die Installation von Apps bei Android auch von nicht offiziellen - sogenannten Drittanbieter (engl. Third-Party) Quellen - durchaus möglich ist. Hierbei ist aus sicherheitstechnischen Gründen auch großes Bedenken, da es keine *trusted applications* sind. Jeder Benutzer oder Benutzerin eines Android Smartphones hat die Möglichkeit eine APK-Datei[6] von fremden – also Apps die nicht von Google‘s offiziellen Play-Store bezogen wurden – zu installieren, die an dieser Stelle Sicherheitsrisiken mitbringen können.[21] Kopierte Apps – auch Cloned-App genannt – werden in der Regel von Drittanbietern online gestellt. Diese sind für Benutzer, speziell für jene die im asiatischen Raum wie beispielsweise China, Russland aber auch in Europa leben, eine ernstzunehmende Bedrohung. Google hat entsprechende Maßnahmen implementiert, um solche Clones aus deren Play-Store ausschließen zu können.[21] Eine der ersten zwingenden Sicherheitsmaßnahmen, welche von Google implementiert wurde, ist die Authentizität jeder App, welche bis zum App-Entwickler oder Entwicklerin nachvollziehbar sein muss.[22] Um eine App über den Play-Store zum Download bereitstellen zu können, ist ein Benutzerkonto bei Google notwendig. Sowie die Bezahlung einer Gebühr, um als App-Entwickler oder Entwicklerin bei Google aufzuscheinen. Dies ist Voraussetzung, um in weiterer Folge die entwickelte App mit dem speziellen Signaturschlüssel des Kontos zu signieren, um die Echtheit zu bestätigen. Google hat dafür eine eigenes System im Einsatz, welches den Paketnamen der App, den Herausgeber sowie weitere App-Komponenten prüft.[21] Daher müssen alle Apps mit einem kryptografischen Schlüssel signiert sein. Die Zertifikate sind im APK-Format enthalten, um dem Endbenutzer oder Endbenutzerin das verifizieren eines APK-Pakets zu ermöglichen.[22] Aufgrund von Richtlinien, die in einigen Staaten eingeführt wurden, ist der Zugriff auf den Play-Store teilweise sehr beschränkt. Im speziellen ist dies in China und Russland der Fall. Daher sind auch gerade in diesen Regionen Anbieter von eigenen App-

[5] Das Entfernen von Nutzungsbeschränkungen am System

[6] Installationsdatei einer Android-App

Stores zu erkennen.[21]

Folgend eine kurze Auflistung von Drittanbietern, welche Apps für Android-Geräte zur Verfügung stellen[22]:

- AppChina[7] – Dieser App-Store ist einer der größten Alternativen zum offizielen Google Play-Store.
- Anzhi[8] – Dieser App-Store ist in China präsent und ist hauptsächlich für chinesische Android-User gedacht. Dieser speichert und verteilt Apps, welche in chinesischer Sprache bereitstehen. Des Weiteren sind hier weniger strengere Sicherheitsrichtlinien implementiert als bei Google Play.
- Slideme[9] – Slideme ist ein amerikanischer Anbieter und ist ein direkter Konkurrent des Google Play-Store. Angeboten werden kostenlose, sowie kostenpflichtige Apps.
- FreewareLovers[10] – Ein App-Markt betrieben von einem deutschen Unternehmen. Es werden kostenlose Apps für alle gängigen mobilen Plattformen angeboten, inkludiert ist auch Android. Um Apps von diesem Store zu beziehen, ist keine spezielle Applikation notwendig, lediglich ein Webbrowser wird benötigt.
- ProAndroid[11] – Operiert in Russland. ProAndroid ist ein sehr kleiner App-Store und verteilt kostenlose Apps.
- F-Droid[12] – Bei F-Droid wird ein Archiv an freien und Open-Source Software angeboten, sowie ein Android-Client zur Durchführung von Installationen und Updates, Rezensionen und andere Funktionen rund um Android und Softwarefreiheit.
- 1mobile[13] – Dieser App-Store bietet freie Apps für Android an. Im Sortiment sind mehrer tausend Apps verfügbar.

3.5. Android Application Package - APK

Apps für Android werden als APK-Datei verteilt. Das *Android Application Package* ist ein Archivformat zur Verteilung von Applikationen für Android basierende Endgeräte. Das Format ist ein verbreitetes

[7] http://www.appchina.com
[8] http://www.anzhi.com
[9] www.slideme.org
[10] http://www.freewarelovers.com
[11] http://proandroid.net
[12] http://f-droid.org
[13] market.1mobile.com

ZIP-Dateiformat und beinhaltet sämtliche Informationen bzw. Dateien die notwendig sind, um eine Installation am Endgerät zu ermöglichen. Der Vorteil dieses Formats ist die Handhabung mit einem einzigen File, welches auf das Smartphone übertragen werden muss.[3] Eine interessante Bemerkung ist, dass alle Dateien die für eine App notwendig sind, vom Entwickler oder Entwicklerin bis zum Endbenutzer oder Endbenutzerin nicht mehr modifiziert werden, wie beispielsweise das letzte Änderungsdatum. Der gesamte Bytecode, also derjenige Code, welcher die App repräsentiert, ist in einer Datei *classes.dex* enthalten. Dies bedeutet, dass das letzte Änderungsdatum zugleich dem *packaging date*, also jenem Datum, an dem die APK-Datei erstellt wurde, ist.[22]

Folgende Struktur weist typischerweise ein APK-Archiv auf[23]:

- META-INF: Dieses Verzeichnis beinhaltet Metainformationen über das Paket selbst, sowie weiter Dateien:
  - Manifest.mf – Diese Datei beinhaltet eine Auflistung über alle Ressourcen und deren SHA-1 Übersicht.
  - Cert.Rsa/Dsa – Das Zertifikat der Applikation bzw. auch erkennbar als das Zertifikat des Entwicklers oder Entwicklerin
  - Cert.sf – Die Liste der verwendeten Ressourcen und SHA-1 Überblick der entsprechenden Zeilen in der Manifest.mf Datei.
  - lib – Dieses Verzeichnis beinhaltet kompilierte Softwareteile, welche für spezielle Teile der Applikation bzw. Prozess benötigt werden.
  - res - Dieses Verzeichnis beinhaltet alle Ressourcen, welche nicht in der Datei resources.arsc kompiliert sind.
  - assets – In diesem Verzeichnis werden alle Dateien aufbewahrt (Bilder, Text-Dateien usw.), die von der Applikation verwendet werden. Diese Dateien können mittels AssetManager abgerufen werden.
  - AndroidManifest.xml – Ein weiteres Android Manifest-File welches den Namen, Version, Zugriffsrechte, sowie Bibliotheken auf denen referenziert wird, beinhaltet. Diese Datei liegt in binärer Form vor und kann mittels XML-Konverter in für Menschen lesbare Form konvertiert werden.

Eine Prüfung auf die Echtheit eines APK-Archivs ist relativ einfach. Listing 3.1 zeigt wie mittels dem Kommandozeilentool *hexdump* am Beispiel *com.whatsapp.apk* folgende Bytes betrachtet werden kön-

nen. Für die Ausführung von *hexdump* sind keine zusätzlichen Benutzerrechte notwendig.[3]

```
$hexdump -C -n 4 com.whatsapp.apk
00000000  50 4b 03 04                                       |PK..|
00000004
```

Listing 3.1: Ausgabe hexdump am Beispiel com.whatsapp.com

Die ersten beiden Bytes - 50 4b – stellen die Zeichen PK da, welche die Initialen des Erfinder Phil Katz darstellen. Darauf folgenden sind die Bytes – 03 04 – zu erkennen. Diese vier Bytes stellen somit ein APK-Format da und kann auf einfachste Weise entpackt werden, umso mehr über den Inhalt zu erfahren.[3]

3.6. Build Process

Da der Build Process – jener Prozess, der notwendig ist um aus dem reinen Source-Code eine vollständige APK-Datei generieren zu können – ein sehr umfangreicher ist, ist es gut einen Überblick darüber zu haben.[19] Abbildung 3.1 zeigt den typischen Prozessablauf von der Erstellung einer Android Applikation.

1. Der Compiler konvertiert den erstellten Source-Code in das ausführbare Format DEX (Dalvik Executable). Dieses beinhaltet den Bytecode, welcher in weiterer Folge am Android Endgerät ausgeführt werden kann, sowie alle anderen benötigten Ressourcen die von der App gebraucht werden.

2. Der APK-Packager fügt in weiterer Folge die erstelle DEX-Datei und kompilierten Ressourcen in eine APK-Datei zusammen.

3. Danach wird die soeben erstellte APK-Datei vom APK-Packager signiert. Hier wird unterschieden zwischen zwei Möglichkeiten, ob die App für Debugging Zwecke - somit nur für Testzwecke - erstellt wird oder veröfftentlich wird. Dies hat zur Folge, dass die App mittels „Debug Keystore“ signiert wird. Wenn die App jedoch für die Veröffentlichung erstellt wird benötigt der APK-Packager den „Release Keystore“.

4. Bevor die finale APK-Datei erstellt wird, nutzt der APK-Packager ein Tool (zipalign), um eine Optimierung der App durchzuführen, umso nur den benötigten Speicher am System zu nutzen, damit dadurch das Endgerät optimal ausgenutzt werden kann.

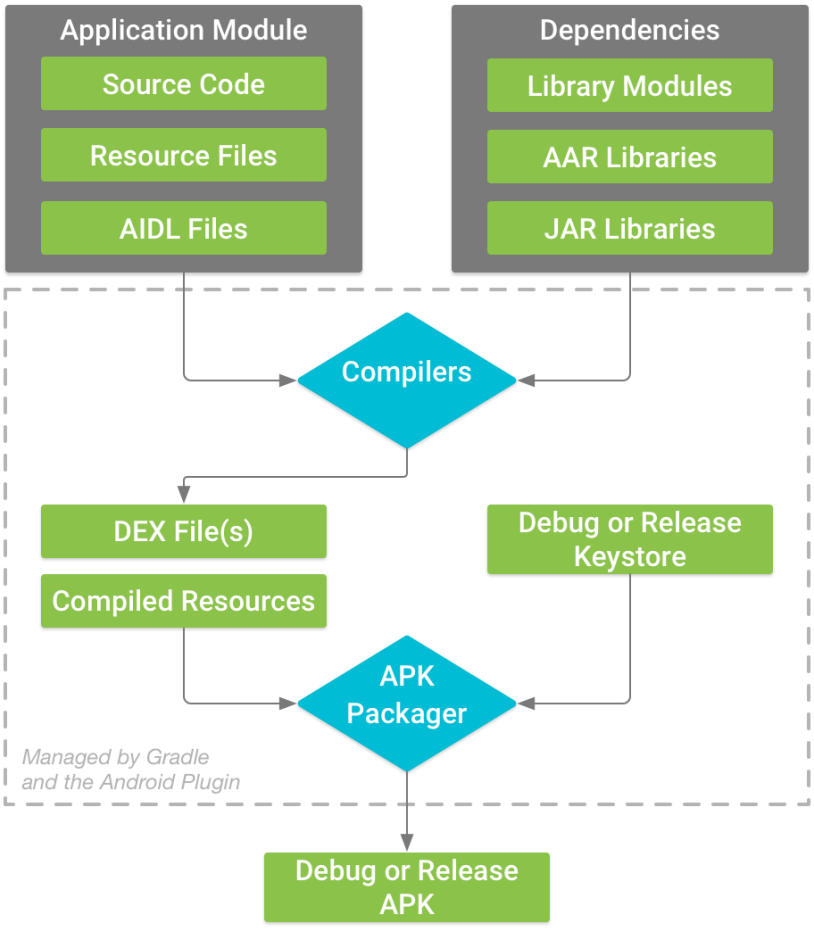

Abbildung 3.1.: Build Process - Android App Quelle: android.com

Daher ergibt sich am Ende des „Build process“ eine „Debug APK“ oder eine „Release APK“ die getestet, veröffentlicht oder für andere User in App-Stores verteilt werden kann.[19]
Seit der Android Version 4.4 bietet Google ein weiteres Format für die Erstellung von Apps an, das *Android App Bundle* Format bietet einige Vorteile gegenüber der alt-bekannten Methode Apps zu erstellen. Unter anderem sind dies *Dynamic Delivery, automatic Multi-APK Verteilung, dynamic feature modules*.[19]

3.6.1. Android Bundle

Das Android App-Bundle (.aab) ist ein neues Uploadformat von Google, welches sämtlichen kompilierten Sourcecode und Ressourcen beinhaltet. Das Generieren, sowie das Signieren der App wird jedoch von Google Play übernommen. Dies bedeutet, dass der Entwickler oder die Entwicklerin nicht mehr APK-Dateien selbst kompiliert, erstellt und managed um Mulit-APK’s - eine App für mehrere Android Versionen zu Verfügung stellt - sondern dies bei Google Play auslagert. Ein Android-Bundle kann daher

nicht direkt auf einem Smartphone installiert werden. Möchte man daher schnell eine App auf einem Gerät testen, muss zum üblichen APK-Build Process zurück gegriffen werden.
Das Android App-Bundle hat den Vorteil, dass eine APK-Datei aus mehreren Einzelteilen individuell zusammengebaut wird, abhängig vom jeweiligen Smartphone und Android-Version. Dadurch fällt die Dateigröße einer APK-Datei geringer aus, da nur mehr die Ressourcen am Smartphone sind, welche wirklich benötigt werden. Für die Erstellung eines Android App-Bundle kann die Entwicklungsumgebung *Android Studio* verwendet werden bzw. steht ein Kommandozeilen-Tool (*bundletool*) zur Verfügung.[19]

3.6.2. Dynamic Delivery

Android's neuestes Zustellungsservice für Apps namens *Dynamic Delivery* nutzt *Android App Bundles* um optimierte APK-Dateien zu generieren und diese für alle Smartphones bereitzustellen. Dies bedeutet, dass nur mehr der notwendige Source-Code und Ressourcen heruntergeladen werden, die benötigt werden. Somit bekommt der Benutzer oder die Benutzerin eine optimale APK-Datei in einer minimalen Dateigröße.
Ein erheblicher Vorteil von *Dynamic Delivery* ist die Möglichkeit gewisse Komponenten in Modulen zu betrachten. Damit ist es möglich bei der Installation einer App gewisse Komponenten zur Installationszeit herauszunehmen. Bei der Entwicklung einer App muss dafür die Unterstützung für *dynamic feature modules* im *Android App Bundle* aktiviert sein. Durch *Dynamic Delivery* ist es dem Benutzer oder der Benutzerin möglich, benötigte dynamische Feature erst dann herunterzuladen, wenn diese zum Zeitpunkt (On-Demand) benötigt werden. [19]

3.6.3. Dynamic Delivery mit Split-APKs

Split-APK's sind regulären APK-Datein sehr ähnlich. Diese inkludieren kompilierten Bytecode (DEX-Datei) und eine Android Manifest-Datei. Die Android-Plattform ist jedoch in der Lage, mehrere installierte Split-APKs als eine einzige App zu behandeln. Der Vorteil von geteilten APK-Dateien ist die Möglichkeit, ein monolithisches APK in kleinere, diskrete Pakete aufzuteilen, die bei Bedarf auf dem Gerät installiert werden.[24] Die folgende Auflistung zeigt die unterschiedlichen Typen von Split-APKs:

- **Base APK -** In dieser APK-Datei sind alle notwendigen Ressourcen und Code-Dateien enthalten, die in jedem Fall benötigt werden. Wenn die App auf das Smartphone geladen wird ist immer die Base APK-Datei dabei, unabhängig davaon welche Android-Version und Hardware eingesetzt wird.[24]

- **Configuration APK -** Solch eine APK-Datei beinhaltet ausschließlich nur native Bibliotheken die

für ein Gerät benötigt werden wie z.B. Bildschirmgröße, Auflösung, Sprache oder CPU-Architektur. Somit werden diese Bibliotheken erst dann runtergeladen, wenn zuvor das entsprechende Base-APK bezogen wurde.[24]

- **Dynamic Feature APK -** Jedes dieser APK-Datei enthält Source-Code und Ressourcen, die bei der Erstinstallation einer App nicht erforderlich sind, aber später heruntergeladen und installiert werden können.[24]

Abbildung 3.2 zeigt nochmals veranschaulicht die Abhängigkeiten zwischen einzelnen APK-Dateien wenn Split-APKs zum Einsatz kommen.

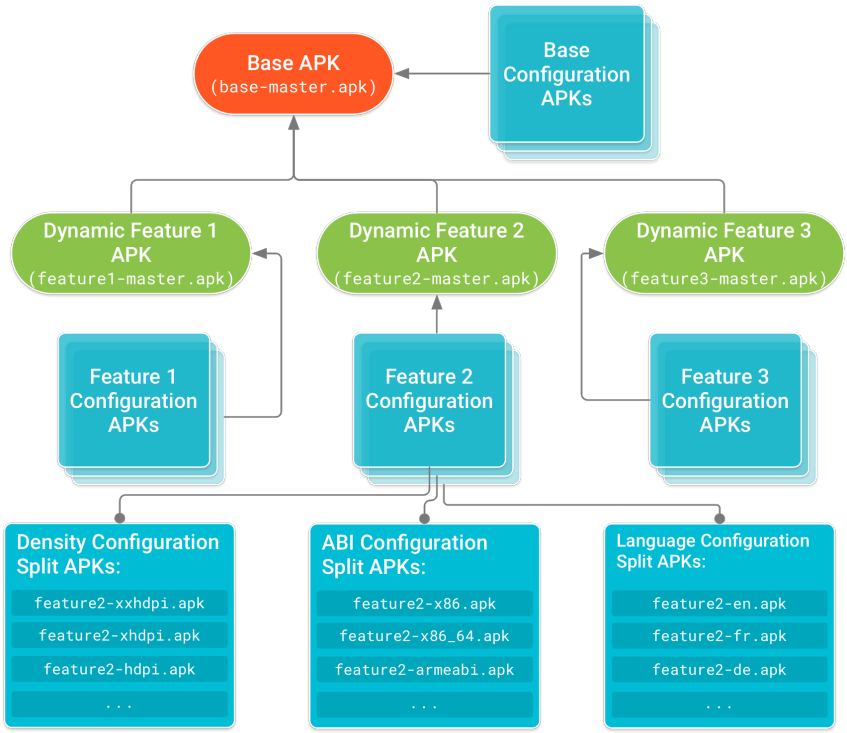

Abbildung 3.2.: Abhängigkeitsbaum für eine App, die mit Split-APKs bedient wird. Quelle:android.com

Da Geräte mit einer Version unter 4.4 (API-Level[14] 19) Split-APKs nicht unterstützen, stellt Google Play stattdessen Multi-APKs zur Verfügung. Dies bedeutet, dass eine vollständige App, speziell für einen Hardwaretyp optimiert ist, keine unnötigen Code oder Ressourcen wie beispielsweise Bildschirmeinstellungen oder spezielle Hardwarearchitekuren von anderen Herstellern beinhaltet. Jedoch enthalten diese Multi-APKs Ressourcen für Sprachen, die von der App unterstützte werden. Somit kann der Benutzer oder die Benutzerin die Spracheinstellungen ändern, ohne eine andere Multi-APK Datei herunterzuladen.

[14] Application Programming Interface - Gibt den Entwicklungsstand der verfügbaren Funktionen an.

3.7. App-Signatur

Grundsätzlich ist es notwendig jede App zu signieren, da Android die Installation am Smartphone ansonsten verweigert. Dies gilt für die Veröffentlichung über den Google Play-Store, sowie bei der direkten Installation einer APK-Datei am Smartphone. Für das signieren sind einige wichtige Komponenten notwendig:[19]

- Public-Key Zertifikat – Das Zertifikat beinhaltet den Nachweis des Entwicklers oder Entwicklerin, sowie den öffentlichen Schlüssel des Public/Private Schlüsselpaars. Des Weiteren beinhaltet es noch einige Metadaten, die den Besitzer des Zertifikats identifizieren. Beispielsweise Name, Herkunft usw.. Dies ist jene Person die den privaten Schlüssel auch besitzt. Das Public-Key Zertifikat wird mit dem privaten Schlüssel signiert.
- Upload-Key – Der Upload-Key ist notwendig um die signierte App am Google Play-Store zur Verfügung zu stellen. Dieser wird rein für das Uploaden der App verwendet und nicht um die App zu signieren. Dies wurde von Google aus sicherheitstechnischen Gründen implementiert, da bei Verlust des App-Signing Key weiterhin die Möglichkeit besteht eine App mit Updates zu versorgen. Sollte jedoch der Upload-Key verloren gehen bzw. gestohlen werden, kann dieser Upload-Key widerrufen (engl. revoked) werden. Dafür gibt es von Google einen eigens definierten Prozess, der über ein spzielles Onlineformular ausgelöst werden kann. Der Upload-Key ist jedoch nur notwendig, wenn Apps über den Google Play Store veröffentlicht werden. Möchte man eine App über andere Stores bereitstellen bzw. Apps direkt auf Endgeräten installieren, wird dieser nicht benötigt.
- Keystore – Der Keystore entspricht einer Datei welche verschlüsselt ist. Diese beinhaltet alle Informationen über den Entwickler oder Entwicklerin. Das Keystore-File kann nicht nur für das signieren einer App verwendet werden, sondern auch für die Authentifizierung. Somit ist der Keystore einer der schützenswertesten Dateien für die Entwicklung und Veröffentlichung von Apps. Sollte jenes System, auf dem sich der Keystore befindet, kompromittiert werden und die Datei in falsche Hände geraten, können keine weiteren Updates für betroffene Apps veröffentlicht werden.

3.8. Erkennen von manipulierten Apps im laufendem Betrieb.

Das Erkennen von Schadsoftware bzw. manipulierten Apps im laufenden Betrieb ist nicht einfach. Der Unterschied zwischen einer originalen App und einer gefälschten kann kaum bzw. nur mit erhöhtem Aufwand geklärt werden. Jedoch kann der Verdacht, eine solche schadhafte Software am Smartphone

zu haben, dadurch erkannt werden, dass jegliche Aktionen mit großer Verzögerung durchgeführt werden. Grund dafür ist, dass diese schadhaften Apps einen großen Teil der verfügbaren Systemressourcen, beispielsweise für das schürfen von Kryptowährungen, missbrauchen. Dies kann dazu führen, dass am Smartphone keine Eingaben mehr getätigt werden können. Sogar der Neustart des Geräts ist in einigen Fällen keine Lösung. Ein weiteres Merkmal ist die Festellung von erhöhtem Akku-Verbrauch. Dies ist nur eine Schlussfolgerung auf hohe Aktivität es Geräts, jedoch keine forensische Analysetechnik.[21]

4. Mobile Forensik - Datensicherung und Analyse

Wenn eine forensiche Analyse durchgeführt wird, müssen sämtliche Daten auf einem Gerät unverändert ausgewertet werden und dürfen nicht verändert werden, da ansonsten wesentliche Informationen zerstört werden könnten und die Einbringung der Beiweismittel beispielsweise in einem Gerichtsverfahren nicht gültig sind. Grundsätzlich lässt sich die digitale Forensik in Computer-Forensik, Mobile-Forensik, Memory-Forensik, Network-Forensik, Malware-Forensik, OS-Forensik unterteilen. Nichtsdestotrotz, ist der Prozess von der Sammulng der Daten bis zur Berichterstellung sehr generisch und daher in allen Bereichen anzuwenden.
Nachdem die Entscheidung für eine forensische Untersuchung getroffen wurde, kommen vier wichtige Teilprozesse zum Einsatz.

- Sammeln der Daten
- Prüfen der Daten
- Analyse der Daten
- Berichterstellung

In Abbildung 4.1 wird der Prozessablauf nochmals grafisch dargestellt. Zu erkennen ist, dass die Abfolge der einzelnen Teilprozesse auf einander abgestimmt sind und der jeweils zuvor liegende Teilprozess als Input für den nächsten dient.[25]

Sammeln der Daten

Im ersten Schritt werden sämtliche Daten gesammelt und ein entsprechendes Image[1] erstellt. Hierbei ist es wichtig entsprechende Sicherheitsmaßnahmen zu treffen, um gewährleisten zu können, dass die Beweismittelkette (engl. Chain of Custody) für weitere Verarbeitung nicht verändert wurde. Eine gängige Methode ist die Erstellung einer Prüfsumme (Hashwert) des Image. Dieser Vorgang wird jedes mal

[1] Ein Abbild aller gesammelten Daten

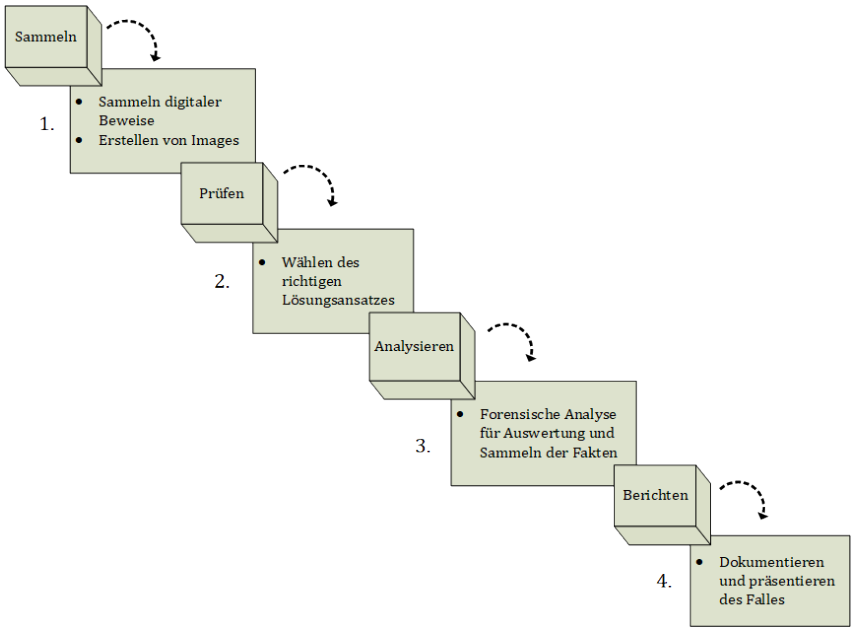

Abbildung 4.1.: Prozessablauf in der digitalen Forensik

bei der Verwendung bzw. Weitergabe des Beweismittel durchgeführt und geprüft, um die Integrität der Daten feststellen zu können.

Prüfen der Daten

Bei der Prüfung der Daten versteht man die Entscheidung der eingesetzten Analysetools. Abhängig der gesammelten Daten und Zweck der forensischen Untersuchung, arbeiten Tools unterschiedlich. Daher ist der Teilprozess ein sehr wesentlicher, da ansonsten das Ergebnis verfälscht werden könnte.

Analyse der Daten

Hierbei wird die eigentliche Analyse der gesammelten Daten durchgeführt. Diese wird unter Berücksichtigung der Anforderung seitens des Auftraggebers vollzogen.

Berichterstellung

Nachdem die Analyse bzw. Auswertung der Daten beendet ist, müssen diese in eine entsprechende Form gebracht werden. Wichtige Funde werden dokumentiert.

4.1. Prozessablauf der Datensicherung

Bevor eine forensiche Analyse durchgeführt werden kann, müssen die entsprechenden Daten gesichert werden. Dafür wird ein Backup über die ADB-Schnittstelle erstellt. Dieses Backup beinhaltet sämtliche Apps, die am Gerät installiert sind, sobald von einem App-Store, oder anderer Quelle, APK-Dateien bezogen und installiert wurden.
Im konkreten Fall wird der Prozess der Datensicherung - angepasst an diese Arbeit - in Abbildung 4.2 grafisch dargestellt.

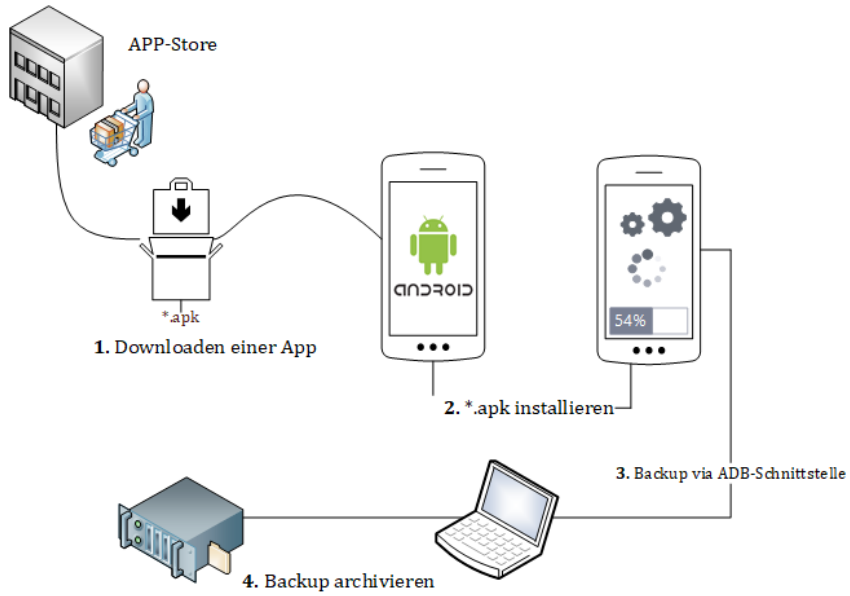

Abbildung 4.2.: Prozessablauf für die Erstellung eines Backups

Überlicherweise wird eine App von einer gewählten Quelle bezogen und am Smartphone gespeichert und schließlich installiert. Nach der Installation aller benötigten Apps wird danach mittels ADB-Verbindung eine Datensicherung durchgeführt. Um die gesicherten Daten über einen längeren Zeitraum zu sichern - da eventuell Systemvergleiche über mehrere gesicherte Systemzustände gewünscht sind - empfiehlt es sich auf einem entsprechenden großen Speichermedium zu archivieren.
In Listing 4.1 wird gezeigt wie das Smartphone mit dem ADB-Tool angesprochen wird und wie ein Backup erstellt wird. Mit dem Kommando *adb devices* kann ermittelt werden ob das Smartphone am PC erkannt wurde. Sollte es hier nicht gelistet sein, kann auch keine Verbindung für die Erstellung einer Datensicherung aufgebaut werden. Zeile vier in Listing 4.1 führt die eigentliche Erstellung des Backups durch. Die Parameter *backup -all -shared* sind für ein vollständiges Backup - Systemdaten und App-Daten - sowie sämtliche Daten die auf externen Speichermedien vorliegen. Der Parameter *-f* gibt lediglich das Zielverzeichnis für das erstellte Backup an.

```
root@l440:~# adb devices
List of devices attached
33003cf6be2da271    device
root@l440:~# adb backup -all -shared -f /home/thomas/backup_samsung/backup
Now unlock your device and confirm the backup operation...
root@l440:~#
```

Listing 4.1: Verbindung via ADB-Schnittstelle - Backup starten

Das Ergebnis der Datensicherung über die ADB-Schnittstelle ist eine Binär-Datei (*.ab-Format), welche eine Verzeichnisstruktur aller gesicherten Apps beinhaltet. Um die Integrität der gesicherten Daten gewährleisten zu können, wird mittels Hash-Funktion eine Prüfsumme darüber gebildet. Dies hat den Vorteil, dass durch die Weitergabe der Daten an andere Personen, sowie beim einlesen in andere Systeme für weitere Analysen, die Echtheit bzw. Manipulationen festgestellt werden kann. Um an die eigentlichen Daten sämtlicher Apps zu kommen, muss die Binär-Datei in ein lesbares Format gebracht werden, dazu eignet sich das TAR-Format[2]. Für diesen Schritt wird das Tool *Android Backup Extractor*[3] verwendet. Listing 4.2 zeigt den Auszug auf der Kommandozeile für die Erstellung einer Prüfsumme mittels SHA1-Algorithmus, sowie die Konvertierung der Binär-Datei in das TAR-Format.

```
root@l440:/home/thomas/backup_samsung# sha1sum backup.ab
42b515790580aa0d10e509ccf968bf5d50cfb998  backup.ab

root@l440:/media/thomas/encryptdisk/Diplomarbeit/android-backup-extractor
    -20180203-bin# java -jar abe.jar unpack backup.ab backup.tar
Backup encrypted, enter password (will NOT be displayed):
Password:
```

Listing 4.2: Binär-Datei Hashing, sowie konvertieren in TAR-Format

Bei der Erstellung des Backups über die ADB-Schnittstelle besteht die Möglichkeit ein Passwort zu setzen und so die Datensicherung zu verschlüsseln. Daher wird bei der Ausführung des *Android Backup Extractor* je nach Situation ein Passwort verlangt, um das Backup zu entschlüsseln. Nachdem der Vorgang abgeschlossen ist, ist im jeweiligen Verzeichnis das Backup als Binär-Datei, sowie im TAR-Format für eine weitere Verarbeitung zur Verfügung. Zu erkennen ist, dass die Binär-Datei weniger Speicherplatz benötigt als in extrahierter Version. Listing 4.3 zeigt beide Dateien am Dateisystem.

[2] TAR - tape archiver. Format um Daten zu komprimieren

[3] https://github.com/nelenkov/android-backup-extractor

```
443M -rw-r----- 1 thomas thomas 443M Feb  6  2018 backup.ab
617M -rw-r--r-- 1 thomas thomas 617M Apr  3 15:33 backup.tar
```

Listing 4.3: Backup als Binär-Datei und TAR-Format im Größenvergleich

Eine weitere Möglichkeit um gezielt APK-Datein von Apps über die ADB-Schnittstelle zu sichern, kann wie in Listing 4.4 gezeigt, erreicht werden. Gründe dafür sind beispielweise nur gezielte Apps zu überwachen. Sollte der Bedarf bestehen explizit gewünschte Apps auf Manipulationen zu untersuchen, zeigt sich diese Variante als empfehlenswert.

```
root@l440:/home/thomas# adb shell pm list packages
# Auflistung aller installierten Pakete
# Ausgabe wurde gekuertzt
package:com.sec.android.app.SecSetupWizard
package:com.microsoft.office.onenote
package:at.willhaben
package:com.android.statementservice

# Ausgabe des Pfades am System des gewuenschten Pakets
root@l440:/home/thomas# adb shell pm path at.willhaben
package:/data/app/at.willhaben-1/base.apk

# Sichern der Base-APK am PC
root@l440:/home/thomas# adb pull /data/app/at.willhaben-1/base.apk .
[100%] /data/app/at.willhaben-1/base.apk
```

Listing 4.4: Gezielte Auswahl von APK-Dateien

Um jedoch automatisiert sämtliche Third-Party Apps zu sichern, kann dies mit folgendem Skript durchgeführt werden. Hierbei wird nach den installierten Apps gesucht, die APK-Dateien gesichert und zusätzlich werden diese auf den Paketnamen der App umbenannt, da ansonsten alle gesicherten APK-Dateien base.apk lauten und dadurch mögliche Verwechslungen auftreten könnten. Listing 4.5 zeigt das zuvor beschriebene Shell-Skript.

```
for package in $(adb shell pm list packages -3 | tr -d '\r' | sed 's/
    package://g'); do apk=$(adb shell pm path $package | tr -d '\r' | sed 's
    /package://g'); echo "Pulling $apk"; adb pull -p $apk "$package".apk;
    done
```

Listing 4.5: Sichern aller Third-Party Apps

4.2. Forensische Analyse am Smartphone

Nachdem eine Datensicherung über die ADB-Schnittstelle durchgeführt wurde, muss das System selbst etwas näher betrachtet werden. Dabei wird der Fokus auf die Struktur des Dateisystems gelegt, um zu analysieren, wo Apps nach dem kompilieren am System installiert werden. Da ab der Version 5.0 von Android auf die *Android Runtime* gesetzt wird, werden Apps bei der Installation einmalig einer Optimierung des Programmcodes unterzogen und folglich zu Bytecode bzw. Maschinencode kompiliert. Dieser Vorgang wird lediglich einmal bei der Installation durchgeführt. Dadurch ergeben sich einige Vorteile, wie beispielsweise verbesserte Performance, geringere Speicherauslastung.[19]

4.2.1. Strukturierung des Dateisystems von Android

Die Dateistruktur des Betriebssystems ist eine modifizierte Version des traditionellen Linux-Dateisystems. Es gibt jedoch, abhängig von Smartphone-Herstellern, einige Abweichungen. Listing 4.6 zeigt einen kurzen Überblick über die Partitionierung, die für weitere Untersuchungen relevant ist.[26]

```
shell@s5neolte:/ $ df
Filesystem                Size     Used     Free   Blksize
/dev                    926.9M   124.0K   926.8M      4.0K
/sys/fs/cgroup          926.9M    12.0K   926.9M      4.0K
/mnt                    926.9M     0.0K   926.9M      4.0K
/mnt/secure             926.9M     0.0K   926.9M      4.0K
/system                   2.9G     2.7G   129.7M      4.0K
/efs                     15.7M     2.3M    13.4M      4.0K
/cache                  192.8M     1.4M   191.5M      4.0K
/data                    11.1G     9.2G     1.8G      4.0K
/persdata/absolute        7.9M    88.0K     7.8M      4.0K
/cpefs                    3.9M   548.0K     3.3M      4.0K
/storage                926.9M     0.0K   926.9M      4.0K
```

```
/mnt/knox                   11.1G     9.2G     1.8G      4.0K
/mnt/shell/enc_media        11.0G     9.2G     1.8G      4.0K
/storage/emulated           11.0G     9.2G     1.8G      4.0K
/data/enc_user              11.1G     9.2G     1.8G      4.0K
/mnt/shell/enc_emulated     11.0G     9.2G     1.8G      4.0K
/storage/6337-3633           7.4G   372.0M     7.0G     32.0K
```

Listing 4.6: Auflistung aller Patitionen am Smartphone

Für Apps am Smartphone sind die Partitionen */system, /cache, /data, /storage* relevant. Diese können jeweils Apps, in APK-Format, oder jeweils Teile von Apps beinhalten. Abhängig von der Laufzeitumgebung am Gerät befinden sich Dateien, bezogen auf Apps, auf unterschiedlichen Orten. Daher muss vor einer Analyse zuerst festgestellt werden welche Laufzeitumgebung verwendet wird. Welche Laufzeitumgebungen Android einsetzt und worin diese sich grundsätzlich Unterscheiden wurde bereits in Kapitel 3.1.1 und 3.1.2 erklärt.
Um zu verifizieren welche Laufzeitumgebung am Smartphone aktiv ist, zeigt Listing 4.7, wie dies mittels ADB-Schnittstelle verifiziert werden kann.

```
shell@s5neolte:/ $ getprop
## Ausgabe wurde gekuertzt
[persist.sys.dalvik.vm.lib.2]: [libart.so]
```

Listing 4.7: *getprop* zum Auslesen von Systemparametern

Mit dem Kommandozeilentool *getprop* können sämtliche Informationen über das System ausgelesen werden. Zu sehen ist, dass als Rückgabewert *libart.so* zurückgegeben wurde. Dies bedeutet, dass als Laufzeitumgebung *ART* verwendet wird. Einige Distributionen bieten die Möglichkeit zwischen *ART* und *Dalvik VM* zu wählen. Daher ist es hilfreich mittels *getprop* schnell verifizieren zu können, welche Laufzeitumgebung tatsächlich aktiv ist. Wäre auf diesem Smartphone *Dalvik VM* aktiv, würde als Rückgabewert *libdvm.so* erscheinen, jedoch lässt dieses Smartphone mit dem originalen Herstellerimage am Gerät keine Wahlmöglichkeit zu. Um dies zu erreichen, muss dafür ein *Custom ROM*[4] installiert werden bzw. eine kompatible *Android Open Source Projekt AOSP* Lösung verwendet werden. Systemimages von Herstellern werden *Stock ROM* bezeichnet.

[4] Alternative Firmware, welche nicht vom Hersteller entwickelt wurden.

4.2.2. Benutzerrechte und Einschränkungen

Smartphones mit einem *Stock ROM* sind in der Regel sehr beschränkt, wenn es um Benutzerrechte geht. Dies hat zur Folge, dass der Standardbenutzer ausschließlich auf Verzeichnisse zugreifen kann, welche nicht unmittelbar mit dem System bzw. Konfigurationsdateien in Relation stehen. Folglich bedeutet dies auch, dass auf sämtliche Dateien von installierten Apps nicht zugegriffen werden kann. Listing 4.8 zeigt sämtliche Gruppenzugehörigkeiten am System für den Benutzer Shell.

```
shell@s5neolte:/ $ id
uid=2000(shell) gid=2000(shell) groups=2000(shell),1004(input),1007(log)
   ,1011(adb),1015(sdcard_rw),1028(sdcard_r),3001(net_bt_admin),3002(net_bt
   ),3003(inet),3006(net_bw_stats) context=u:r:shell:s0
```

Listing 4.8: Benutzerrechte des Benutzers shell

Zu erkennen ist, dass der Benutzer Shell eine User-ID und Group-ID 2000 besitzt. Die gelisteten Gruppen berechtigen den Benutzer Lese-, Schreib-, Ausführ- Rechte zu erlangen und diese dort auszuführen. Ein weiteres Merkmal ist das *$*-Symbol. Dies lässt darauf schließen, dass der Benutzer keine Root-Berechtigung besitzt. Sollten diese vorhanden sein, würde statt dem *$*-Symbol ein *#*-Symbol (Raute) angezeigt werden. Benutzer mit der User-ID bzw. Group-ID 0 (Null) haben Root-Rechte und haben daher in allen Verzeichnissen Lese-, Schreib- und Ausführ-Rechte.

Root-Berechtigung

Da es für die forensische Analyse am Dateisystem notwendig ist, tiefere Einblicke auf die Strukturen zu haben, ist es unabdingbar Root-Berechtigungen am Smartphone zu erlangen. Dies wurde mit einem entsprechenden *Custom ROM* erreicht. Da der Fokus in diesem Kapitel auf der forensische Analyse liegt, wird auf die Durchführung zur Erlangung von Root-Berechtigungen nicht näher eingegangen. Als Beweis für die erfolgreiche Implementierung des *Custom ROM* mit Root-Berechtigungen, wird in Listing 4.9 gezeigt. Zu sehen ist, dass der Benutzer *shell* auf den Benutzer *root* gewechselt wird und die User-ID 0 besitzt.

```
shell@s5neolte:/ $ su
root@s5neolte:/ # id
uid=0(root) gid=0(root) groups=0(root) context=u:r:init:s0
root@s5neolte:/ #
```

Listing 4.9: Benutzerwechsel von shell auf root

4.2.3. Apps im /data/app/ Verzeichnis

Im Verzeichnis */data/app* befinden sich alle installierten Apps. Mit dem Tool *dex2oat* werden Apps in das Dateiformat *Android Optimized Application File OAT* kompiliert. Dieses Tool nimmt DEX-Dateien als Input und generiert daraus eine ausführbare Datei für das Zielsystem.[27] Da nun entpsrechende Berechtigungen vorhanden sind, kann auf dieses Verzeichnis zugegriffen werden. Der Standardbenutzer *shell* hätte nicht ausreichend Rechte diese Daten zu sichten.
Im Verzeichnis befindet sich für jede installierte App ein weiteres Verzeichnis, mit der folgenden Ordnerstrukur. Listing 4.10 zeigt am Beispiel at.willhaben-1 wie diese aufgebaut ist. Zu erkennen ist, dass die Datei *base.apk* vorhanden ist, welche die ursprüngliche App darstellt. Sie beinhaltet alle notwendigen Ressourcen um die App ausführen zu können. Des Weiteren das Verzeichnis *lib* mit den benötigten Bibliotheken der App.

```
root@s5neolte:/data/app/at.willhaben-1 # ls -l
-rw-r--r-- system   system   15272980 2019-04-16 12:27 base.apk
drwxr-xr-x system   system            2019-04-16 12:27 lib
drwxrwx--x system   install           2019-04-16 12:27 oat
root@s5neolte:/data/app/at.willhaben-1/oat/arm # ls -l
-rw-r--r-- system   u0_a29999 43516336 2019-04-16 12:27 base.odex
```

Listing 4.10: Ordnerstruktur am Beispiel at.willhaben-1

Im Verzeichnis *oat* befindet sich die Binärdatei *base.odex* der App. Diese wird beim Installationsprozess einmalig kompiliert. Das Betriebssystem optimiert dabei sämtliche App-Daten und erstellt dazu eine entsprechende OAT-Datei. Dies zeigt die Vorteile von ART, da dadurch eine App schneller gestartet werden kann und der Kompiliervorgang nur einmal durchgeführt werden muss.

4.2.4. Das ODEX-Dateiformat näher betrachtet

OAT-Dateien wurden mit der Android Version 5.0 (Lillipop) erstmalig eingeführt. Vorherige Versionen nutzten bis dahin das *Optimized Dalvik Executable .ODEX* Dateiformat, anstatt OAT für ausführbare Dateien.[28] Jedoch ist in Listing 4.11 zu sehen, dass auch hier das ODEX Format zu Einsatz kommt, obwohl es sich herbei um die Android Version 6.0.1 handelt. Betrachtet man die Datei jedoch mit dem Kommandozeilentool *readelf*, zeigt sich, dass innerhalb der ODEX-Datei die kompilierte OAT-Datei enthalten ist.

Eine OAT-Datei ist eine *ELF shared object*[5]-Datei, welche mehrere Sektionen enthält. Diese beinhaltet Header-Dateien, um die Struktur der OAT-Datei zu beschreiben, sowie DEX-Dateien und den kompilierten nativen Code. Die ELF-Datei hat drei dynamische Tabellen names *oatdata, oatexec und oatlastword*. Diese weisen darauf hin, welche Sektion die korrespondierenden OAT-Daten beinhaltet.[29]
Die genannten Sektionen beinhalten folgende Informationen:

- oatdata - Beinhaltet die OAT-Headerdateien und die inkludierten DEX-Dateien.
- oatexec - Beinhaltet sämtlichen generierten nativen Programmcode für kompilierte Funktionen.
- oatlastword - Signalisiert das Ende einer OAT-Datei und beinhaltet die letzten vier Bytes des generierten nativen Code.

Listing 4.11 zeigt den entsprechenden Abschnitt.

```
1  root@l440:# readelf -e base.odex
2  Section Headers:
3    [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
4    [ 0]                   NULL            00000000 000000 000000 00      0   0  0
5    [ 1] .dynsym           DYNSYM          00000134 000134 000040 10   A  2   0  4
6    [ 2] .dynstr           STRTAB          00000174 000174 000027 00   A  0   0  1
7    [ 3] .hash             HASH            0000019c 00019c 000020 04   A  1   0  4
8    [ 4] .rodata           PROGBITS        00001000 001000 15b4000 00   A  0   0 4096
9    [ 5] .text             PROGBITS        015b5000 15b5000 13ca28e 00  AX  0   0 4096
10   [ 6] .dynamic          DYNAMIC         02980000 2980000 000038 08   A  2   0 4096
11   [ 7] .shstrtab         STRTAB          00000000 2980038 000038 00      0   0  1
```

Listing 4.11: Headerauszug von ODEX-Datei mit dem Tool *readelf*

Zeile 8 des Listing 4.11 zeigt den Header-Teil *.rodata* bzw. *oatdata*. Dieser beinhaltet den Header der OAT-Datei, beginnend mit der Speicheradresse 00001000. Diese Section beinhaltet sämtliche Informationen über mögliche weitere DEX-Dateien und deren Header-Informationen, sowie alle benötigten OAT-Klassen. In Zeile 9 beginnt der Abschnitt *.text* - oder auch *.oatexec* genannt - welcher den eigentlichen kompilierten OAT-Programmcode enthält. Dieser beginnt mit der Startadresse 15b5000. In der Sektion *.text* wird *oatexec* und *oatlastword* zusammengefasst. Des Weiteren kann die ODEX-Datei mittels *hexdump* betrachtet werden. Zu sehen ist, dass bei der Speicheradresse 00001000 der OAT-Header beginnt. Listing 4.12 zeigt einen Ausschnitt.

[5]ELF - Executable and Linking Format beschreibt das Standard-Binärformat für ausführbare Dateien

```
root@l440:# hexdump -C base.odex | head -n 200
00001000
6f 61 74 0a 30 36 34 00  88 ec 75 21 03 00 00 00  |oat.064...u!....|
00001010
07 00 00 00 02 00 00 00  00 40 5b 01 00 00 00 00  |.........@[.....|
00001040
00 90 b0 70 a3 01 00 00  63 6c 61 73 73 70 61 74  |...p....classpat|
00001050
68 00 00 64 65 62 75 67  67 61 62 6c 65 00 66 61  |h..debuggable.fa|
00001060
6c 73 65 00 64 65 78 32  6f 61 74 2d 63 6d 64 6c  |lse.dex2oat-cmdl|
00001070
69 6e 65 00 2d 2d 7a 69  70 2d 66 64 3d 31 31 20  |ine.--zip-fd=11 |
```

Listing 4.12: ODEX-Datei mit *hexdump* betrachtet

OAT Header

Der OAT-Header beschreibt die allumfassende OAT-Datenstruktur. Beginnend mit dem *magic field 'oat'*, gefolgt von der derzeitigen Version des OAT-Formats, welches zum derzeitigen Moment '064' ist. Als drittes Feld im Header befindet sich *adler32_checksum*. Dieses dient als Checksumme für alle anderen Felder im Header bzw. wird diese auch zum prüfen von Manipulationen (engl. file corruption) herangezogen. Darauf folgend befindet sich das Feld *instruction_set*, welches auf die Architektur des Zielsystem schließen lässt. Das OAT-Format unterstützt mehrere Architekturen wie beispielsweise ARM, ARM 64-Bit, x86, x64, MIPS usw..

```
Feld                        Type        Beschreibung
magic                       ubyte[4]    Magic Wert fuer OAT-Format
version                     ubyte[4]    OAT-Version
adler32_checksum            uinit32     Adler32 Pruefsumme
instruction_set             uinit32     Instruction-Set Architektur
instruction_set_features    uinit32     Bitmaske Architektur-Features
dex_file_count              uinit32     Anzahl DEX-Dateien in OAT
executable_offset           uinit32     Offset von Exe-Section bis OAT-Data
```

Listing 4.13: Auszug des Aufbaus des OAT-Headers

dex_file_count beschreibt die Anzahl an integrierter DEX-Dateien in der APK-Datei. *executable_offset* zeigt zum Beginn des generierten nativen Programmcode, welcher der selbe aus der Sektion *oatexec* der

ELF-Datei ist. Listing 4.13 zeigt eine Zusammenfassung der sieben Anfangsfelder im OAT-Header. Es befinden sich grundsätzlich weitere Felder im Header, da diese jedoch nicht für diese Arbeit relevant sind, wird nicht näher darauf eingegangen.[29]

Da es sich bei dem Dateiformat um optimierten ausführbaren Programmcode handelt, wurde getestet welches Verhalten auftritt, wenn die ODEX-Datei vom Smartphone entfernt wird bzw. welche Folgen dies auf die App hat. Resultierend daraus zeigte sich, dass die App weiterhin aktiv ist, und diese benutzt werden kann. Da offensichtlich sämtlicher Programmcode im Speicher geladen wurde, sowie sämtliche benötigten Bibliotheken, die von der App genutzt werden. Durch einen Neustart des Smartphones wurde am Dateisystem eine neue *base.odex* Datei erstellt. Einen wesentlichen Unterschied zeigte die Dauer des Starts einer App nach einem Neustart des Smartphones. Bevor die App erstmalig ausgeführt wurde, wurde die entsprechende *base.odex* Datei der jeweiligen App entfernt. Dadurch entstand eine erhebliche Verzögerung bis die App vollständig geöffnet wurde. Der Start dauerte rund vier Sekunden, wobei im Vergleich mit dem optimierten Programmcode, die Dauer auf rund zwei Sekunden reduziert ist. Dies zeigte sich am benötigten Speicher, welcher höher ausgelastet ist, sofern keine *base.odex* Datei vorhanden ist. Resultierend aus den beschriebenen Tests gehen bemerkbare Veränderungen hervor, welche sich an Systemressourcen messen lassen. In weiterer Folge wurde getestet, welche Auswirkung die Manipulation bzw. Austausch der *base.odex* Datei mit sich bringt. Dazu wurde eine andere *base.odex* Datei herangezogen, welche auf den ursprünglichen Benutzer *system* und Gruppe *u0_a29999* geändert wurde. Im laufenden Betrieb zeigten sich keine Veränderungen, die App konnte normal gestartet werden und zeigte bei der Verwendung der App keine Anomalien. Nach einem Neustart des Geräts konnte die App auch weiter in einem normalen Zustand verwendet werden. Betrachtet man die *base.odex* Datei vor dem Neustart - also die gefälschte *base.odex* - sowie nach dem Neustart, sind unterschiedliche Prüfsummen zu erkennen. Listing 4.14 zeigt beide Dateien und den berechnenten MD5 Hash-Wert dazu.

```
1 root@s5neolte:/data/app/com.fluxguide.drosendorf-2/oat/arm # ls -l
2 -rw------- system   u0_a29999  4084144 2019-04-24 07:20 base.odex
3 //Original base.odex
4 root@s5neolte:/data/app/com.fluxguide.drosendorf-2/oat/arm #md5sum base.
     odex
5 8b0918d9efa6764d341cd0e25dfb9784  base.odex
6 //Manipulierte base.odex
7 root@s5neolte:/data/app/com.fluxguide.drosendorf-2/oat/arm # md5sum base.
     odex
8 143c2752c994a2dbfdbf90af247c988a  base.odex
```

Listing 4.14: MD5 Hash-Werte von base.odex vor und nach Neustart

Ein weiterer Test, resultierend aus den zuvor erfahrenen Erkenntnissen, war die Entfernung der *base.apk* Datei. Es zeigte sich, dass durch das entfernen die App nicht mehr gestartet werden konnte. Möchte man diese ausführen, wird eine Fehlermeldung angezeigt, dass beim Starten der App ein Fehler aufgetreten ist: *willhaben angehalten - Melden / OK*.
Dieser Test beweist, dass auch die *Android Runtime* weiterhin die *base.apk* Datei am System benötigt. Nach einem Neustart des Smartphones ist auch die Verknüpfung zur App entfernt, sowie die *base.apk* Datei bleibt weiterhin entfernt. Die einzigen Hinweise auf die App am Smartphone zeigt nur noch der Anwendungsmanager. Hier kann die App lediglich nur mehr deinstalliert werden. Dies bedeutet, dass am Dateisystem das Verzeichnis unter /data/app/ entfernt wird, sowie das generierte Profil unter /data/dalvik-cache/profiles. Erstellte Profile von Apps werden in folgendem Kapitel behandelt.

4.2.5. Profiles in /data/dalvik-cache/profiles

Im Verzeichnis */data/dalvik-cache/profiles* befinden sich sogenannte *Profiles* von Apps.[30] Wenn der dalvik-cache gelöscht wird, werden diese neu erstellt. Im Detail bedeutet dies, dass *profiles* bei der Kompilierung einer App erstellt werden. Ein Profil enthält sämtliche Informationen über Funktion einer App, welche bei jedem Start ausgeführt werden. Dies macht Sinn, da im Falle einer Neukompilierung einer App, dieses Profil herangezogen wird, um schneller ein *Optimized Dalvik Executable ODEX* zu generieren, umso schneller Ausführungszeiten beim erstmaligen Start (auch Cold-Start genannt) zu erreichen.[31] Dies bedeutet daher, dass bei der Neukompilierung die *base.apk* wieder gebraucht wird.

5. Manipulationen Erkennen - Proof-of-Concept

In dieser Arbeit wird ein Lösungsansatz vorgestellt, der den Fokus auf das Erkennen von Manipulationen bzw. Veränderung von Applikationen legt. Daher ist der wichtigste Bestandteil dieses Konzepts, die Analyse von Apps zu bestimmten Zeitpunkten. Dies bedeutet, dass ein Vergleich zwischen zwei Systemzuständen erst dann möglich ist, wenn dafür Referenzdaten vorliegen - somit ein Zustand installierter Apps am Smartphone zu einem bestimmten Zeitpunkt - sowie ein weiterer Zustand, nachdem Referenzdaten erstellt wurden. Natürlich können weitere gesicherte Systemzustände zu beliebigen Zeitpunkten als Referenz gewählt werden.
Des Weiteren wird dieser Lösungsansatz mit den Boardmitteln von Android und Ubuntu realisiert. Damit sind sämtliche Tools gemeint, die Android in Version 5 und höher mit sich bringt, sowie sämtliche Tools, welche Ubuntu Kommandozeilen-basierend über Repositories zur Verfügung stellt.
Im Zuge der Realisierung dieses Proof-of-Concept wurden insgesamt drei Skripts entwickelt, um eine einfache Handhabung mit den Tools zu bieten, sowie automatisiert Informationen gewinnen zu können. Zu Beginn wird das Shell-Skript *getAPK.sh* auf der Kommandozeile ausgeführt. Damit wird erreicht, dass sämtliche Third-Party Apps über die ADB-Schnittstelle auf den externen Rechner extrahiert werden. Im zweiten Schritt müssen von den zuvor gesicherten APK-Dateien sämtliche Informationen über eingesetzte Zertifikate extrahiert werden. Dies geschieht mit dem entwickeltem Python-Skript *app_forensics.py*. Welche Informationen darin enthalten sind werden im Kapitel 5.3 näher erläutert. Ab diesen Zeitpunkt sind für die Analyse alle notwendigen Daten gesammelt und daher werden die zuvor gesicherten APK-Dateien nicht mehr benötigt. Im dritten und letzten Schritt, müssen die extrahierten Daten analysiert werden. Dabei wird mit dem Python-Skript *signatures_check.py* auf definierte Identifikationsmerkmale untersucht, welche in Kapitel 5.2 beschrieben sind.
Als Resultat der Analyse wird die Anzahl der untersuchten Apps ausgegeben, sowie die Anzahl an positiven und negativen Ergebnissen. Positive Ergebnisse sind jene, welche keine Veränderungen der definierten Kriterien vorweisen. Negativ bewertet sind jene Apps, welche beim Vergleich zweier Systemzuständen Abweichungen vorweisen. Diese sollten anschließend manuell betrachtet werden.

5.1. Ausgangslage definieren

Um ein Proof-of-Concept realisieren zu können, müssen vorab wichtige Entscheidungskriterien definiert und berücksichtigt werden. Dafür wurde ein plausibles Szenarion gewählt, welches in der Praxis durchaus vorkommt. Ein Großteil von Smartphone-Nutzern haben in der Regel mehrere Third-Party Apps aus diversen Quellen installiert. Diese gelten jedoch als sehr attraktiv für Angreifer, um diese zu manipulieren, umso dem Benutzer oder Benutzerin Informationen zu stehlen bzw. anderweitig Schaden anrichten. Deshalb bezieht sich dieser Lösungsansatz auf Apps, welche von offiziellen App-Stores bezogen wurden bzw. anderer Fremdquellen und nicht vom Hersteller vorinstallierte Apps.
In weiterer Folge soll dieser Lösungsansatz mit den Standardbenutzerrechten durchführbar sein. In der Regel sind herkömmliche Smartphones nicht mit Root-Berechtigungen vorgesehen und verfügen daher nicht über ein *Custom ROM*, sondern eine vom Hersteller angepasste *Stock ROM* Variante. In dem in Folge durchgeführten Szenario wird davon ausgegangen, ein Smartphone mit einer beliebigen Anzahl an installierten Third-Party Apps zu sichern und diese einer entprechenden Untersuchungen zu unterziehen. Dabei wird angenommen, dass die erstmalige Datensicherung via ADB-Schnittstelle als Referenz definiert wird. Darauffolgend wird zu einem beliebigen Zeitpunkt nochmals dieser Prozess durchlaufen, um für einen entsprechenden Abgleich zweier Systemzustände verwendet werden zu können. Es wird auch zusätzlich davon ausgegangen, dass in der Zeit zwischen der Sicherung der Referenzdaten und der Sicherung des Vergleichszustandes keine Apps deinstalliert oder weitere hinzugefügt werden. Die Azahl der Apps soll somit gleich bleibend sein, um mögliche Fehlinterpretationen (False-Positives) zu vermeiden.
Oft gelebte Praxis zeigt, dass Smartphones nicht überall mitgenommen werden dürfen, aufgrund von möglichen Störeinflüssen auf andere elektronische Geräte bzw. möchte man sicherstellen nicht Opfer von Spionage zu werden. Zweiteres ist oftmals im geschäftlichen Umfeld zu sehen. Die Folge daraus ist, dass das persönliche Smartphone aus den Händen zu geben ist und in dieser Zeit außenstehenden Personen anvertraut werden muss. Da es speziell im asiatischen Raum bekanntlicherweise strenge Regulierungen und teilweise offizielle Überwachung der Nutzer und Nutzerinnen im Netz gibt, auch entsprechende Schadsoftware bzw. Spionagesoftware weit verbreitet ist, ist ein erhöhtes Risiko gegeben Opfer solcher Software zu werden. Umso wichtiger ist es, im entsprechendem Zeitraum solcher Aufenhalte, vorweg eine Sicherung durchzuführen, umso bei der Rückkehr in sicheres Umfeld, eine Analyse auf manipulierte Smartphone-Apps zu tätigen.

5.2. Identifikationsmerkmale definieren

Um eine mögliche Manipulation an Apps zu erkennen, müssen eindeutige Kriterien definiert werden, welche für die Prüfung herangezogen werden. Diese müssen für jede App einzigartig sein, da es sonst zu falschen Ergebnissen kommen kann. Bei diesem Lösungsansatz wurden folgende Kriterien herangezogen:

- APK-Name - Der APK-Name ist die Bezeichnung der App. Dieser zeigt an um welche installierte App es sich handelt, um diese leichter zu erkennen.
- Serial-Number - Jede APP besitzt eine Serial-Number um identifiziert zu werden. Diese wird bei der Entwicklung der App generiert, sobald die App in das APK-Format konvertiert wird.
- SHA1-Prüfsumme - Der berechnete SHA1-Wert ist der Fingerprint eines Zertifikats einer App.

Diese Kriterien müssen übereinstimmen, da sonst von einer Manipulation ausgegangen werden muss. Der Grund für die Wahl dieser Attribute liegt darin, dass diese in jeder APK-Datei enthalten sind. Die Prüfung der kompilierten Binär-Dateien wird deshalb ausgeschlossen, da diese als normaler Standardbenutzer oder Benutzerin nicht gesichert werden können. Durch die zuvor durchgeführte forensischen Analysen zeigte sich, dass diese sich in Verzeichnissen befinden, welche Root-Berechtigungen benötigt um gelesen zu werden.

5.3. Sichern installierter Third-Party Apps

Im Kapitel 4.1 wurde der grundlegende Prozessablauf der Datensicherung in dieser Arbeit beschrieben und wie dieser über die ADB-Schnittstelle durchgeführt werden kann. Um nun automatisiert sämtlichen Third-Party Apps sichern zu können, wurde dafür ein eigenes Shell-Skript *getAPK.sh* erstellt. Dieses Skript beinhaltet den Inhalt aus Listing 4.5 und wurde minimal erweitert, um als Shell-Skript zu funktionieren. Listing 5.1 zeigt einen Ausschnitt der Ausführung des Skripts.

```
1 root@l440:Diplomarbeit/forensic# ./getAPK.sh
2 * daemon not running; starting now at tcp:5037
3 * daemon started successfully
4 Pulling /data/app/com.amazon.mShop.android.shopping-2/base.apk
5 /data/app/com.amazon.mShop.android.shopping-2/base.apk: 1 file pulled. 7.6
     MB/s (47929857 bytes in 6.010s)
6 Pulling /data/app/org.telegram.messenger-2/base.apk
```

```
7 /data/app/org.telegram.messenger-2/base.apk: 1 file pulled. 7.4 MB/s
    (17385175 bytes in 2.255s)
8 Pulling /data/app/com.whatsapp-1/base.apk
9 /data/app/com.whatsapp-1/base.apk: 1 file pulled. 7.3 MB/s (25133621 bytes
    in 3.271s)
```

Listing 5.1: Sichern der Third-Party Apps mittels Shell-Skript *getAPK.sh*

In Zeile zwei ist zu erkennen, dass der Dienst für die ADB-Schnittstelle noch nicht gestartet wurde und dieser im Anschluss erfolgreich läuft. Danach beginnt die eigentliche Sicherung der Apps. Dieser Schritt kann je nach Anzahl installierter Apps zeitlich variieren.

5.4. Informationen extrahieren

Nachdem eine Sicherung der Apps, wie schon im Kapitel 4.1 erklärt, oder mittels Shell-Skript *getAPK.sh* über die ADB-Schnittstelle durchgeführt wurde, sind alle benötigten Informationen vom Smartphone extrahiert. Dies bedeutet, dass die Verbindung zum Smartphone getrennt werden kann. Ab diesem Zeitpunkt werden alle weiteren Schritte getrennt vom Smartphone durchgeführt. Nun müssen daraus sämtliche definierten Entscheidungskriterien herausgefiltert werden. Dieser Schritt wird mit dem selbst geschriebenen Python-Skript *app_forensics.py* durchgeführt. Listing 5.2 zeigt den entsprechenden Abschnitt des Skripts.

```
1 location = "find . -name \"*.apk\" -exec echo \"APK: {}\" \\; -exec sh -c
    \'keytool -printcert -jarfile \"{}\"\' \\; > $(date +'%Y-%m-%d')
    _RawSignatures.txt"
2 getRawData = os.system(location)
```

Listing 5.2: Extrahieren der Roh-Daten

Um mit den Boardmitteln des Betriebssystems zu arbeiten, wurde die *Python Library os* importiert. Damit ist es möglich über ein Python-Skript, Shell-Commands auszuführen. Hierbei werden im aktuellen Verzeichnis, sowie allen Unterverzeichnissen nach APK-Dateien gesucht. Im Anschluss wird dem Tool *keytool* jede gefunde APK-Datei übegeben und mit den Parameter *-printcert* sämtliche Attribute des verwendeten Zertifikats extrahiert. Im Rohformat werden mehr Informationen gewonnen als für die eigentliche Prüfung notwendig sind. Jedoch sind diese nicht uninteressant und werden daher gespeichert, sollten diese für weitere Lösungsansätze relevant sein. Im Detail betrachtet werden folgende Daten extrahiert:

- APK - APK beschreibt den Paketnamen der App. Beispielsweise *com.diepresse.apk*
- Owner - Der Owner ist der Besitzer des Zertifikats. Am Beispiel der Presse-App sieht dies wie folgt aus: *Owner: CN=Die Presse, L=Vienna, C=AT*. *CN* ist der *CommonName*, somit der Name des Besitzers des Zertifikats. *L* gibt das jeweilige Land an. In diesem Fall Wien. *C* enspricht dem *Country Code* und dieser ist für Österreich *AT*
- Issuer - Dieses Attribut beschreibt den Aussteller des Zertifikats. In diesem Fall ist dieser gleich mit dem Attribut *Owner*. Beschrieben wird dieser mit den selben Attributen - *CommonName, Land, Country Code*.
- Serial Number - Jede App hat eine eigene *Serial Number*. Über diese kann eine App identifiziert werden. Die Presse App hat die *Serial Number: 4ec6724f*
- Valid from - Dieses Attribut gibt den Zeitrahmen an, wielange ein Zertifikat Gültigkeit hat. Am Beispiel der Presse App sieht dies folgend aus: *Fri Nov 18 15:57:19 CET 2011 until: Sun Oct 25 15:57:19 CET 2111*. Gültigkeit vom Jahr 2011 bis zum Jahr 2111, somit wurde die Dauer der Gültigkeit auf 100 Jahre für das Zertifikat vergeben.
- Certificate fingerprints - Wie schon die Bezeichnung verrät handelt es sich hierbei um die generierten Fingerprints des eingesetzten Zertifikats für die App. Diese sind als SHA1 und SHA256 vorhanden. Als Beispiel wie folgt SHA1 gezeigt:
  SHA1: F1:CA:DF:03:29:1F:F1:5C:1E:B1:EF:A4:20:26:ED:7C:5E:A4:31:96 und SHA256:
  SHA256: F5:4A:CB:EA:1F:5B:9B:0F:6A:66:EC:14:FB:8D:52:3A:16:22:55:77:94:3C:C3:0F:7C:8D:3C:B4:D2:43:DF:44. Der Fingerprint ist einer App eindeutig zuordenbar.
- Signature algorithm name - Gibt den Verwendetet Algorithmus an, welcher für die Erstellung des Zertifikats verwendet wurde. In diesem Beispiel ist dieser *SHA1withRSA*
- Subject Public Key Algorithm - Gibt an welcher Algorithmus für das signieren des Public-Key verwendet wird. Hier wurde *1024-bit RSA key* verwendet.
- Version - Gibt die Version des Zertifikats an. Bei dieser App ist *Version 3* im Einsatz.

Das Tool *keytool* ist ein Java-basierendes Key- und Zerifikats- management Programm. Es bietet die Möglichkeit öffentliche und private Schlüsselpaare zu verwalten. Unter anderem können damit auch Zertifikate ausgelesen werden. *keytool* ist Teil des *Java Development Toolkit JDK* und muss daher eventuell nachinstalliert werden, sollte Java am System nicht vorhanden sein.

Alle gewonnen Informationen werden anschließend in eine Text-Datei geschrieben. Diese werden mit dem Datum des jeweiligen Durchführungszeitpunkts, sowie der Bezeichnung *_RawSignatures.txt* benannt. Somit ist die Verwechslung, eine falsche Datei zu analysieren, eingeschränkt und es können beliebig viele Dateien in einem Verzeichnis eindeutig identifiziert werden.
Das Python-Skript führt anschließend noch einen weiteren Schritt durch, nämlich die Extrahierung der definierten Identifikationsmerkmalen. Dafür wird die zuvor erstellte Text-Datei geöffnet und in eine Liste eingelesen. Danach wird mit Hilfe der *Python3 Library re* über Regex-Funktionen[1] nach dem APK-Namen, Serial-Number, sowie SHA-1 Fingerprint des Zertifikats gesucht und gefiltert. Listing 5.3 zeigt den entsprechenden Ausschnitt im Programm-Code der dafür zuständig ist.

```
/* Filtern der Identifikationsmerkmale */
r = re.compile(r'(APK:|number|SHA1:)')
newlist = filter(r.search, signature_list)
filter_list = list(newlist)
/* Speichern der gefilterten List als Text-Datei */
save_file = str(datum) + "_extracted_signatures"
with open(save_file, mode='wt', encoding='utf-8') as myfile:
    myfile.write('\n'.join(filter_list))
```

Listing 5.3: Filtern auf Identifikationsmerkmale

Die Ausführung des Skripts wird im Listing 5.4 demonstriert. Da das Skript auf der Version Python3 basiert, wird dies entsprechend beim Ausführen angegeben. Darauf folgend die Skriptbezeichnung *app_forensics.py*.

```
root@l440:/Diplomarbeit/forensic# python3 app_forensics.py
##########################################
Get Signature Information from APK-Files

This could take few moments
##########################################

Signaturen werden extrahiert
Done!
```

Listing 5.4: Ausführen des Skripts *app_forensics.py*

[1] Regular Expressions - Erkennen von Zeichenketten durch definierte Regeln

Nach der erfolgreichen Durchführung dieses Skripts wurden zwei Dateien im Verzeichnis angelegt. Siehe dazu Listing 5.5.

```
-rw-r--r-- 1 root   root       3943 Apr 28 15:58
2019-04-28_extracted_signatures
-rw-r--r-- 1 root   root      22991 Apr 28 15:58
2019-04-28_RawSignatures.txt
```

Listing 5.5: Erstellte Textdateien durch *app_forensics.py*

Somit ist die erste Sicherung erfolgreich abgeschlossen. Diese kann nun als Referenz dienen, um weitere gesicherte Systemzustände damit auf Manipulationen abzugleichen. In weiterer Folge muss nun der Prozess der Datensicherung bis zum Extrahieren der Identifikationsmerkmale nochmals durchgeführt werden. Danach liegen zwei gesicherte Systemzustände vor, welche miteinander verglichen werden können.

5.5. Vergleich gesicherter Systemzustände

Nachdem eine Datensicherung als Referenz und eine weitere zum Abgleich auf Manipulationen vorliegen, kann diese mit dem erstellten Python-Skript *signatures_check.py* durchgeführt werden. Dieses Skript erwartet daher zwei Text-Dateien als Übergabeparameter bei der Ausführung des Skripts. Der Ablauf des Skripts besteht im ersten Schritt aus dem Einlesen der beiden Dateien in den Datentyp *List*.
Im zweiten Schritt werden die jeweiligen Listen-Elemente gegenüber gestellt und entsprechend auf Gleichheit verifiziert.
Hierbei sind die zuvor definierten Identifikationsmerkmale mit logischen 'UND' verknüpft. Dies bedeutet, dass alle drei Merkmale Wahr - also gleich sein müssen - um die Bedingung zu erfüllen. Sollte dies nicht der Fall sein, wird dies als negative App-Signatur - somit als manipulierte App - eingestuft. Am Ende werden die Resultate der Analyse aufgezeigt. Sofern keine Manipulationen erkannt wurden, ist lediglich die Anzahl der untersuchten Apps gelistet, sowie wieviel davon positiv und negativ eingestuft wurden. Listing 5.6 zeigt den Code-Ausschnitt der angewandten UND-Logik.

```
reference_sig_file = sys.argv[1]   # Path of reference file
signature_to_check = sys.argv[2]    # Path of file to check

for i in range (0,list_len):
    if(ref_signature_list[check_apk] == check_signature_list[check_apk] and
        ref_signature_list[check_serial] == check_signature_list[
```

```
        check_serial]  and ref_signature_list[check_sha] ==
        check_signature_list[check_sha]):

print("positive Signatures: " + str(good))
print("negative Signatures: " + str(suspicious))
print("Total number of checked signatures: " + str(list_len) + "\n")
```

Listing 5.6: Funktionsumfang des Skripts *signatures_check.py*

Im folgendem wird das Skript *signautes_check.py* ausgeführt. Dabei wird die Referenzdatei und die zu überprüfende Datei übergeben. Listing 5.7 zeigt die erfolgreiche Analyse.

```
root@l440:/Diplomarbeit/forensic# python3 signature_check.py 2019-04-27
    _extracted_signatures 2019-04-28_extraced_signatures

    ### RESULT OF INVESTIGATION ###
positive Signatures: 30
negative Signatures: 0
Total number of checked signatures: 30
```

Listing 5.7: Ausführen des Skripts *signatues_check.py*

Listing 5.7 zeigt einen positiven Vergleich zweier Systemzustände. Insgesamt wurden 30 Signaturen von Apps analysiert. Null davon sind als negative Signatur erkannt worden. In diesem Fall kann davon ausgegangen werden, dass keine Manipulationen am Smartphone stattgefunden haben.
Als nächstes wird demonstriert, wie die Darstellung bei einer Manipulation aussieht. Listing 5.8 zeigt die Ausführung des Python-Skripts, jedoch wird eine Manipulation festgestellt.

```
root@l440:Diplomarbeit/forensic# python3 signature_check.py 2019-04-27
    _extracted_signatures 2019-04-28_extraced_signatures

    ### RESULT OF INVESTIGATION ###

Signature manipulated!
reference :['APK:', './org.mozilla.firefox.apk']
check :['APK:', './org.mozilla.firefox.apk']
reference :['Serial', 'number:', '4c72fd88']
check :['Serial', 'number:', '4b83f4c2']
reference :['SHA1:', '92:0F:48:76:A6:A5:7B:4A:6A:2F:4C:CA:F6:5F:7D:29:CE
```

```
    :26:FF:2C']
check :['SHA1:', '9E:A5:84:EB:2D:82:B6:1D:78:EF:88:5A
    :53:45:36:21:39:41:17:4D']

positive Signatures: 29
negative Signatures: 1
Total number of checked signatures: 30
```

Listing 5.8: Ausführen des Skripts *signatues_check.py* erkennt Manipulation

Wie in Listing 5.8 nun zu sehen ist, wird eine Manipulation an der Signatur der App *org.mozilla.firefox.apk* erkannt. Die Ausgabe ist so aufgebaut, dass jeweils der APK-Name der Referenzdatei, gefolgt von dem Namen der zu prüfenden APK-Datei erscheint. Dies wird genauso für die beiden weiteren Identifikationsmerkmale ausgegeben.
Zusätzlich ist in Zeile 14 zu erkennen, dass eine Signatur als negativ erkannt wurde. Schlussendlich zeigt ein genauerer Blick auf die Ausgabe, dass die *Serial Number* und *SHA-1 Fingerprint* von den Referenzwerten abweichen.

Somit ist ein möglicher Lösungsansatz und den dazugehörigen Proof-of-Concept in dieser Arbeit vorgestellt worden. Die Erkenntnis dieses Lösungsansatzes zeigt, dass es möglich ist, nur mit den Boardmitteln des Smartphones, sowie dem Betriebssystem des ausführenden Rechners, Daten zu extrahieren und diese in Folge zu verifizieren.

6. Schlussfolgerung

Smartphones zeichnen sich durch ihre Vielfältigkeit an Möglichkeiten aus. Sie sind als ständiger Begleiter mit dabei, da diese einen großen Teil unseres Lebens digital verwalten. Doch erst durch die Installation von Apps, umso mit anderen Personen Nachrichten, E-Mails, Chats uvm. auszutauschen, macht es erst zu einem nicht mehr wegzudenkenden Gegenstand in unserer Gesellschaft.
Dadurch ist es umso wichtiger, die Sicherheit in Apps zu verbesseren, notwendige Maßnahmen zu implementieren, umso nicht Opfer von Cyberkriminellen zu werden. Die Herausforderung für Entwickler und Entwicklerinnen ist es, schon beim Design einer App die notwendigen Sicherheitsfeatures einzuplanen, um dadurch nicht mit erforderlichen Sicherheitsupdates hinterherhinken zu müssen. Aber auch der Besitzer oder Besitzerin eines Smartphones selbst, sollte sich vor dem installieren einer App gedanken machen, von welcher Quelle Apps bezogen werden und ob diese einer seriösen Quelle abstammen.
In dieser Arbeit ging hervor, dass eine installierte App, zur Laufzeit nicht manipuliert werden kann. Dafür sorgt die Android Runtime mit entsprechenden Sicherheitsmechanismen. Der Programmcode einer App wird während der Installation von der Andorid Runtime optimiert und in das entsprechende ODEX-Binärformat kompiliert. Dieser Vorgang wird einmalig durchgeführt. Das Android Betriebssystem erkennt beim Start des Smartphones, sollte eine Binärdatei nicht der original Binärdatei entsprechen und kompilliert diese neu. Dadurch wird auf die APK-Datei der App zurückgegriffen.
Daher wurde ein Lösungsansatz entwickelt, welcher die Prüfung von APK-Dateien, speziell deren Signaturen, durchführt. Ein Angreifer oder Angreiferin könnte eine App originalgetreu nachprogrammieren, umso die Erkennung einer Fake-App zu erschweren. Ein möglicher Angreifer oder Angreiferin könnte einen Key-Logger in die App integrieren, umso Kennwörter des Benutzer oder Benutzerin abzugreifen, oder andere Daten zu stehlen. Daher ist die Prüfung von Signaturen eine Möglichkeit, zu verifizieren, wer eine App erstellt und signiert hat. Des Weiteren kann dadurch festgestellt werden, ob es sich um ein Debugging-Zertifikat, also ein Zertifikat, dass nicht über den offiziellen Play-Store verteilt wird, oder es sich um ein gültiges Release-Zertifikat handelt. Ein wesentliches Merkmal dieses Lösungsansatzes ist, dass für die Sicherung der Apps und die darauf folgende Analyse kein Root-Berechtigungen erforderlich sind und dies getrennt vom Smartphone statt findet.
Für weitere wissenschaftliche Forschungstätigkeiten in diesem Bereich, könnten noch zusätzliche Iden-

tifikationsmerkmale herangezogen werden bzw. die Prüfung auf Manipulationen am Smartphone selbst durchgeführt werden. Die Entwicklung einer eigenen App, welche direkt am Smartphone automatisiert, in definierten Intervallen, selbstständig prüft ob Signaturen verändert wurden und diese Ergebnisse am Dateisystem ablegt, könnte den Prozess der Analyse weiter vereinfachen.

Eine weitere mögliche Forschungsarbeit könnte zusätzlich, parallel zur Prüfung installierter Apps, eine Verbindung mit dem offiziellen App-Store herstellen, umso eine aktuelle Vergleichsdatei zu beziehen, um in weiterer Folge einen zusätzlichen Vergleichswert bei der Prüfung auf Manipulation zu bekommen.

Abschließend kann festgehalten werden, dass es verschiedene Ansätze zur Erkennung von manipulierten Apps gibt, jedoch ist die Entwicklung weiterer Konzepte unbedingt notwendig. Die Technologie von Smartphones und Apps ist ein sehr schnelllebiger und benötigt daher große Aufmerksamkeit aus Sicht der IT-Sicherheit.

A. Anhang

```
#!/bin/bash

for package in $(adb shell pm list packages -3 | tr -d '\r' | sed 's/
    package://g'); do apk=$(adb shell pm path $package | tr -d '\r' | sed 's
    /package://g'); echo "Pulling $apk"; adb pull -p $apk "$package".apk;
    done

echo DONE!
```

Listing A.1: Shell-Skript *getAPK.sh*

```
#!/usr/bin/env python3

#Title:              app_forensics.py
#Autor:              Thomas Pokorny, BSc
#Course of studies: Information Security
#Date:               18.4.2019
#Version:            0.1
##########################################

import os
import sys
import re
import datetime

print("##########################################")
print("Get Signature Information from APK-Files\n")
print("This could take few moments\n")
print("##########################################\n")


```

```
21
22 location = "find . -name \"*.apk\" -exec echo \"APK: {}\" \\; -exec sh -c
       \'keytool -printcert -jarfile \"{}\"\' \\; > $(date +'%Y-%m-%d')
       _RawSignatures.txt"
23 getRawData = os.system(location)
24
25 datum = datetime.date.today()
26 filename = str(datum) + "_RawSignatures.txt"
27 print(filename + "created!\n")
28 print("Signaturen werden extrahiert\n")
29 print("\nDone!")
30
31 #define empfty list
32 signature_list = []
33
34 # open file and read the content in a list
35 with open(filename, 'r') as filehandle:
36     for line in filehandle:
37         # remove linebreak \n
38         list_element = line[:-1]
39
40         # add item to list
41         signature_list.append(list_element)
42
43 # regex for APK-Name, Serial Number, SHA1 Fingerprint
44 r = re.compile(r'(APK:|number|SHA1:)')
45
46 # filter only of Regex
47 newlist = filter(r.search, signature_list)
48 filter_list = list(newlist)
49
50 # defined filename for extracted signatures
51 save_file = str(datum) + "_extracted_signatures"
52
53 with open(save_file, mode='wt', encoding='utf-8') as myfile:
54     myfile.write('\n'.join(filter_list))
```

Listing A.2: Python-Skript *app_forensics.py*

```
#!/usr/bin/env python3

#Title:              signature_check.py
#Autor:              Thomas Pokorny, BSc
#Course of studies: Information Security
#Date:               18.4.2019
#Version:            0.1
##########################################

import os
import sys
import re
import datetime
import difflib

totalArgs = len(sys.argv)   #amount of arguments

#If the number of arguments is <= 1 exit the programm
if totalArgs <= 2:
    print("Enter <reference-signature-file> <signature-to-check-file>")
    exit(0)

reference_sig_file = sys.argv[1]   # Path of reference file
signature_to_check = sys.argv[2]    # Path of file to check


#define empty list for reference signature
ref_signature_list = []

# open file and read the content in a list
with open(reference_sig_file, 'r') as filehandle:
    for line in filehandle:
        # remove linebreak \n
        list_element = line.split()

        # add item to list
        ref_signature_list.append(list_element)
```

```
38
39 #define empty list for signatures to check
40 check_signature_list = []
41
42 # open file and read the content in a list
43 with open(signature_to_check, 'r') as filehandle:
44     for line in filehandle:
45         # remove linebreak \n
46         list_element = line.split()
47
48         # add item to list
49         check_signature_list.append(list_element)
50
51 print("\n\t### RESULT OF INVESTIGATION ###\n")
52
53 list_len = int(len(check_signature_list) / 3)
54 check_apk = 0
55 check_serial = 1
56 check_sha = 2
57 good = 0
58 suspicious = 0
59 for i in range (0,list_len):
60     if(ref_signature_list[check_apk] == check_signature_list[check_apk] and
          ref_signature_list[check_serial] == check_signature_list[
        check_serial]
61         and ref_signature_list[check_sha] == check_signature_list[check_sha
            ]):
62         good = good +1
63     else:
64         print("Signature manipulated!\n")
65         print("reference :" + str(ref_signature_list[check_apk]))
66         print("check :" + str(check_signature_list[check_apk]))
67         print("reference :" + str(ref_signature_list[check_serial]))
68         print("check :" + str(check_signature_list[check_serial]))
69         print("reference :" + str(ref_signature_list[check_sha]))
70         print("check :" + str(check_signature_list[check_sha]))
71         print("\n")
72         suspicious = suspicious +1
```

```
73      check_apk = check_apk +3
74      check_serial = check_serial +3
75      check_sha = check_sha +3
76
77  print("positive Signatures: " + str(good))
78  print("negative Signatures: " + str(suspicious))
79  print("Total number of checked signatures: " + str(list_len) + "\n")
```

Listing A.3: Python-Skript *signatures_check.py*

Abbildungsverzeichnis

Listingverzeichnis

Literaturverzeichnis

[1] N. Y. P. Lukito, F. A. Yulianto, and E. Jadied, "Comparison of data acquisition technique using logical extraction method on unrooted android device," in *2016 4th International Conference on Information and Communication Technology (ICoICT)*, May 2016, pp. 1–6.

[2] S. Beiersmann. (2017, August) Android steigert weltweiten marktanteil auf 87,7 prozent. ZDNet.de. [Online]. Available: http://www.zdnet.de/88308701/gartner-android-steigert-weltweiten-marktanteil-auf-877-prozent/?inf_by=5a92dcee681db88d688b4847

[3] A. H. Daniel Regalado, Shon Harris, *Gray Hat Hacking - The Ethical Hacker's Handbook - Fourth Edition*, 2015, p. 724.

[4] P. Faruki, V. Laxmi, V. Ganmoor, M. S. Gaur, and A. Bharmal, "Droidolytics: Robust feature signature for repackaged android apps on official and third party android markets," in *2013 2nd International Conference on Advanced Computing, Networking and Security*, Dec 2013, pp. 247–252.

[5] E. Kim, S. Kim, and J. Choi, "Detecting illegally-copied apps on android devices," in *2013 International Conference on IT Convergence and Security (ICITCS)*, Dec 2013, pp. 1–4.

[6] J. Bang, H. Cho, M. Ji, T. Cho, and J. H. Yi, "Tamper detection scheme using signature segregation on android platform," in *2015 IEEE 5th International Conference on Consumer Electronics - Berlin (ICCE-Berlin)*, Sep. 2015, pp. 459–462.

[7] S. Sahu, "An analysis of whatsapp forensics in android smartphones," in *International Journal of Engineering Research*, 2014.

[8] Z. Jovanovic, "Android forensics techniques," in *International Academy of Design and Technology*, 2012.

[9] J. Grover, "Android forensics: Automated data collection and reporting from a mobile device," in *Rochester Institute of Technology*, 2013.

[10] G. C. K. Jeff Lessard, "Android forensics: Simplifying cell phone examinations," in *Small scale digital device forensics journal*, 2011.

[11] J. Li, D. Gu, and Y. Luo, "Android malware forensics: Reconstruction of malicious events," in *2012 32nd International Conference on Distributed Computing Systems Workshops*, June 2012, pp. 552–558.

[12] D. K. Namheun Sona, Yunho Leea, "A study of user data integrity during acquisition of androiddevices," in *Digital Investigation 10;UCD Digital Forensic Investigation Research Group;Center for Information Security Technologies (CIST)*, 2013.

[13] N. C. Timothy Vidasa, Chengye Zhangb, "Toward a general collection methodology for android devices," in *Digital Investigation 8;Carnegie Mellon ECE/CyLab, USA*, 2011.

[14] J. H. C. Seung Jei Yang, "New acquisition method based on firmware update protocolsfor android smartphones," in *Digital Investigation 14; The Affiliated Institute of ETRI*, 2015.

[15] T. Z. Xiaodong Lina, Ting Chenb, "Automated forensic analysis of mobile applications on android devices," in *Digital Investigation 26; DFRWS 2018 USA - Proceedings of the Eighteenth Annual DFRWS USA*, 2018.

[16] I. B. Daniel Walnyckya, "Network and device forensic analysis of android social-messaging applications," in *Digital Investigation 14; DFRWS 2015 USA*, 2015.

[17] E. Latifa and E. K. M. Ahmed, "Android: Deep look into dalvik vm," in *2015 5th World Congress on Information and Communication Technologies (WICT)*, Dec 2015, pp. 35–40.

[18] DroidWiki. (2018) dx. [Online]. Available: https://www.droidwiki.org/wiki/Dx

[19] Android.com. (2018) The build process. [Online]. Available: https://developer.android.com/studio/build/

[20] C. Hoffman. (2014, July) Android usb connections explained: Mtp, ptp, and usb mass storage. [Online]. Available: https://www.howtogeek.com/192732/android-usb-connections-explained-mtp-ptp-and-usb-mass-storage/

[21] M. Baykara and E. Çolak, "A review of cloned mobile malware applications for android devices," in *2018 6th International Symposium on Digital Forensic and Security (ISDFS)*, March 2018, pp. 1–5.

[22] K. Allix, Q. Jerome, T. F. Bissyandé, J. Klein, R. State, and Y. L. Traon, "A forensic analysis of android malware – how is malware written and how it could be detected?" in *2014 IEEE 38th Annual Computer Software and Applications Conference*, July 2014, pp. 384–393.

[23] N. Lo, S. Lu, and Y. Chuang, "A framework for third party android marketplaces to identify repackaged apps," in *2016 IEEE 14th Intl Conf on Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, Aug 2016, pp. 475–482.

[24] D. A. Developer. (2018) Android dynamic feature modules : The future. [Online]. Available: https://medium.com/mindorks/dynamic-feature-modules-the-future-4bee124c0f1

[25] S. Dogan and E. Akbal, "Analysis of mobile phones in digital forensics," in *2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, May 2017, pp. 1241–1244.

[26] B. Anderson. (2013) Understanding the android file hierarchy. [Online]. Available: https://www.all-things-android.com/content/understanding-android-file-hierarchy

[27] Android.com. (2019) Art und dalvik. [Online]. Available: https://source.android.com/devices/tech/dalvik

[28] fileinfo.com. (2016) .oat file extension. [Online]. Available: https://fileinfo.com/extension/oat

[29] P. Sabanal, "Hiding behind art," in *HITBSecConf2015 Amsterdam*. IBM Security Systems, 2015. [Online]. Available: https://www.blackhat.com/docs/asia-15/materials/asia-15-Sabanal-Hiding-Behind-ART-wp.pdf

[30] G. Bansal. (2018) Android runtime improvments. [Online]. Available: https://medium.com/mindorks/android-runtime-improvements-e69bf7c1d10c

[31] Android.com. (2019) Implementing art just-in-time (jit) compiler. [Online]. Available: https://source.android.com/devices/tech/dalvik/jit-compiler