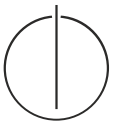
Technische Universität München
Fakultät für Informatik

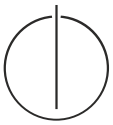


Bachelorarbeit in Informatik

Implementierung eines Reengineeringtools für Android

Philipp Schreitmüller

Technische Universität München
Fakultät für Informatik



Bachelorarbeit in Informatik

Implementation of a Reengineeringtool for Android

Implementierung eines Reengineeringtools für Android

Bearbeiter:	Philipp Schreitmüller
Aufgabensteller:	Prof. Dr. Uwe Baumgarten
Betreuer:	Nils Kannengießer, M.Sc.
Abgabedatum:	15. März 2014

Aus Gründen der besseren Lesbarkeit wird auf die gleichzeitige Verwendung männlicher und weiblicher Sprachformen verzichtet. Sämtliche Personenbezeichnungen gelten gleichwohl für beiderlei Geschlecht.

Ich versichere, dass ich diese Bachelorarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 13. März 2014 Philipp Schreitmüller

Zusammenfassung

In meiner Bachelorarbeit soll eine Reengineering-Applikation für das mobile Betriebssystem Android entwickelt werden. Mit der App soll es möglich sein, andere vorhandene Apps in ihre Einzelteile, nämlich Programmressourcen und entsprechenden Quellcode, zu zerlegen. Dazu wird eine angepasste Version des Programms apktool verwendet. Anschließend können diese in einem Filebrowser betrachtet und bei Bedarf in einem Editor beliebig verändert werden. Die dekompilierte App wird als Projekt gespeichert und bleibt deshalb auch nach Neustart der App erhalten. Außerdem besteht die Möglichkeit, Quellcode und Ressourcen mit anderen installierten Apps zu teilen. Zu beliebigem Zeitpunkt kann wieder eine funktionierende App erstellt werden, welche mit einem Testschlüssel signiert ist. Dadurch kann sie direkt auf dem Android-Gerät installiert werden. Die App wurde sowohl für Tablet als auch für Smartphones entwickelt.

Abstract

The subject of this thesis is the development of a reengineering app for the mobile operating system Android. The app should make it possible to disassemble existing apps into its resources and source code which are shown in a filebrowser and can be edited in an editor. For this step an adapted version of apktool is used. The decompiled app is saved as a project and is therefore still available after relaunching the app. Additionally it is possible to share source code and resources with other apps. In another step the edited artefacts can be compiled to an app again which is signed with a testkey. This makes it possible to install the app directly on the Android device. The app was developed for smartphones and tablets.

Akronyme

IPC	Inter-process Communication
App	Application
DVM	Dalvik Virtual Machine
JVM	Java Virtual Machine
API	Application programming interface
UI	User interface
APK	Android application package file
adb	Android Debug Bridge
NEL	U.S. Navy Electronic Labs
DMCA	Digital Millennium Copyright Act
DRM	Digital Rights Management
IPR	Intellectual Property Rights
NDK	Android Native Development Kit
JAR	Java Archive
IDE	Integrated development environment
aapt	Android Asset Packaging Tool
AIDL	Android Interface Definition Language
GUI	Graphical user interface
JDK	Java Development Kit

Kapitelübersicht

Einführung

Einführend wird Grundsätzliches über die Entwicklung von Android erklärt. Außerdem wird die Gesamtarchitektur des Systems beschrieben, insbesondere die Dalvik Virtual Machine (DVM).

Allgemeines zu Reengineering

Ausgehend von einem kurzen Abschnitt über die Entwicklung des Reengineerings allgemein wird auf rechtliche, aber auch moralische Aspekte des Reengineerings eingegangen. Abschließend werden grundlegende Schutzmethoden gegen Decompiling erläutert.

Reengineering von Android Apps

Nachdem auf Grundsätzliches über Android und Reengineering eingegangen worden ist, soll nun im Speziellen das Decompiling von Android Applikationen untersucht werden. Anschließend werden Sicherheitskonzepte von Android erklärt.

Verwandte Arbeiten und Projekte

Auch andere Projekte haben sich mit Decompiling & Analyse von Android-Application (App)s beschäftigt. Relevante Arbeiten werden in diesem Kapitel vorgestellt. *Dexplorer* und *AppGuard* sind Android-Applikationen, während *Baksmali/Smali* und *Apktool* als Desktop-Programme konzipiert worden sind.

Implementierung

In diesem Kapitel wird auf die Implementierung des Reengineeringtools *AppEditor* eingegangen. Nach der Beschreibung der implementierten Funktionen, der Zielplattform und der Modifikationen des *apktool*, wird die Architektur von *AppEditor* erklärt.

Qualitätsanalyse

In diesem Kapitel wird der Testplan von *AppEditor* besprochen. Zum Testen wurde eine App mit Namen *AppEditor Tests* entwickelt. Anhand *AppEditor Tests* und der App *JDiary*, die im Rahmen des *Android Praktikum* bei Herrn Kannengießer im Sommersemester 2013 entwickelt wurde, wird die Funktionalität von *AppEditor* sichergestellt. Am Schluss wird noch die Performance von *AppEditor* analysiert.

Ergebnis

Nach einer Zusammenfassung der Ergebnisse folgt ein Ausblick in die Zukunft des Android App-Format und es werden zudem Verbesserungen, die in der Zukunft noch implementiert werden können, angesprochen.

Inhaltsverzeichnis

1 Einführung

Einführend wird Grundsätzliches über die Entwicklung von Android erklärt. Außerdem wird die Gesamtarchitektur des Systems beschrieben, insbesondere die DVM.

1.1 Geschichte und Verbreitung von Android

Die Geschichte von Android ist untrennbar mit dem Namen *Andy Rubin* verbunden. Er gilt als Erfinder des Betriebssystems Android und gründete 2003 das Unternehmen *Android Inc* [13]. 2005 wird das noch junge Unternehmen von Google für 50 Millionen US-Dollar aufgekauft. Aus heutiger Sicht ein unglaublich niedriger Preis, wenn man bedenkt, dass mittlerweile einzelne Apps für Millionenbeträge den Besitzer wechseln.
2007 macht Google den nächsten wichtigen Schritt in ihrer Mission, Android als mobiles Betriebssystem für die Massen zu etablieren, indem sie die *Open Handset Alliance* gründen, in der Handylieferanten, Netzbetreiber und Chiphersteller zusammensitzen. Diese große Allianz ist wichtig, um Apple wirksam Widerstand zu leisten, das zu diesem Zeitpunkt den Smartphone Markt dominiert. Google stellt das Betriebssystem *open-source*, also auch ohne Lizenzgebühren, für die Handyhersteller zur Verfügung.
Im Herbst 2008 kommt schließlich das erste, von HTC gefertigte, Android Smartphone mit Android 1.1 auf den Markt. Mittlerweile ist Android bei Version 4.4 *Kit Kat* (API Level 19) angelangt, es sind mehr als 900 Millionen Android Geräte aktiviert und der *Play Store* hält mehr als 975.000 Apps zum Download bereit [15]. Damit ist auch der langjährige Spitzenreiter Apple übertrumpft [12].

Eine Besonderheit von Android ist, dass verschiedenste Versionen im Umlauf sind und aktuell genutzt werden, was die Entwicklung aufwändiger macht. Jedoch ist positiv zu vermerken, wie in Grafik 1.1 ersichtlich, dass der Großteil der Android-Nutzer *Jelly Bean* (API Level 16-18) verwendet, welches als relativ aktuell betrachtet werden kann.

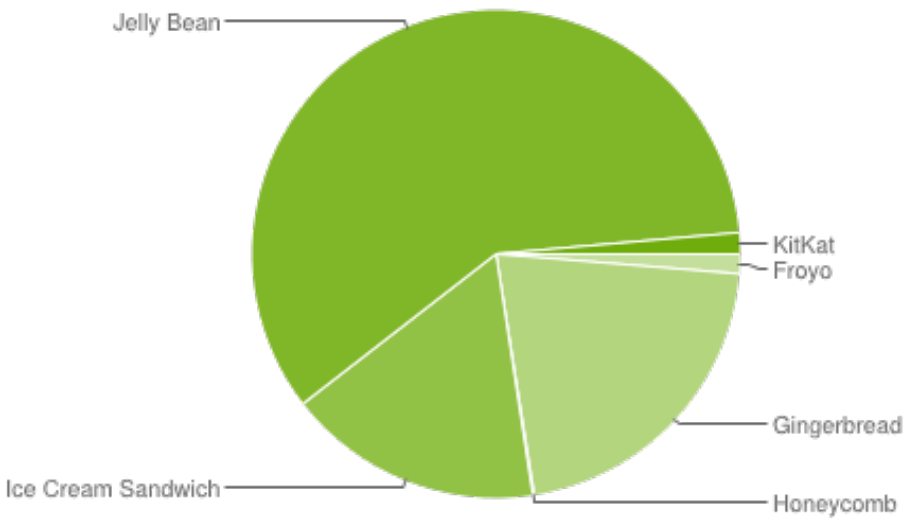


Abbildung 1.1: Android Platform Versionen [16]

Zu der Statistik anzumerken ist, dass Versionen vor *Froyo* (API Level 8) nicht berücksichtigt wurden, weil die Statistik mit Hilfe der neuen *Play Store*-App durchgeführt wurde und diese für die frühen Versionen nicht verfügbar ist. Die Daten wurden in einem 7-tägigen Zeitraum bis zum 8. Januar 2014 erhoben.

Eine weitere Besonderheit ist, dass Android auf unterschiedlichen Bildschirmgrößen mit unterschiedlichen Pixeldichten ausgeführt werden kann. Für Apps, die auf allen Bildschirm-Konfigurationen passend und übersichtlich aussehen, ist entsprechender Aufwand nötig. Graphik 1.2 und 1.3 zeigen aktuelle Statistiken bezüglich Displaygröße und Pixeldichte.

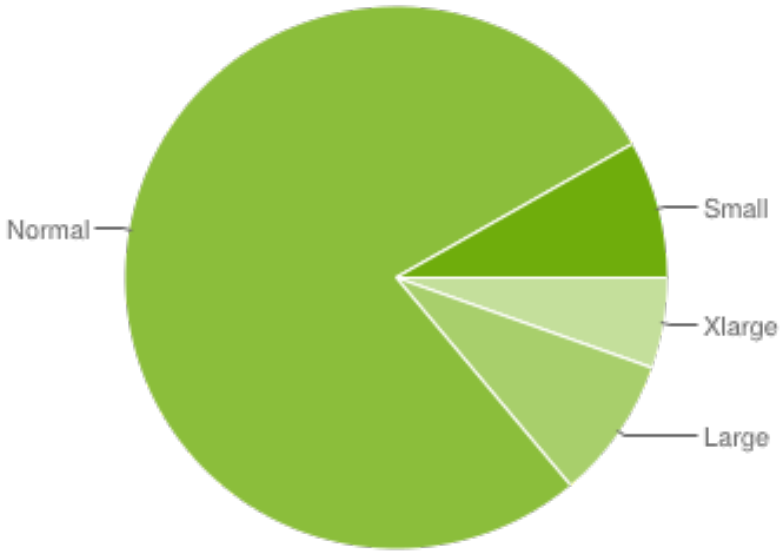


Abbildung 1.2: Android Bildschirmgrößen [16].

Bei den Bildschirmgrößen sieht man deutlich, dass Bildschirmgröße *Normal* überwiegt, während *Small*, *Large* und *Xlarge* haben etwa den gleichen Anteil haben. Bei der Kategorisierung handelt es sich um grobe Klassifizierungen von Geräten unabhängig von ihrer Pixeldichte.

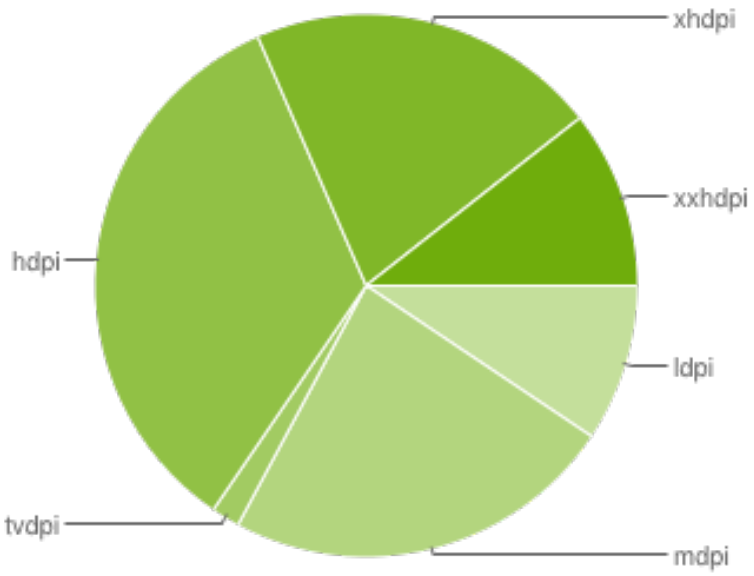


Abbildung 1.3: Android Pixeldichten [16].

Bei den Pixeldichten hat *hdpi* den größten Anteil mit etwa 34 %. *hdpi* entspricht einer *PPI* (pixels per inch) von 240. Jeweils *xhdpi* und *mdpi* sowie *xxhdpi* und *ldpi* haben ähnliche Anteile. Bei der Android Entwicklung kann dies beispielsweise berücksichtigt werden, indem verschieden große Bildressourcen für unterschiedliche Bildschirmdichten zur Verfügung gestellt werden. Die Entwicklung der letzten Jahre hat gezeigt, dass die Bildschirme und Pixeldichten der High-End Geräte stetig größer wurden. Deshalb ist in den nächsten Jahren zu erwarten, dass der Anteil der niedrigen Pixeldichten und kleinen Displays immer kleiner wird. Bei der Entwicklung von *AppEditor* wurde darauf geachtet, dass die Darstellung auf verschiedensten Bildschirmgrößen passend und übersichtlich ist. Dazu später mehr in Kapitel 5.1.

1.2 Architektur von Android

Android kann als Software-Stack betrachtet werden, welcher sich aus vier wesentlichen Teilen zusammensetzt: *Linux Kernel*, *Libraries* und *Runtime*, *Application Framework* und *Applications* [17]. In Abbildung 1.4 ist die Hierarchie der Komponenten zu erkennen. Als Basis nutzt Android einen Linux Kernel, der für die speziellen Ansprüche eines mobilen Gerätes optimiert wurde. Dazu gehört auch, dass einige Funktionen eines Desktop-Linux, wie beispielsweise das *X windowing system*, nicht übernommen wurden.

Andererseits wurden Verbesserungen des Kernels erreicht, welche jetzt auch von der Linux Community in die nächste Version übernommen werden [17]. Linux als Basis eignet

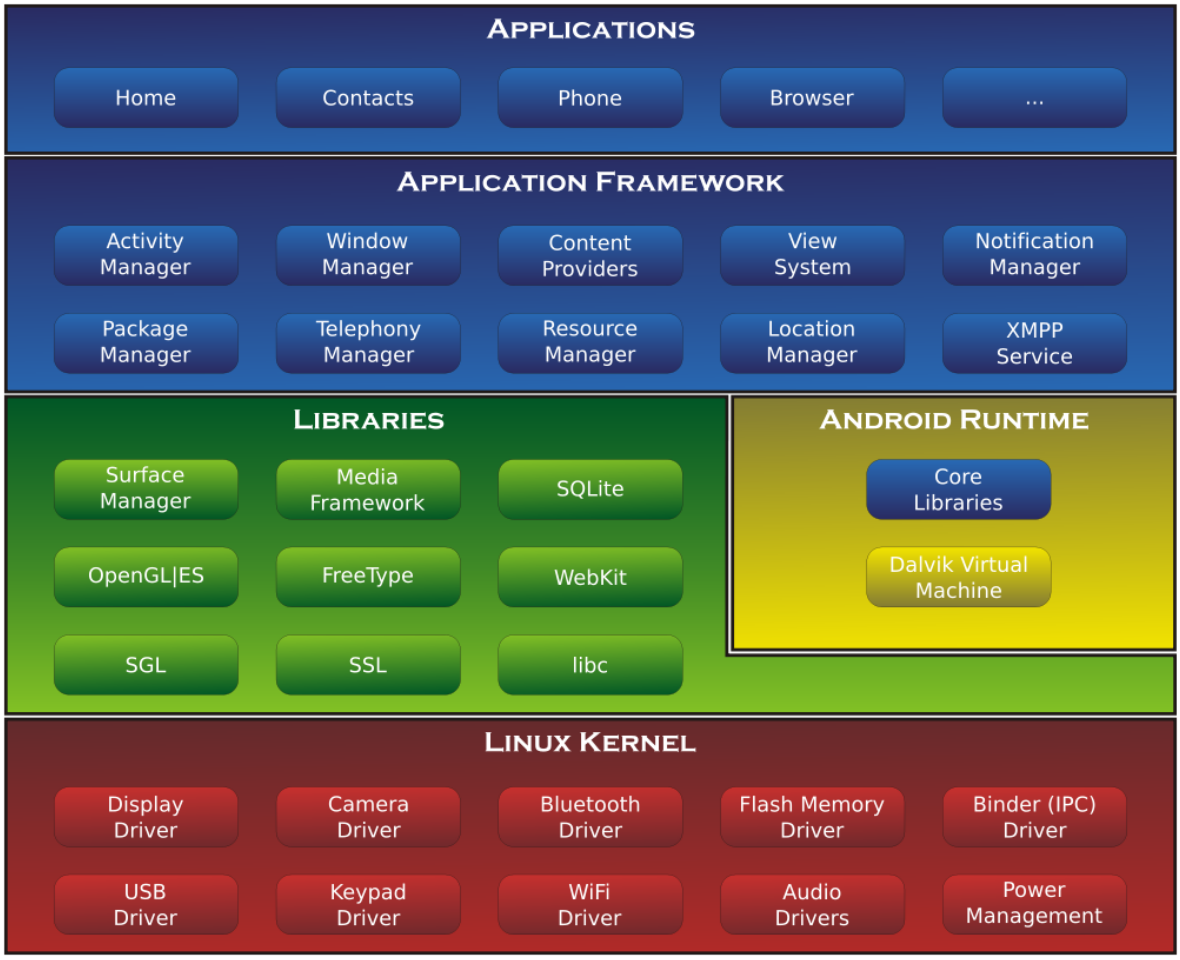


Abbildung 1.4: Android Architektur (Quelle: `http://commons.wikimedia.org/wiki/File%3AAndroid-System-Architecture.svg`)

sich für Android besonders gut wegen seiner Hardware Abstraction, Treibersicherheit und wegen seines Prozess-/Speichermanagements. Gerade die Hardware Abstraction macht es Hardware-Herstellern besonders leicht, Android auf ihren Geräten zu implementieren. Eine besondere Komponente, die speziell für Android entwickelt wurde, ist der sogenannte *Binder*, welcher Inter-process Communication (IPC) mit Hilfe von Shared Memory vereinfacht. Das spielt in Android eine besondere Rolle, weil hier jede App durch einen eigenen Prozess ausgeführt wird [3]. Weitere speziell angepasst Kernel Module sind Power Management, Alarm, Low Memory Killer und der Logger [17].

Auf die Kernel-Schicht setzt die *Library*-Schicht auf, auf welche mittels des Android *Application Frameworks* zugegriffen wird. Hierbei handelt es sich um hardwarenahe C/C++ libraries, die oft ohne große Modifikationen übernommen wurden (SSL, SQLite) [17].

Auf gleicher Schicht ist auch die *Android Runtime* angesiedelt, welche aus *Core Libraries* und der DVM besteht. An dieser Stelle sei nur kurz erwähnt, dass die DVM eine spe-

zielle Java Virtual Machine (JVM) ist. Die *Core Libraries* simulieren zum Teil klassische Java Application programming interface (API), stellen aber auch direkten Zugriff auf beispielsweise SQLite oder OpenGL bereit. In Kapitel 1.3 wird im Speziellen auf die DVM eingegangen.

Auf *Libraries* und *Android Runtime* baut die *Application Framework*-Schicht auf. Hier werden APIs bereitgestellt, die direkt in der App Programmierung verwendet werden können. Auf die wichtigsten Services sei hier kurz eingegangen: Der *Activity Manager* regelt das *Lifecycle* der Apps und ermöglicht beispielsweise den Start anderer Activities. Der *Resource Manager* erlaubt den vereinfachten Zugriff auf Ressourcen wie Strings, Graphiken und verschiedene XMl-Ressourcen. Der *Location Manager* regelt den Umgang mit Positions-Updates. Hier können Listener angegeben werden, welche auf Positions-Updates reagieren. Mit dem *Notification Manager* kann einerseits auf bestimmte Systemereignisse, wie beispielsweise das Ankommen einer SMS, reagiert werden. Andererseits können auch eigene Benachrichtigungen abgegeben werden. Der *Package Manager* kümmert sich um die Verwaltung von Apps auf dem Gerät, wozu unter anderem das Installieren/Desinstallieren von neuen Apps sowie das Auslesen von Informationen wie Namen oder Icon gehört. Das *View System* stellt eine Reihe von User interface (UI)-Komponenten zur Verfügung, die vom Entwickler genutzt werden können. Mit Hilfe der *Content Providers* kann man anderen Apps den Zugriff auf die Daten der eigenen App ermöglichen, umgekehrt aber auch auf die Daten anderer Apps zugreifen. Neben diesen Komponenten existieren noch eine Reihe anderer, auf die in diesem Zusammenhang aber nicht weiter eingegangen wird.

Die oberste Schicht sind die Apps selbst. Android bringt schon von vornherein Apps mit, dazu gehören zum Beispiel Maps, SMS-Programm, Kalender, Browser und Kontakte Apps. Die Benutzer kann über den *Play Store* verschiedene andere, auch kostenpflichtige, Apps installieren, die dann gleichberechtigt mit bereits installierten Apps auf dem System existieren.

1.3 Dalvik Virtual Machine

Für die Ausführung von Apps, welche als Android application package file (APK) vorliegen, wird in Android die DVM verwendet. Dabei handelt es sich im Gegensatz zur JVM, welche auf einem Kellerautomaten basiert, um eine Registermaschine. Diese Architektur wurde gewählt, um möglichst performant auf modernen Prozessoren zu operieren [4].

Trotzdem können Android-Apps in Java programmiert werden, was den Vorteil hat, dass vorhandene Entwicklungsumgebungen genutzt werden können. Außerdem ist die Verbreitung von Java sehr groß, was Android als Plattform für viele Entwickler besonders interessant macht. Die Arbeit vieler Entwickler führt zu einer großen Auswahl an Apps, was maßgeblich die Attraktivität des Betriebssystems für Endbenutzer bestimmt. Trotz der Programmierung in Java reicht der übliche Java-Compiler nicht aus. In Abbildung 1.5 ist der *Build*-Prozess vereinfacht beschrieben.

In diesem Zusammenhang soll nur kurz darauf aufmerksam gemacht werden, dass das Erstellen einer Android-App deutlich aufwändiger ist als das normale Java-Compiling, was nur einem Schritt im gesamten Prozess entspricht. In Kapitel 3.1 wird im Detail auf den

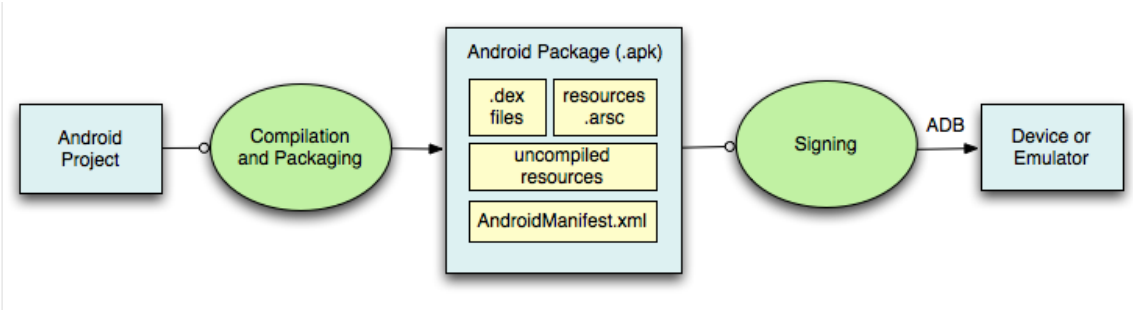


Abbildung 1.5: Vereinfachter Android Building Process [16]

Build-Prozess eingegangen.

Eine weitere Besonderheit ist, dass jede Android-App in einer eigenen DVM und damit als eigener low-level Linux Prozess unter einem eigenen Betriebssystem-Benutzer läuft. Dadurch sind private Daten der App durch das Betriebssystem und die Sandbox der DVM geschützt.

Abschließend soll noch kurz auf den Boot-Vorgang eingegangen werden. Ähnlich wie bei normalen Linux Betriebssystemen wird beim Booten der Kernel in den Hauptspeicher und der *init*-Prozess gestartet. Dieser wiederum startet verschiedene *Daemons*: zum Beispiel die Android Debug Bridge (adb) oder den USB Daemon. Sind alle Daemons geladen, wird der Service Manager, der die Verbindung zum Kernel herstellt, und *zygote*, der Mutterprozess aller Apps, gestartet.

zygote startet direkt die erste *DVM* und lädt wichtige Klassen. Daraufhin wartet *Zygote* auf kommende App-Aufrufe. Der erste *fork* des *zygote*-Prozesses erfolgt automatisch, um den *System Server* zu starten. Dieser kümmert sich um das Starten aller grundlegenden Android Services. Danach können Apps vom Benutzer gestartet werden. Beim Start einer App wird ein *fork* des *zygote*-Prozesses initiiert, welcher auch die bereits erstellte DVM erbt [17]. Dieser Ablauf wird anschaulich in Abbildung 1.6 dargestellt, wobei die eingekreisten Ziffern die zeitliche Reihenfolge wiedergeben.

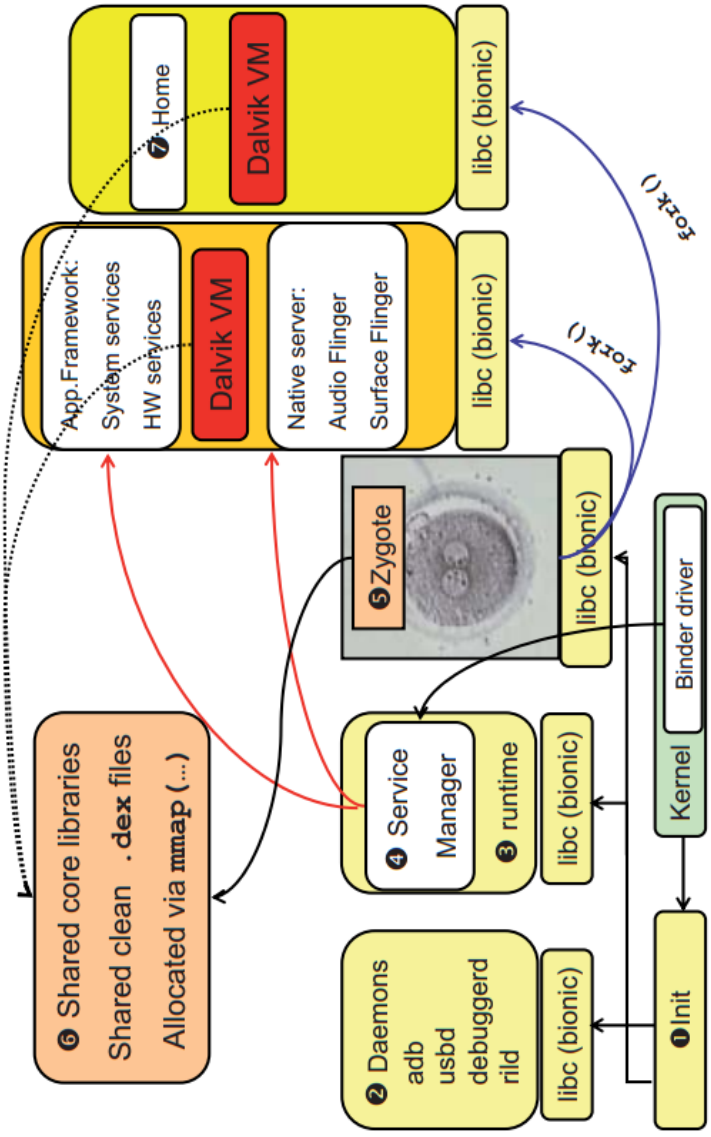


Abbildung 1.6: Android *init* Ablauf [5]

2 Allgemeines zu Reengineering

Ausgehend von einem kurzen Abschnitt über die Entwicklung des Reengineerings allgemein wird auf rechtliche, aber auch moralische Aspekte des Reengineerings eingegangen. Abschließend werden grundlegende Schutzmethoden gegen Decompiling erläutert.

2.1 Entwicklung des Decompiling

Der erste bekannte Decompiler *D-Neliac* wurde bereis im Jahre 1960 für die Programmiersprache *ALGOL* geschrieben [6, 18]. Er wurde von Joel Donnelly und Herman Englander in den U.S. Navy Electronic Labs (NEL) entwickelt. Die Hauptaufgabe von D-Neliac war es, vorhandene Programme für *Neliac* (Navy Electronics Laboratory International *ALGOL* Compiler) zu konvertieren. Auch in den folgenden Jahren wurden Decompilers beispielsweise für *COBOL*, *Ada* oder *Fortan* entwickelt. Hauptgrund war meistens, Software auf anderer Hardware ausführbar zu machen. Aber es gab auch andere Gründe für Decompiling. Zur Jahrtausendwende (*Y2K*) mussten zahlreiche ältere Programme dekompiliert werden, um die weitere Ausführbarkeit zu gewährleisten. Ähnliche Situationen waren zum Beispiel das Erreichen der 10000 Marke des Dow Jones oder die Einführung des Euros in der EU. Auch hier kamen zahlreiche Finanz-Programme aus dem Tritt und mussten entsprechend angepasst werden.
Der erste Java-Decompiler wurde mit großer Wahrscheinlichkeit von Daniel Ford intern bei IBM im Mai 1996 geschrieben und hieß *Jive*. Kurz darauf, im Juli 1996, wurde der bekannte Java-Decompiler *Mocha* von Hanpeter van Vliet veröffentlicht. Dass der Decompiler zum freien Download auf seiner Website zur Verfügung stand, sorgte damals für eine große Kontroverse. Daraufhin nahm ihn van Vliet vom Netz und veröffentlichte ihn erst wieder, nachdem eine große Mehrheit für die erneute Veröffentlichung abgestimmt hatte [18].

Neben den bisher aufgeführten Einsatzzwecken von Decompiling gibt es noch einige andere Möglichkeiten, Decompiling zu nutzen[6]:

- Decompilierung von fremden Quellcode, um bestimmte Algorithmen nachzuvollziehen.
- Wiederherstellung von Quellcode, wenn der Original-Quellcode verloren gegangen ist.
- Herausfinden, ob Programme durch Malware verseucht sind.
- Programme, die in einer veralteten Sprache geschrieben wurden, in eine neue Sprache übersetzen.

2.2 Rechtliche Aspekte

Nicht alle der aufgeführten Einsatzzwecke sind legal. Häufig wurde Decompiling auch genutzt, um bestimmte Schutzmechanismen auszuhebeln und so beispielsweise eine Testversion eines Programms zu einer Vollversion zu machen. Der Entwickler der Software hat unterschiedliche Möglichkeiten, das Decompiling seines Produkts zu unterbinden. Dazu gehören Patentgesetze, Urheberrecht, Lizenzvereinbarungen, *Anti-Reverse-Engineering* Klauseln oder Gesetze wie das Digital Millennium Copyright Act (DMCA). Mögliche tolerierte Ausnahmen (in manchen Ländern) sind:

- Decompilierung, um die Kompatiblität des Programms zu sichern.
- Decompilierung zur Bug-Beseitigung, wenn der ursprüngliche Entwickler dazu nicht mehr zur Verfügung steht.

Trotzdem ist hier immer Vorsicht geboten: Von Land zu Land sind die Gesetze etwas unterschiedlich. Im Zweifelsfall sollte immer ein Rechtsanwalt konsultiert werden. Da der Schwerpunkt dieser Arbeit nicht der rechtliche Aspekt ist, soll dieses Thema hier nicht weiter vertieft werden. In [18] sind ein paar grundsätzliche rechtliche Tipps im Bezug auf Android zu finden, die man unbedingt einhalten sollte:

- Apps sollten nicht dekompiliert, rekompiliert und anschließend als eigenes geistiges Eigentum verbreitet werden.
- Durch Recompiling veränderte Apps sollten niemals an Dritte verkauft werden.
- Apps, deren Lizenzvereinbarung das Decompiling verbietet, sollten auch wirklich nicht dekompiliert werden.
- Mittels Decompiling und Recompiling sollten niemals vorhandene Schutzmechanismen entfernt werden, auch nicht für den persönlichen Gebrauch.

2.3 Moralische Aspekte

Neben den ganzen formalen, rechtlichen Beschränkungen, sollte auch die moralische Seite des Decompiling betrachtet werden. Diese Arbeit sollte allein für wissenschaftliche Zwecke genutzt werden. Mittels Decompiling lässt sich besseres Verständnis für die Architektur der DVM und Android allgemein erwerben. Allein das sollte die Motivation für Decompiling in diesem Zusammenhang sein und eben nicht der Diebstahl von geistigem Eigentum Anderer.
Außerdem möchte sich der Autor klar von der illegalen Nutzung der in diesem Zusammenhang entwickelten App distanzieren.

2.4 Grundlegender Schutz gegen Decompiling

Obwohl im Speziellen Android-Apps relativ leicht zu dekompilieren sind, gibt es doch einige Gegenmaßnahmen. Diese sind jedoch unterschiedlich effektiv [18]:

- *Obfuscation*: Verändert unter anderem Methoden- und Klassennamen und macht es sehr schwierig, den Code nachzuvollziehen. Die Güte hängt dabei vom verwendeten Tool ab.
- *Intellectual Property Rights (IPR) Schutzmaßnahmen*: Mittels Google Play Digital Rights Management (DRM) (Application Licensing) kann überprüft werden, ob der Benutzer eine bestimmte App nutzen darf. Doch diese Maßnahme bietet keine absolute Sicherheit, denn mit modifizierter Software lässt sich dieser Schutz umgehen.
- *Server-side Code*: Bei diesem Ansatz wird Funktionalität auf einen externen Server ausgelagert. Die App sendet lediglich eine Anfrage und bekommt ein entsprechendes Ergebnis. Algorithmen können auf diese Weise versteckt werden. Hierbei problematisch ist, dass die Zugriffsdaten für den Server trotzdem irgendwo im App-Quellcode auftauchen werden. Ein weiterer großer Nachteil ist, dass die App dann nur noch mit Internetzugang funktioniert und entsprechende Rechte im Manifest deklariert werden müssen.
- *Nutzung des Android Native Development Kit (NDK)*: Wichtige Programmteile können als C++-Code ausgelagert werden. Dieser ist ungemein schwieriger zu entschlüsseln als Java Code. Dieser Ansatz gehört definitiv zu den effektivsten hier vorgestellten, aber auch zu den aufwändigsten.
- *Verschlüsselung*: Gerade zusammen mit der Nutzung des NDKs kann mit Verschlüsselung das Decompiling erschwert werden. Es kann aber auch zum sicheren Schlüsselaustausch mit Webservern verwendet werden.

3 Reengineering von Android Apps

Nachdem auf Grundsätzliches über Android und Reengineering eingegangen wurde, soll nun im Speziellen das Decompiling von Android-Apps untersucht werden. Anschließend werden Sicherheitskonzepte von Android erklärt.

3.1 Android App-Dateiformat

Eine Android App ist ein APK-File, ein spezielles Archiv-File, das vom typischen *Java Archive (JAR)*-Format abstammt. Diesem wiederum liegt das *ZIP*-Format zu Grunde. Ein Android APK File ist also im Prinzip nichts anderes als ein Container für verschiedene Dateien und Ordner. Er beinhaltet alle Dateien, die zur Installation und Nutzung der App notwendig sind. Das APK-File hat immer folgenden Aufbau:

- **assets:** Enthält Dateianlagen, auf die im Code mit Hilfe des *AssetManager* zugegriffen werden kann.
- **lib:** Beinhaltet kompilierten Code, der als Bibliothek durch die App genutzt werden kann. Der Ordner enthält Unterordner, die jeweils verschiedenen Systemarchitekturen entsprechen, zum Beispiel *armeabi* für *ARM* basierte Architekturen oder *x86* für Systeme mit *x86* Architektur.
- **META-INF:** Enthält Dateien, die zur Verifizierung der App dienen. Dazu gehören:
  - *MANIFEST.MF:* Datei, in der die Inhalte der APK-Datei aufgelistet sind.
  - *CERT.RSA:* Der Zertifikat des Entwicklers. Während der Entwicklung kann hier auch ein Test-Zertifikat verwendet werden.
  - *CERT.SF:* Enthält zu jeder Datei, die in *Manifest.MF* gelistet ist, einen *SHA-1* Hash.
- **res:** Enthält unkompilierte Ressourcen, die im Quellcode verwendet werden können.
- **AndroidManifest.xml:** Eine zusätzliche Android Manifest-Datei, welche unbedingt mit exakt diesem Namen enthalten sein muss. Diese Datei wird vom *PackageManager* bei der Installation der App benötigt. Zu den wichtigsten enthaltenen Informationen gehören [8]:
  - der Name des Root Java-Package. Dieser muss einmalig sein, weil er später zur Identifizierung der App genutzt wird.
  - Details zu allen verwendeten Komponenten der App, also *Activities*, *Services*, *Broadcast Receivers* und *Content Providers*, und der Zusammenhang mit den implementierenden Java-Klassen.

  - die von der App benötigten Erlaubnisse. Beispielsweise Zugriff auf Kamera, Kontakte oder Internet.
  - die minimale API Version, die zum Ausführen der App benötigt wird.
  - die Bibliotheken, welche zur Ausführung vorhanden sein müssen.
- **classes.dex:** Der gesamte in Byte-Code kompilierte Java-Quellcode befindet sich in dieser Datei. Er kann direkt von der DVM verarbeitet werden.
- **resources.arsc:** In diesem Container befinden sich alle kompilierten Ressourcen. Zum Beispiel binäre XML Layout Dateien.

Der Kompilier-Verfahren einer APK-Datei ist mehrstufig und enthält die in Abbildung 3.1 aufgeführten Stufen. Normalerweise geschieht der Ablauf in einer Integrated development environment (IDE) wie *Eclipse* vollautomatisch und der Entwickler muss sich keine Gedanken über die einzelnen Schritte machen. Da sich *AppEditor* aber hauptsächlich mit Compiling und Decompiling von *APK*-Dateien beschäftigt, sei hier der genaue Ablauf anhand der Abbildung erklärt:

1. Das Android Asset Packaging Tool (aapt) wird ausgeführt. Es kompiliert sowohl die App-Ressourcen als auch die Datei `AndroidManifest.xml`. Außerdem wird die Klasse `R.java` erzeugt. Damit kann im Java-Code direkt auf die *App*-Ressourcen zugegriffen werden.
2. Vorhandene Android Interface Definition Language (AIDL) Schnittstellen, welche zur IPC zwischen Android Apps verwendet werden, werden in Java Schnittstellen konvertiert.
3. Der Java-Compiler kompiliert den Java-Quellcode zusammen mit den zuvor erzeugten Schnittstellen und `R.java` zu den herkömmlichen *.class*-Files.
4. Mit dem Tool *dex* wird aus den Java *.class*-Files und zusätzlich verwendeten Bibliotheken *Dalvik Byte Code* erzeugt und in *.dex*-Files gespeichert, welche später von der DVM verarbeitet werden können.
5. Jetzt werden mit dem *apkbuilder* kompilierte Ressourcen, nicht-kompilierte Ressourcen und *.dex*-Dateien in die APK-Datei gepackt.
6. Damit die App auf einem Android-Device installiert werden kann, muss sie noch mit dem *jarsigner*-Tool, welches im Java Development Kit (JDK) enthalten ist, signiert werden. Dabei ist die Signatur mit einem *debug*- oder *release*-Key zu unterscheiden. Für die Veröffentlichung im Play-Store muss sie mit dem *release*-Key signiert werden.
7. Im letzten Schritt wird das *zipalign*-Tool noch auf die APK-Datei angewendet, um die Speicherauslastung zu optimieren.

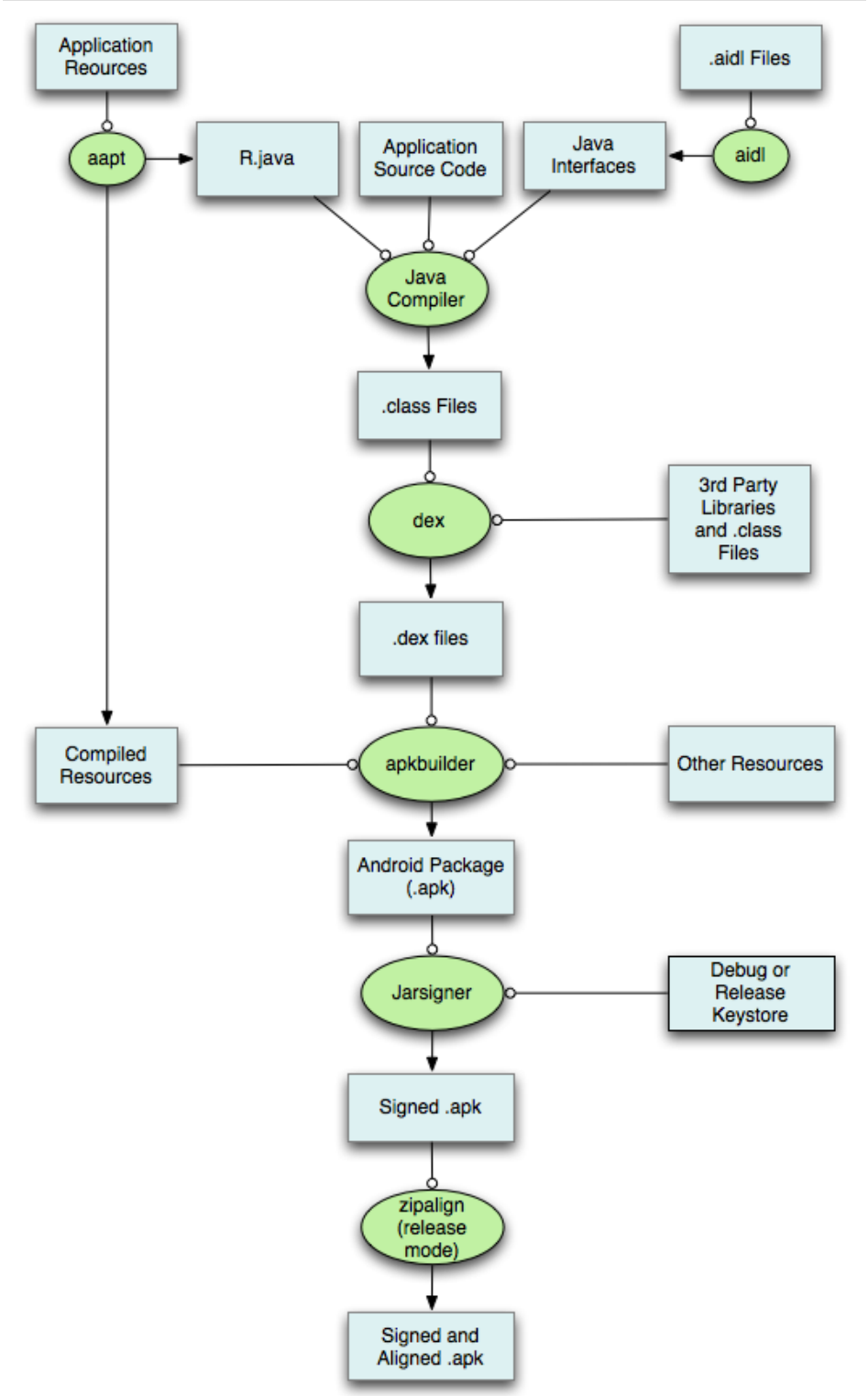


Abbildung 3.1: Detaillierter Android Building Process [16].

3.2 Sicherheit von Android

In Kapitel 1.3 wurde bereits auf einige Sicherheitsfeatures von Android eingegangen. Im folgenden Teil werden weitere Sicherheitsmerkmale von Android erläutert. Dabei wird zwischen Sicherheit auf System- und App-Ebene unterschieden.

3.2.1 Sicherheit auf System-Ebene

Wie bereits bekannt, verwendet Android einen Linux-Kernel als Grundlage. Dieser stellt bereits grundsätzliche Sicherheitsfeatures auf System-Ebene bereit. Dazu gehören [1]:

- ein benutzerbasiertes Berechtigungsmodell
- Isolation einzelner Prozesse voneinander
- sichere IPC
- die Möglichkeit, unnötige und womöglich unsichere Teile des Kernels zu entfernen

Android optimiert die Sicherheits des normalen Linux-Kernels noch, indem es es jeder Applikation eine eigene user ID (UID) zuweist. Dadurch entsteht eine Applikation-Sandbox auf System-Ebene. Wenn also eine Applikation A ohne Weiteres auf die Daten der Applikation B zugreifen will, wird das auf System-Ebene verhindert. Ein anderer Vorteil der Sandbox ist, dass bei Speicherfehlern einer App nicht das ganze System betroffen ist.
Die Sandbox gilt auch für nativen Code und System-Applikationen. Dadurch haben Java-Applikationen und Applikationen, die das NDK, also C-Code, nutzen, das gleiche Sicherheitslevel. Natürlich kann auch diese Sandbox nicht hundertprozentigen Schutz bieten. Der Schutz auf Kernel-Ebene gewährleistet aber ein vergleichsweise hohes Sicherheitsniveau [1].

3.2.2 Sicherheit auf Applikations-Ebene

Durch den Einsatz der gerade beschriebenen Sandbox hat eine Android Applikation à priori beschränkten Zugriff auf System Ressourcen. Android regelt den Zugriff auf sensible APIs mit dem *Android Permission Model*. Folgende Funktionen sind in sensiblen APIs enthalten [1]:

- Kamera-Funktionalität
- Positionsbestimmung (GPS)
- Bluetooth Funktionalität
- Telefon Funktionen
- SMS/MMS Senden und Empfangen
- Netzwerk- und Datenverbindungen
- Zugriff auf persönliche Daten

Eine App kann sich Zugriff auf geschützte APIs verschaffen, indem sie diese im Android Manifest deklariert. Bei der Installation der Applikation wird dem Benutzer angezeigt, auf

welche sensiblen Funktionen zugegriffen werden soll. Der Benutzer hat dann die Möglichkeit, alle Forderungen zu akzeptieren und mit der Installation der Applikation fortzufahren oder die Installation an dieser Stelle abzubrechen. In Abbildung 3.2 soll die App *Sochi 2014 Results* installiert werden. Es wird unter anderem Zugriff auf personenbezogene Daten, Internet und Speicher gefordert. Wenn der Nutzer damit einverstanden ist, kann er die Installation mit Klick auf „Akzeptieren“ fortsetzen.

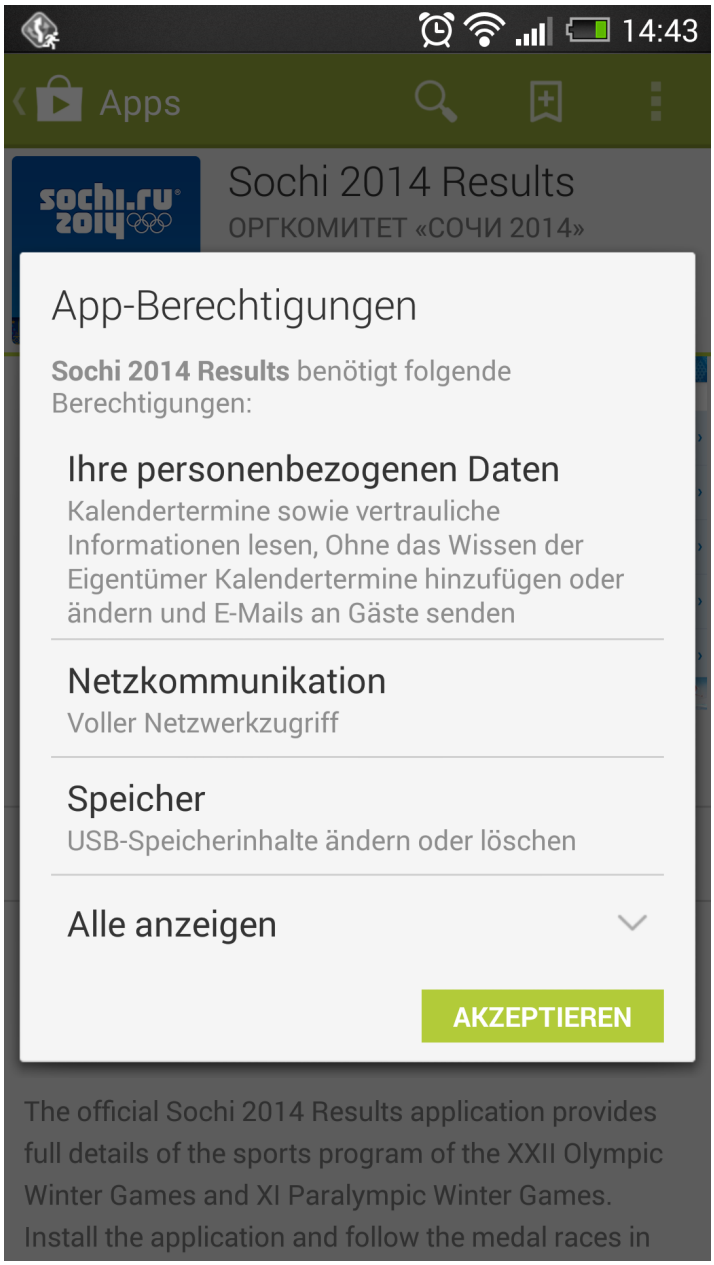


Abbildung 3.2: Anzeige der benötigten Berechtigungen bei der Installation von *Sochi 2014 Results*.

Die Applikationen behält die Zugriffsrechte nach erfolgreicher Installation dauerhaft. Wenn eine App im Code sensible Funktionen benutzen will, diese aber nicht im Manifest deklariert hat, kommt es zu einer `Exception`, was meist in einem Absturz der App endet.

4 Verwandte Arbeiten und Projekte

Auch andere Projekte haben sich mit Decompiling und Analyse von Android-Apps beschäftigt. Relevante Arbeiten sind nachfolgend vorgestellt. *Dexplorer* und *AppGuard* sind Android-Applikationen, während *Baksmali/Smali* und *Apktool* als Desktop-Programme konzipiert wurden.

4.1 Dexplorer

Mit *Dexplorer* lassen sich APK-Dateien direkt auf dem Endgerät untersuchen. Dabei wird `classes.dex` in Java-Code dekompiliert, wobei Methoden-Rümpfe nicht sichtbar sind. Außerdem lassen sich die Applikations-Ressourcen in den Ordnern *assets* und *res* betrachten.

Im ersten Schritt kann aus einer Übersicht aller installierten Apps die zu analysierende App ausgewählt werden, wie in Abbildung 4.1 zu sehen ist.

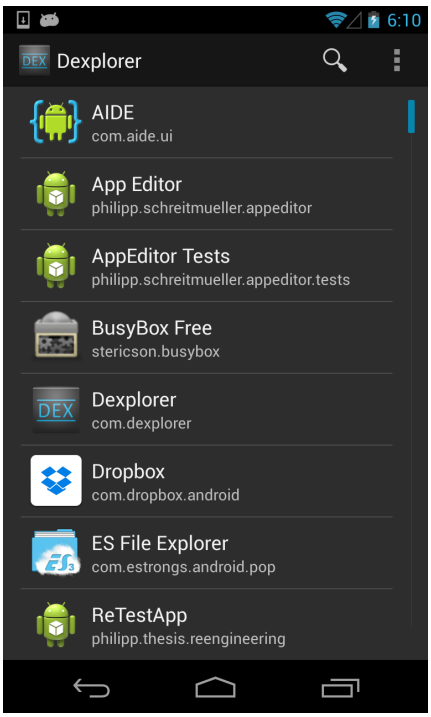


Abbildung 4.1: Startbildschirm von *Dexplorer*.

Nachdem der DVM Byte Code und die Ressourcen dekompiliert wurden, gelangt man auf einen Filebrowser (siehe Abbildung 4.2). Hier kann man sich einen Überblick über die verwendeten Java-Packages verschaffen und die Ordnerhierarchie des *res*-Ordners untersuchen.

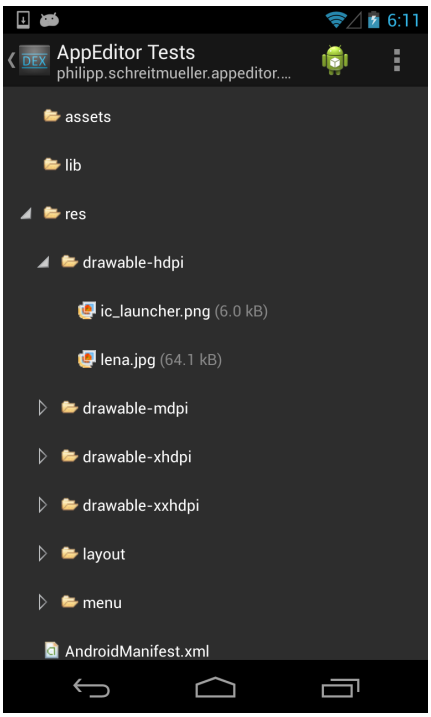


Abbildung 4.2: *Dexplorer* Übersicht über den Inhalt eine APK-Datei.

Beim Klick auf die gewünschte Datei werden diese in einer Detailansicht geöffnet. Bei Bilddateien wird eine Vorschau sowie Bilddetails angezeigt. Bei XML- und Java-Dateien wird der Quellcode mittels *Syntax-Highlighting* übersichtlich präsentiert. Abbildung 4.3 zeigt die Detailansicht einer XML-Layoutdatei.

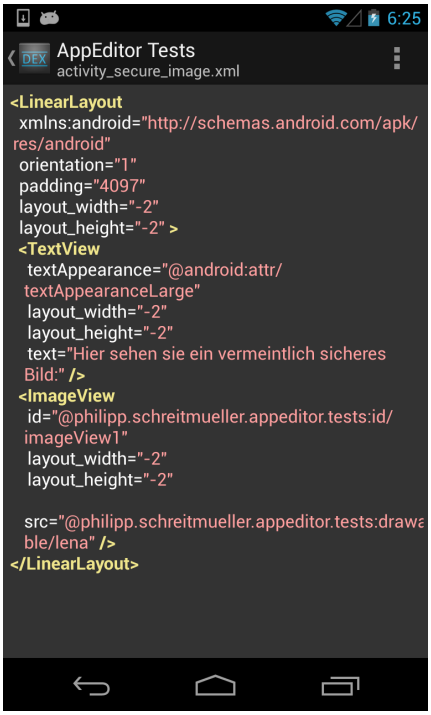


Abbildung 4.3: *Dexplorer* Detailansicht einer XML-Layoutdatei.

Auf größeren Bildschirmen wird Filebrowser und Detailansicht nebeneinander in einem *twopane*-Modus dargestellt. Die Angaben beziehen sich auf die Version 1.0.3 vom 4. November 2013. Die App ist erhältlich unter [1].

4.2 SRT AppGuard

AppGuard von *backes:SRT* ist eine App zur Überwachung anderer Apps. Zu den Schlüsselfunktionen gehört das dynamische Rechtemanagement installierter Applikationen ohne dass Root-Rechte nötig sind [2]. Man kann also der überwachten App bestimmte, angeforderte *Permissions* verwehren und sie trotzdem verwenden. Das ist mit dem normalen *PackageManager* nicht möglich. Um diese Funktionalität bei bereits installierten Apps zu nutzen, müssen diese zuerst deinstalliert werden. Danach wird eine modifizierte Variante durch *AppGuard* installiert, die jedoch weiter Updates über *Google Play* erhält. Dabei ist zu beachten, dass gespeicherte Daten der App beim Deinstallieren verloren gehen. Außerdem bietet *AppGuard* detaillierte Logs der überwachten Apps. In Abbildung 4.4 ist die Detailansicht einer App in *AppGuard* abgebildet. Hier können

[1] `https://play.google.com/store/apps/details?id=com.dexplorer&hl=de`

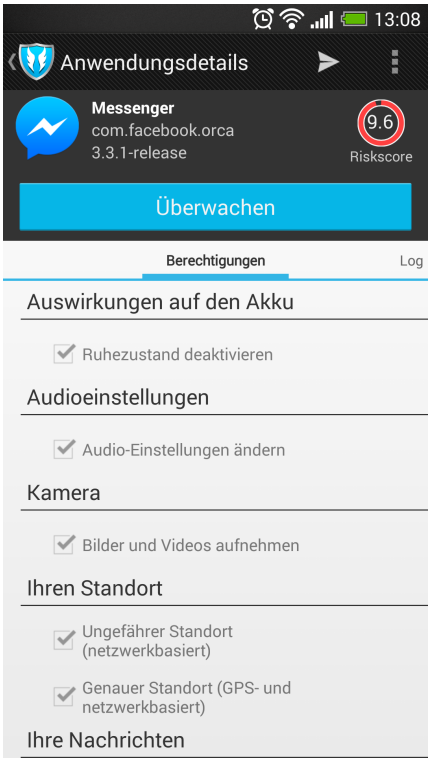


Abbildung 4.4: *AppGuard* Detailansicht einer App.

Auf dem Startbildschirm bietet *AppGuard* eine Übersicht aller installierter Apps und gibt zu jeder einen *Riskscore* an, der aufschlüsselt, welches Gefahrenpotenzial die App mit den von ihr geforderten Berechtigungen hat (siehe Abbildung 4.5).

Die hier gemachten Angaben zu *AppGuard* beziehen sich auf Version 2.1.10. Diese kann von der Website direkt heruntergeladen werden [2]. Es gibt eine kostenlose eingeschränkte Version und eine Pro-Version mit vollem Funktionsumfang für 3,99 €. *AppGuard* war ursprünglich auch über den *Play Store* erhältlich. Auf Anfrage bei *backes:SRT* erklärte Sven Obser, dass die App mit Verweis auf die AGBs des *Play Stores* von Google entfernt wurde. Ein Versuch der Kontaktaufnahme mit Google war nicht erfolgreich.

4.3 Baksmali/Smali

Bei *Smali/ Baksmali* handelt es sich um einen Assembler beziehungsweise Disassembler für das *dex*-Dateiformat, also dem DVM-Bytecode. Die entwickelte App *AppEditor* ermöglicht die Bearbeitung des Smali Codes der App. Das Projekt ist unter der *New BSD License* veröffentlicht [3]. Der Syntax ist dabei an Jasmin, einem JVM Assembler, angelehnt. Nachfolgend soll ein Überblick über die Smali-Syntax gegeben werden.

[2] `http://www.srt-appguard.com/de/`
[3] `https://code.google.com/p/smali/`

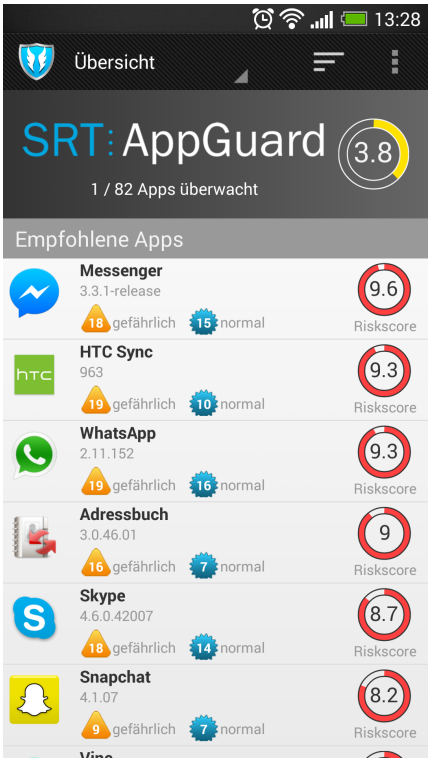


Abbildung 4.5: *AppGuard*-Übersicht aller Apps mit Riskscore.

Prinzipiell wird in Smali zwischen primitiven Typen und Referenztypen unterschieden [19]. Zu letzteren zählen Objekte und Arrays. Für primitive Typen werden folgende Abkürzungen verwendet:

Abkürzung	**Datentyp**
V	void (kann nur als Rückgabetyp verwendet werden)
Z	boolean
B	byte
S	short
C	char
I	int
J	long (64 bits)
F	float
D	double (64 bits)

Tabelle 4.1: Primitive Typen in Smali.

Objekte haben die Form `Lpackage/name/ObjectName;`. Arrays werden mit `[` gekennzeichnet. `[I` ist also ein Integer Array. Methoden haben im Allgemein die Form `Lpackage/name/ObjectName;->MethodName(III)Z`
Im konkreten Fall erwartet die Methode drei Integer als Parameter und gibt einen Boolean-Wert zurück. In Smali können außerdem Register, die immer 32 Bit groß sind, adressiert

werden. v0-vX stehen für lokale Register und p0-pX für Parameter-Register. Nähere Informationen zu den möglichen Op-Codes sind unter [4] und [5] zu finden.
Im Folgenden soll ein einfaches Smali Beispiel mit zugehörigem Java-Code die Syntax besser verständlich machen.

```
1  package philipp.schreitmueller.appeditor.tests;
2
3  public class SmaliExample {
4    int a;
5
6    public SmaliExample(int x){
7      a=x;
8    }
9
10   public void plusDrei(){
11     a=a+3;
12   }
13   public boolean istDrei(){
14     return a==3;
15   }
16 }
```

Listing 4.1: SmaliExample.java

Wenn man *baksmali* auf diese Datei anwendet, erhält man folgendes Ergebnis:

```
1  .class public Lphilipp/schreitmueller/appeditor/tests/SmaliExample;
2  .super Ljava/lang/Object;
3  .source "SmaliExample.java"
4
5
6  # instance fields
7  .field a:I
8
9
10 # direct methods
11 .method public constructor <init>(I)V
12    .locals 0
13    .parameter "x"
14
15      # Objektreferenz wird in p0 geladen
16    .prologue
17    .line 6
18    invoke-direct {p0}, Ljava/lang/Object;-><init>()V
19
20      # Speichern des Parameters im Member a
21    .line 7
22    iput p1, p0, Lphilipp/schreitmueller/appeditor/tests/SmaliExample;->a:
      I
23
```

[4] http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html
[5] https://source.android.com/devices/tech/dalvik/instruction-formats.html

```
24      .line 8
25      return-void
26  .end method
27
28
29  # virtual methods
30  .method public istDrei()Z
31      .locals 2
32
33          # a wird nach v0 geladen
34      .prologue
35      .line 14
36      iget v0, p0, Lphilipp/schreitmueller/appeditor/tests/SmaliExample;->a:
        I
37
38          # 3 wird in das Register v1 geladen
39      const/4 v1, 0x3
40       # Falls v0 und v1 nicht gleich sein sollten, wird nach :cond_0
        gesprungen
41      if-ne v0, v1, :cond_0
42       # Wird dieser Teil ausgefuehrt sind v0 und v1 gleich. Es wird 1 in v0
        gespeichert
43      const/4 v0, 0x1
44
45          # Rueckgabe von v0 als Ergebnis der Methode
46      :goto_0
47      return v0
48
49          # Hier angekommen sind v0 und v1 ungleich, deshalb wird 0 in v0
        gespeichert
50      :cond_0
51      const/4 v0, 0x0
52
53      goto :goto_0
54  .end method
55
56  .method public plusDrei()V
57      .locals 1
58
59      .prologue
60      .line 11
61
62          # Laden des Members a in v0
63      iget v0, p0, Lphilipp/schreitmueller/appeditor/tests/SmaliExample;->a:
        I
64
65          # Addieren von 3 und v0. Ergebnis wird in v0 gespeichert
66      add-int/lit8 v0, v0, 0x3
67
68          # v0 wird zurueckgeschrieben in das Member a
69      iput v0, p0, Lphilipp/schreitmueller/appeditor/tests/SmaliExample;->a:
        I
70
```

```
71     .line 12
72     return-void
73 .end method
```

Listing 4.2: SmaliExample.smali

4.4 Apktool

Apktool ist ein nützliches Kommandozeilen-Programm, mit dem man Android Apps mit einem einzigen Befehl dekompilieren kann. Dabei verwendet es intern die in Kapitel 4.3 vorgestellten Programme *smali* und *baksmali*. Außerdem werden auch die Ressourcen aus dem File `resources.arsc` extrahiert und das Manifest-File lesbar gespeichert. Nach dem Dekompilieren können die Quelldateien beliebig bearbeitet und anschließend wieder zu einem APK-File kompiliert werden. Nachdem man die geänderte App noch signiert hat, kann sie direkt installiert werden. Ein simples Verwendungsbeispiel soll die einfache Bedienung demonstrieren [7]:

```
1 apktool d PFAD_ZUR_APK_DATEI
2 # Bearbeiten des Quellcodes
3 apktool b PFAD_ZUM_QUELLORDNER PFAD_ZUR_NEUEN_APK_DATEI
```

Listing 4.3: Beispielhafte Verwendung von apktool.

Neben der Möglichkeit, das *apktool* über die Kommandozeile zu nutzen, gibt es auch verschiedene Graphical user interface (GUI), die die Nutzung noch einfacher machen. Ein GUI ist unter [6] zu finden. *Apktool* ist unter *Apache License 2.0* veröffentlicht und unter [7] einsehbar.

[6] `http://forum.xda-developers.com/showthread.php?t=2326604`
[7] `https://code.google.com/p/android-apktool/`

5 Implementierung

In diesem Kapitel wird auf die Implementierung des Reengineeringtools *AppEditor* eingegangen. Nach der Beschreibung der implementierten Funktionen, der Zielplattform und der Modifikationen des *apktool*, wird die Architektur von *AppEditor* erklärt.

5.1 Implementierte Funktionalität

- **Decompiling von installierten Android-Apps:** Mit *AppEditor* ist es möglich, installierte APK-Dateien zu dekompilieren. Dabei wird `resources.arsc` und `classes.dex` in entsprechend dekompilierte, menschenlesbare Ressourcen und *Smali*-Code dekompiliert. Außerdem wird `AndroidManifest.xml` menschenlesbar. Eine kompilierte App wird als Projekt gespeichert und ist bei späterem Öffnen von *AppEditor* noch sichtbar. Die existierenden Projekte werden neben einer Liste aller installierten Apps direkt in der Start-Activity präsentiert.
- **Filebrowser für APK-Projekte:** Nachdem auf der Start-Activity ein APK-Projekt gewählt wurde, wird ein Filebrowser geöffnet, der alle dekompilierten Dateien in einer Baumdarstellung präsentiert.
- **Bild-Detailansicht:** Wenn im Filebrowser eine Bilddatei gewählt wurde, wird eine Detailansicht des Bildes geöffnet. Zugehörige Informationen wie Ausmaße und Dateigröße werden angezeigt. Außerdem besteht die Möglichkeit, das Bild in einem anderen Bildbetrachter zu öffnen oder die Datei zu teilen, um sie beispielsweise per Email zu versenden.
- **Textdatei-Detailansicht:** Neben dem Bildbetrachter wurde auch ein Editor für textbasierte Dateien entwickelt. Wird also beispielsweise eine XML Layout-Datei oder eine Smali-Datei im Filebrowser ausgewählt, wird diese im Editor angezeigt. Der Editor stellt Grundfunktionen wie Syntax-Highlighting oder *Undo/Redo* bereit. Außerdem kann eine Opcode-Nachschlagewerk für Smali-Dateien geöffnet werden. Auch für textbasierte Dateien besteht die Möglichkeit zum Teilen, beispielsweise per Dropbox oder Email.
- **Fragment-basierte Darstellung:** Die Darstellung des Filebrowser und der Detailansicht ist der Google Design Philosophie entsprechend umgesetzt worden [10]. Das bedeutet, dass Filebrowser und Detailansicht als Fragmente realisiert wurden. Auf Tablet-Bildschirmen werden beide Fragments nebeneinander angezeigt, während bei Smartphone-Bildschirmen Filebrowser und Detailansicht jeweils in eigenen Activities dargestellt werden. Abbildung 5.1 veranschaulicht die Umsetzung.

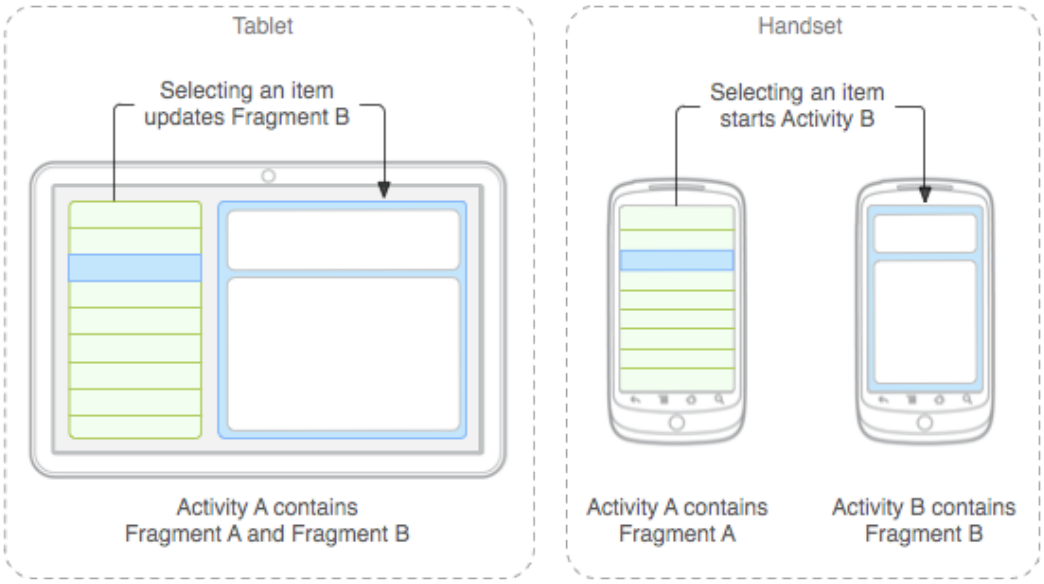


Abbildung 5.1: Android Fragment Konzept [10].

- **Teilen des Projekts als Zip-Datei:** Es wurde die Funktion implementiert, das komplette Projektverzeichnis zu zippen, um es zu teilen.
- **Kompilierung des APK-Projekts:** Nach der Bearbeitung des Quellcodes besteht die Möglichkeit, wieder eine funktionierende APK-Datei zu erzeugen.
- **Signatur der erzeugten APK-Datei:** Nachdem eine APK-Datei erzeugt wurde, wird sie direkt mit einem Testschlüssel signiert. Danach kann sie auf dem Gerät installiert werden oder anderen Apps geteilt werden.
- **Disclaimer und About:** Informationen zu *AppEditor* und seinen genutzten Bibliotheken sowie Anzeige eines Haftungsausschlusses.
- **Einstellungen:** Hier können Einstellungen des Editors und Einstellungen für das Dekompilierens festgelegt werden.

5.2 Zielplattform

AppEditor wurde mit der IDE *Eclipse Juno* für Android 4.2.2 (API-Level 17) entwickelt. Zum Testen kamen ein gerootetes LG Nexus 4 E 960 mit Android 4.2 und ein nicht-gerootetes Asus Nexus 7 mit Android 4.4.2 zum Einsatz. Dadurch war das Testen der unterschiedlichen Varianten des Layouts optimal möglich.

5.3 Anpassung des *apktool* für Android

AppEditor verwendet intern das bereits in Kapitel 4.4 vorgestellte *apktool*. Die entsprechende JAR-Datei wurde dazu als Bibliothek in *AppEditor* eingebunden. Aus Kompabiltätsgründen

wurde hierzu Version 1.5.3 verwendet, welche noch keine Java 7 benötigt. Die ersten Tests zeigten, dass das Decompiling bereits weitestgehend funktionierte. Jedoch stürzte die die App immer kurz vor Ende des Decompiling-Vorgangs, beim Erstellen des Meta-Files `apktool.yml`, ab. Die Analyse der Logcat-Ausgabe zeigte, dass ein `java.lang.Verify.Error` in `org/yaml/snakeyaml/representer/Representer$RepresentJavaBean` aufgetreten ist. Das deutete auf die Verwendung von DVM inkompatiblem Code hin. Nach Recherche gibt es neben der DVM inkompatiblen *snakeyaml*-Variante eine Variante extra für Android, welche zum Download bereit steht [1].

Im *apktool*-Quellcode musste nun die `build.gradle`. im Ordner `/brut.apktool/apktool-lib` so geändert werden, dass *snakeyaml* beim Build-Prozess nicht mehr aus dem Maven-Repository ([2]) geladen wird, sondern die lokale, Android-kompatible Version verwendet wird. Mit dieser neuen Version des *apktool* konnte auch die `apktool.yml` fehlerfrei erstellt werden.

Bei weiteren Tests kam es zu Problemen beim Decompiling von Apps, welche Nine-Patch-Bilder enthielten. Nine-Patch ist ein spezielles Bildformat, welches besonders einfach skaliert werden kann, um beispielsweise als Hintergrund verschieden großer Buttons zu dienen [9]. Das *apktool* beinhaltet einen eigenen Decoder für Nine-Patch Bilder, dieser verwendet aber Klassen aus `java.awt.*` und `javax.*`, welche mit der aktuellen Version der DVM nicht kompatibel sind. Deshalb musste das Decoding von Nine-Patch-Bildern in der Datei `ResFileDecoder.java` im Source-Ordner `/brut.aptkool/apktool-lib/src/main/java/brut/androlib/res/decoder` deaktiviert werden.

Weitere Besonderheiten waren bei der Angabe des Framework-Files und der *aapt*-Datei zu beachten. Dazu wird in Kapitel 5.4.2 Näheres erklärt, da keine Änderungen des *apktool*-Quellcodes erforderlich waren.

5.4 Architekturbeschreibung

Im nachfolgenden Teil wird die Architektur der App *AppEditor* komplett beschrieben. Die Architekturbeschreibung ist dazu auf die entsprechenden Packages der App aufgeteilt. Insgesamt wurden fünf Packages verwendet:

- `*.activities`: Hier befinden sich alle Activities, die in *AppEditor* verwendet werden. Es gibt sowohl eine englische als auch deutsche Lokalisierung.
- `*.fragments`: Enthalten sind Fragmente für den Filebrowser, die Detailansicht und den Einstellungsbereich.
- `*.apkediting`: Umfangreiches Package mit Klassen, die die Ausführung der *apktool*-Befehle steuern. Außerdem sind Klassen zum Syntax-Highlighting und zur Umsetzung der *Undo/Redo* Funktionalität enthalten.

[1] `https://code.google.com/p/snakeyaml/`
[2] `http://search.maven.org/#search%7Cga%7C1%7Cg%3A%22org.yaml%22`

- `*.treeview`: Hier ist der angepasste Quellcode von *tree-view-list-android* ([3]) zu finden.
- `*.utilities`: Package mit drei Klassen, dessen Funktionalität mehrfach in den anderen Packages verwendet wird.

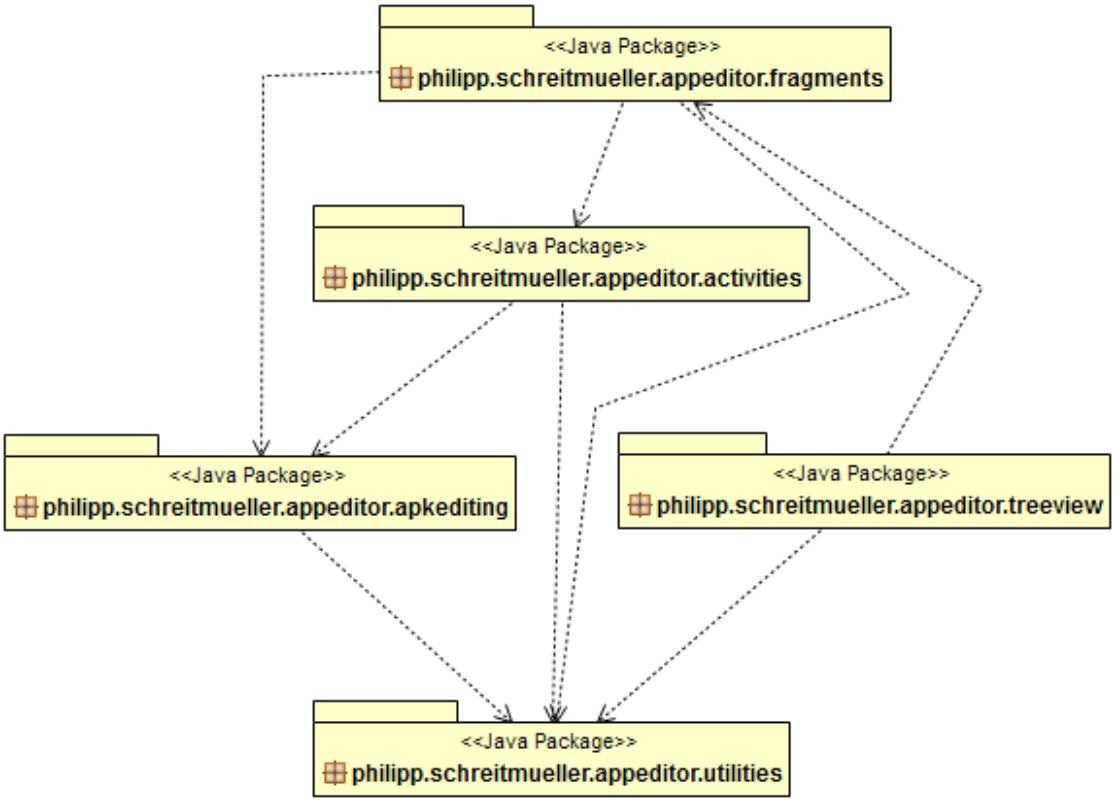


Abbildung 5.2: Package Übersicht von *AppEditor*.

5.4.1 *.activities

Abbildung 5.3 zeigt das Zustandsdiagramm von *AppEditor*. Zu beachten ist hierbei, dass, wie bereits beschrieben, bei einem Tablet-Bildschirm Filebrowser und Detailansicht in einer Activity angezeigt werden. Nachfolgend wird auf die einzelnen Activities eingegangen:

[3] `https://code.google.com/p/tree-view-list-android/`

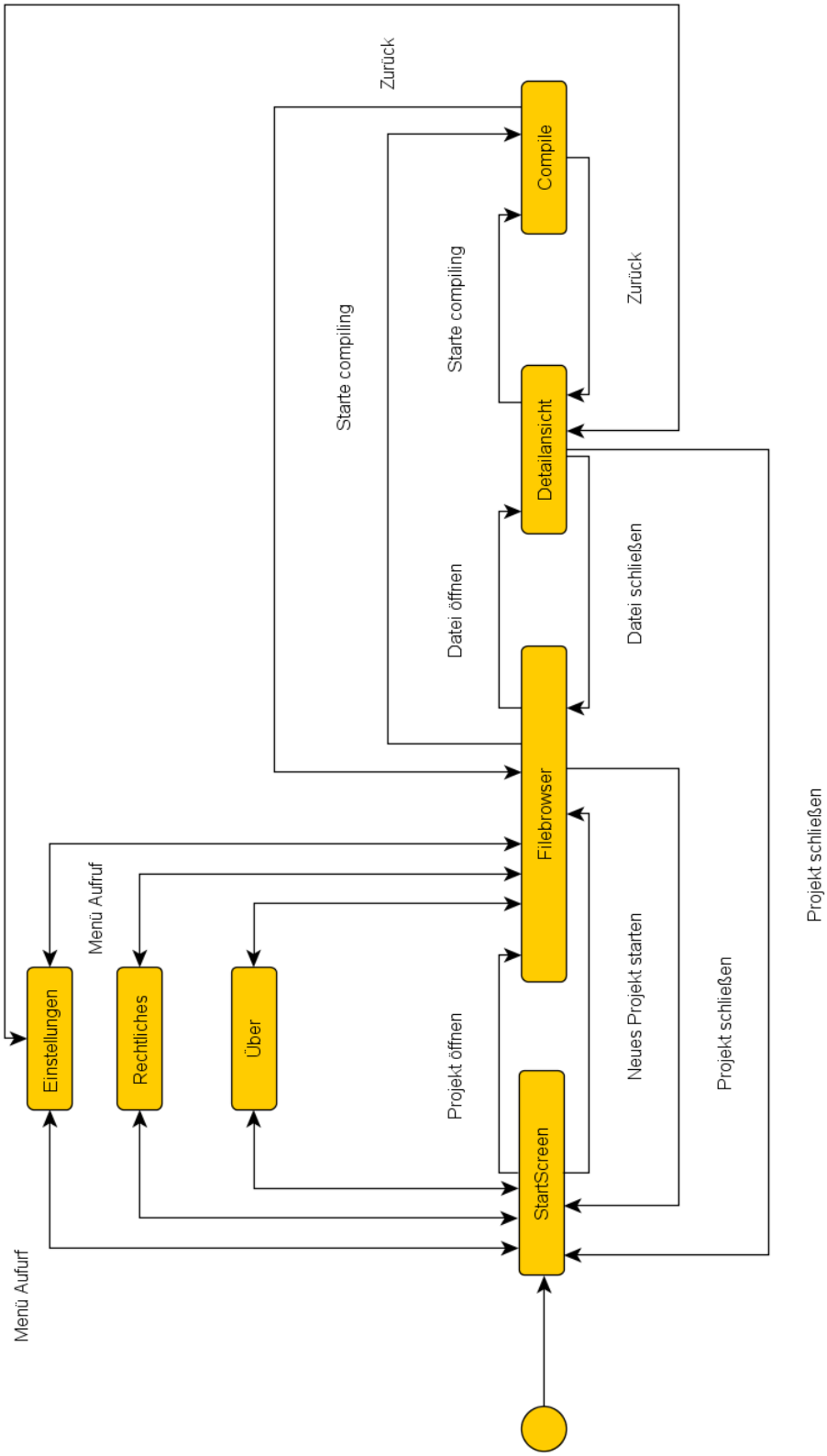


Abbildung 5.3: Zustandsdiagramm von *AppEditor*.

- **StartScreen** (`StartScreen.java`): Die Activity besteht im Wesentlichen aus zwei `ListView`. Eine zeigt eine Übersicht der verfügbaren APK-Projekte an, während die Andere eine Liste aller installierten Apps mit zugehöriger Dateigröße, die ab 4 MB rot dargestellt wird, präsentiert. Das dient als Hinweis an den Benutzer, dass das Dekompilieren einen hohen Hauptspeicherverbrauch nach sich zieht. Für die Darstellung der Listenelemente wurden zwei Inline-Klassen, `AppListAdapter` und `ProjectListAdapter` implementiert, die jeweils von `ArrayAdapter` erben. Für das Laden und Sortieren der Projekte bzw. Apps wurde die Inline-Klasse `LoadAndSort` implementiert, welche von `AsyncTask<Void, Void, Void>` erbt.

  Bei kurzem Klick auf ein Projekt wird der Filebrowser für das entsprechende APK-Projekt geöffnet. Bei langem Klick auf ein APK-Projekt wird ein Dialog geöffnet, der das Löschen des Projekts ermöglicht. Beim Klick auf eine App aus der Liste wird der Decompile-Prozess gestartet, indem in einem Dialog nach dem Namen des neuen Projekts gefragt wird. Nach der Eingabe wird der `AsyncTask DecompileAsync` gestartet, der das Decompiling übernimmt. Nach erfolgreichem Decompiling wird der Filebrowser mit dem neuen Projekt geladen.

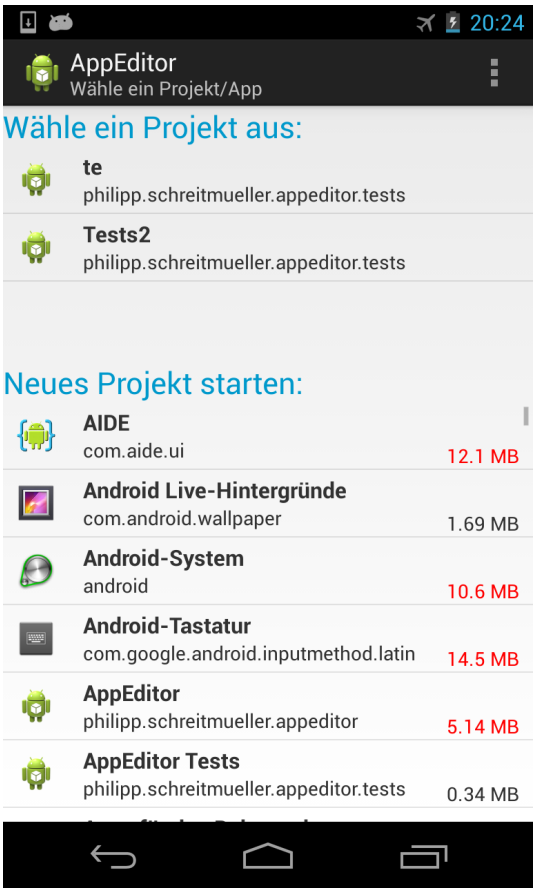


Abbildung 5.4: StartScreen auf dem Nexus 4.

- **Filebrowser** (`FileListActivity.java`): Je nach Displaygröße werden in der Filebrowser Activity nur `FileListFragment.java` oder `FileListFragment.java`

und `FileDetailFragment.java` (*twopane*-Modus) angezeigt. Prinzipiell bekommt `FileListActivity.java` ein `APKProject` Objekt übergeben. Darüber erhält die Activity Auskunft, welches Verzeichnis im Browser dargestellt werden soll.
`FileListActivity` erbt von `FragmentActivity` und implementiert das Interface `OnFileSelected`, welches in `FileListFragment.java` definiert ist. Letzteres sorgt dafür, dass die Methode `public void onFileSelected(String path)` implementiert wird, welche im *twopane*-Modus eine `FragmentTransaction` durchführt, die die neue Datei in die Detailansicht lädt. Wenn die zu schließende Datei verändert wurde, wird eine Warnung angezeigt, die ungewollten Datenverlust verhindern soll. Analog wird im *singlepane*-Modus bei der Auswahl einer Datei im Filebrowser eine neue `FileDetail`-Activity mit der entsprechenden Datei geöffnet. Außerdem ist in `FileListActivity` ein `AsyncTask` implementiert, der das komplette Projekt in eine Zip-Datei komprimiert, um sie per Intent mit anderen Apps zu teilen.

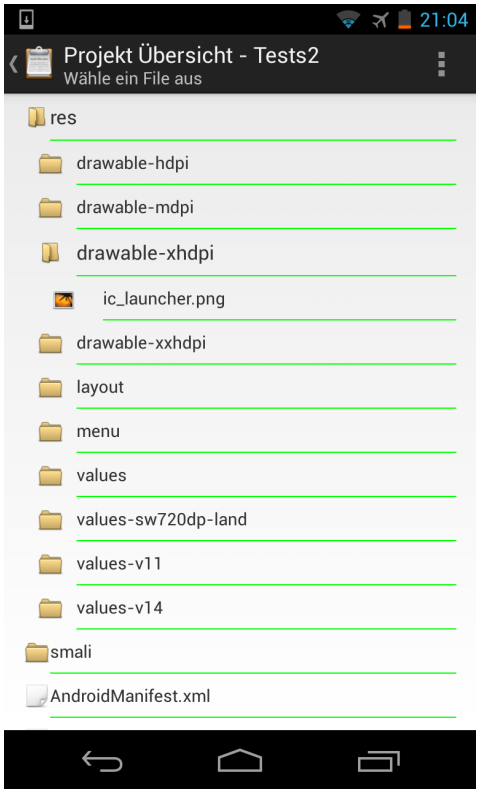


Abbildung 5.5: Filebrowser auf dem Nexus 4.

Abbildung 5.6: Filebrowser auf dem Nexus 7.

- **Detailansicht** (`FileDetailActivity.java`): Diese Activity ist nur für kleinere Bildschirmgrößen relevant. Sie erweitert `FragmentActivity` und erhält beim Aufruf den Pfad zu der Datei, die detailliert betrachtet werden soll. Intern verwaltet

`FileDetailActivity.java` ein `FileDetailFragment`, welches auch den Pfad zur Datei enthält. Alles Weitere wird vom `FileDetailFragment` verwaltet. Lediglich beim Schließen der Activity wird überprüft, ob dadurch mögliche Änderungen der Datei verloren gehen würden. Wenn dem so sein sollte, wird eine entsprechende Warnung dargestellt.

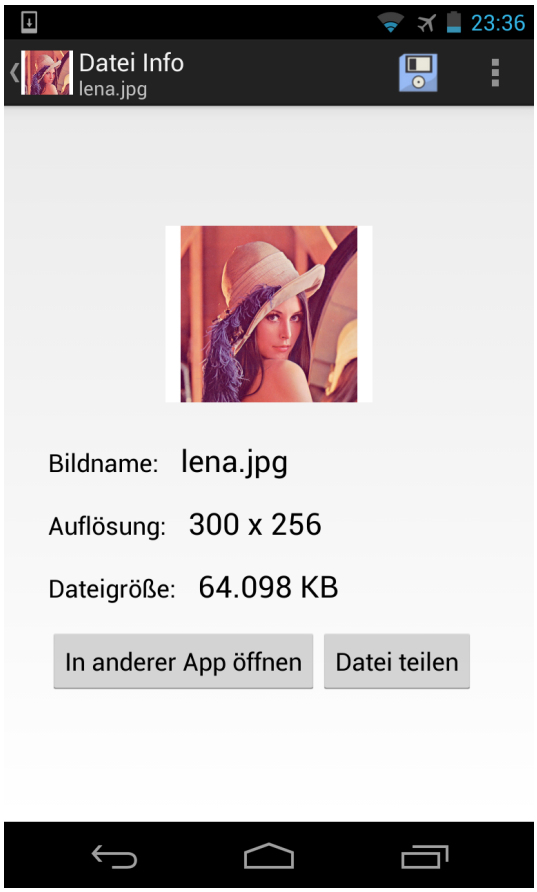


Abbildung 5.7: Detailansicht auf dem Nexus 4.

- **Compile** (`Compile.java`): Die Activity erhält beim Aufruf ein `APKProject` Objekt. Im ersten Schritt muss ein Name für die zu erstellende APK-Datei spezifiziert werden. Mittels eines regulären Ausdrucks wird bei Textänderungen überprüft, ob der Name formal korrekt ist. Sobald der Name gültig ist, kann in Schritt 2 das Kompilieren und Signieren erfolgen. Signiert wird die App mit einem Testkey, um sie direkt auf dem Gerät installieren zu können. Zum Signieren wird die Bibliothek *zip-signer* ([4]) verwendet, welche unter der *Apache License 2.0* veröffentlicht ist. Schritt 2 kann durchaus längere Zeit in Anspruch nehmen und ist deshalb in den `AsyncTask CompileSignAsycn` ausgelagert.
  Nach erfolgreichem Kompilieren kann die App geteilt oder geöffnet/installiert werden. Bei Problemen wird dem Benutzer ein Dialog mit Fehlermeldung angezeigt.

[4] `https://code.google.com/p/zip-signer/`

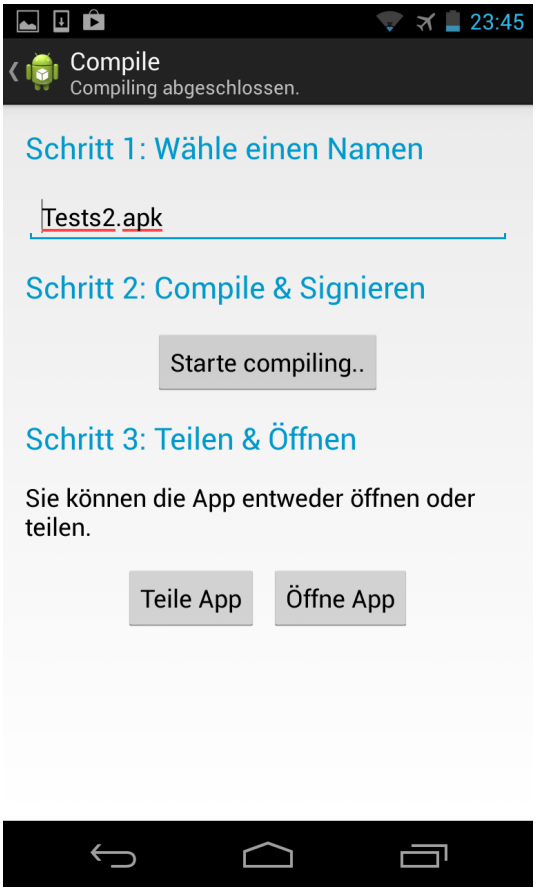


Abbildung 5.8: `Compile`-Activity auf dem Nexus 4.

- **Über** (`About.java`): Die Activity enthält eine kurze Erklärung, in welchem Zusammenhang die App entwickelt wurde, sowie Name und Referenzen zu den verwendeten Dritt-Bibliotheken.
- **Rechtliches** (`Disclaimer.java`): In dieser Activity wird darauf hingewiesen, dass der Entwickler der App keinerlei Haftung für etwaige Gesetzesverstöße, die mit der App begangen werden, übernimmt.
- **Einstellungen** (`SettingsActivity.java`): Die Activity lädt `SettingsFragment.java` in den Anzeigebereich. Dabei wird das von Google vorgeschlagene Vorgehen angewendet ([5]).

[5] `http://developer.android.com/guide/topics/ui/settings.html`

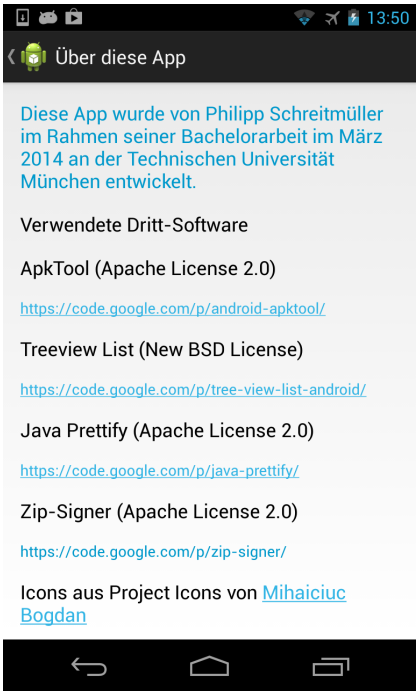


Abbildung 5.9: `About` auf dem Nexus 4.

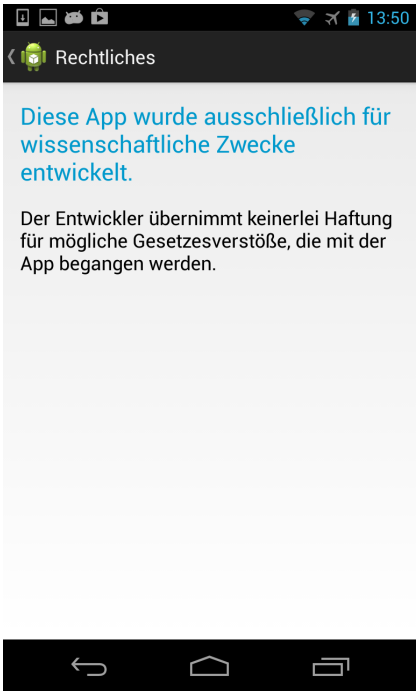


Abbildung 5.10: `Disclaimer` auf dem Nexus 4.

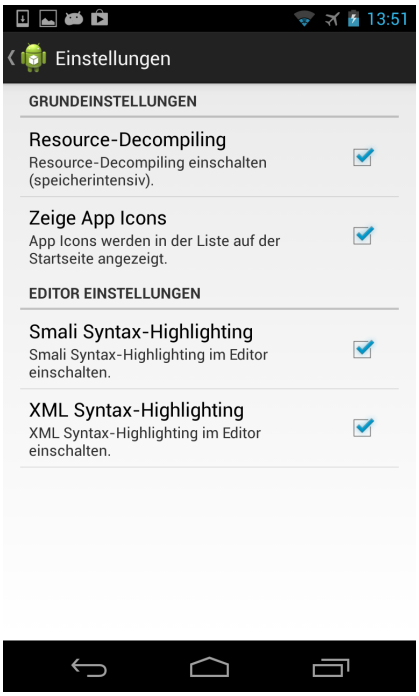


Abbildung 5.11: `SettingsActivity` auf dem Nexus 4.

5.4.2 *.apkediting

Das Package `*.apkediting` bildet die Schnittstelle zum *apktool*. Wie in Abbildung 5.12 ersichtlich, wurde hier das *Fassaden*-Entwurfsmuster verwendet, um die lose Kopplung zu fördern und die Komplexität zu reduzieren. `APKToolFacade.java` stellt Compiling- und Decompiling-Operationen nach außen zur Verfügung. Dafür verwaltet es intern ein `APKToolOffline`- und ein `APKToolOnline`-Objekt. Je nachdem, welcher `APKAccessMode` beim Aufruf von `decompile` bzw. `compile` übergeben wurde, wird dann die Operation an `APKToolOffline` oder `APKToolOnline` delegiert.

Diese Architektur wurde ursprünglich verwendet, weil es ungewiss war, ob Compiling auf dem Gerät selbst durchführbar ist. Durch das *Fassaden*-Entwurfsmuster ist es nun möglich, werden von außen nur Methoden der Fassade aufgerufen. Dadurch bleibt die wirkliche Implementierung versteckt, flexibel und leicht austauschbar. Da Decompiling und Compiling auf dem Android Gerät vollständig möglich sind, wurde die Implementierung einer Online-Variante nicht weiter verfolgt. Mit Blick auf die Android Zukunft wurde aber `APKToolOnline.java` im Projekt gelassen. Bei Bedarf ist so eine Online-Auslagerung leicht möglich.

`APKToolOffline` arbeitet direkt mit der als Bibliothek eingefügten *apktool* JAR-Datei. Zum Decompiling wird dazu die Klasse `ApkDecoder` instanziiert. Ihr wird der Pfad zur Framework-Datei übergeben, welche zwingend benötigt wird. Anhand des übergebenen `APKProject` wird der Zielpfad für das Compiling ermittelt. Dieser hat folgende Form: `<AppEditorDirectory>/packageName/projectName/apktool/`.

Für das Compiling wird die Klasse `Androlib` des *apktool* instanziiert. Auch ihr wird der Pfad zum Framework-File übergeben. Außerdem wird noch der Pfad zur aapt-Datei benötigt. Eine *ARM*-kompatible Version konnte aus dem Projekt *java-ide-droid* übernommen werden, welches unter der *GNU GPL v2* veröffentlicht wurde ([6]). Außerdem wird das Zielverzeichnis gesetzt, welches folgende Form hat: `<AppEditorDirectory>/packageName /projectName/apk/apkName.apk`.

Im Package sind außerdem die Klassen `EditHistory` und `EditItem` enthalten, die vom Editor genutzt werden, um Undo/Redo Operationen zu ermöglichen. `EditHistory` verwaltet eine chronologische Liste an Textänderungen, welche jeweils durch ein `EditItem` repräsentiert werden. Dazu wurde der unter [7] vorgestellte Ansatz angepasst. Der Zusammenhang ist in Abbildung 5.13 dargestellt.

Zuletzt sind noch `PrettifyHighlighter` und `SmaliHighlighter` im Package enthalten. `PrettifyHighlighter` wird zum Syntax-Highlighting von XML und YML Dateien verwendet. Zum Parsing des Quellcodes wird die Bibliothek *java-prettify* ([8]) verwendet. Nach dem Parsing wird jedem Element eine entsprechende Textfarbe zugewiesen. `SmaliHighlighter` übernimmt das Syntax-Highlighting für Smali-Dateien. Dazu sind Gruppen von Keywords

[6] `https://code.google.com/p/java-ide-droid/`
[7] `https://code.google.com/p/android/issues/detail?id=6458`
[8] `https://code.google.com/p/java-prettify/`

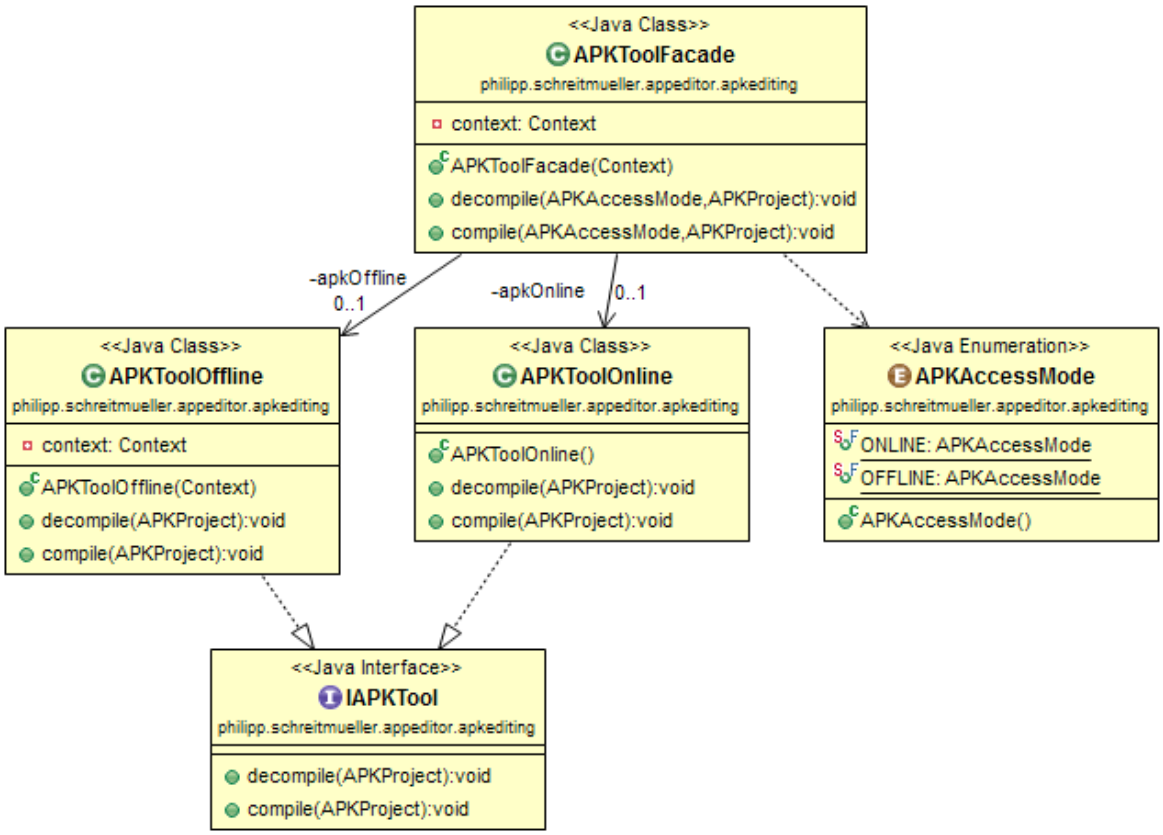


Abbildung 5.12: UML-Diagramm zum Umgang mit *apktool*.

hinterlegt. Nach diesen wird im Quellcode gesucht und die entsprechende Hintergrundfarbe gesetzt.

5.4.3 *.treeview

Dieses Fragment enthält den angepassten Code des *tree-view-list-android* ([9]), welches unter der *New BSD License* veröffentlicht ist. Im Wesentlichen wurde die Datei `SimpleStandardAdapter.java` optimiert: `SimpleStandardAdapter` erweitert `AbstractTreeViewAdapter<String>`. Die einzelnen Elemente des Filebrowsers werden also durch ihren Dateipfad eindeutig identifiziert. Außerdem verwaltet `SimpleStandardAdapter` ein Objekt `callback`, welches das Interface `OnFileSelected` implementiert. `callback` wird später `FileListActivity` referenzieren. Wenn also ein Element im Filebrowser gewählt wird, wird zuerst `public void handleItemClick( final View, final Object)` in `SimpleStandardAdapter` aufgerufen, diese wiederum ruft direkt `callback.OnFileSelected(String)` auf.
Außerdem wird die Darstellung von Files im Filebrowser in `SimpleStandardAdapter` gesteuert. Über `protected Drawable getDrawable(final TreeNodeInfo<String>)` kann den einzelnen Filebrowser-Einträgen ein Icon zugewiesen werden, so beispiels-

[9] `https://code.google.com/p/tree-view-list-android/`

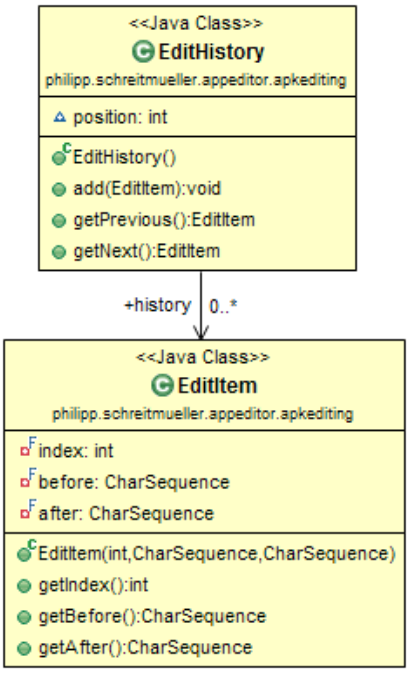


Abbildung 5.13: Klassen zur Festhalten von Textänderungen.

weise ein Ordner-Symbol für Ordner-Einträge oder ein Graphik-Symbol für Bilddateien. Die Icons sind unter Creative Commons Attribution 3.0 von Mihaiciuc Bogdan unter [10] veröffentlicht. Über die Methode `public LinearLayout updateView(final View, final TreeNodeInfo<String>)` wird der Filename aus dem Pfad der Datei extrahiert und in der View sichtbar gemacht.

5.4.4 *.utilities

In diesem Package sind Klassen enthalten, die die Funktionalität, welche in verschiedenen anderen Packages benötigt wird, implementieren. Konkret sind es folgende Klassen:

- **`APKProject.java`**: Dient als Datenobjekt für den Austausch von Informationen über Projekte. Unter anderem werden hier Package- und Projektname, Pfad zum Projekt sowie der Name der zu erzeugenden APK-Datei gespeichert. Die Klasse implementiert `Serializable`. Dadurch können `APKProject` Objekte als Anhang in einem Intent definiert werden.
- **`FileTypes.java`**: Enthält drei statische Listen, in denen Dateiendungen gespeichert werden. Es gibt eine Liste für XML-basierte Dateien, eine für Bilddateien und eine für Smali-Dateien. Dadurch kann beispielsweise entschieden werden, in welchem Modus `FileDetailFragment` betrieben werden soll.
- **`Utils.java`**: Enthalten sind hier hauptsächlich statische Hilfsmethoden, die aus anderen Packages aufgerufen werden, um den Code übersichtlicher zu gestalten. Dazu gehören verschiedene File-Operationen: Löschen von Projekten, Erstellen einer

[10] `http://bogo-d.deviantart.com`

List mit allen vorhandenen Projekten, Übertragen von aapt und Framework-File aus den raw-Ressourcen in den privaten App-Speicher und Packen und Entpacken von Zip-Archiven. Für das Teilen von Dateien wird zuerst `public static File copyToExternalStorage(Context , File)` aufgerufen, um die Datei aus dem privaten Speicher zu kopieren. Danach kann mit `public static void openInApp( Context, File, String)` ein Intent gestartet werden.

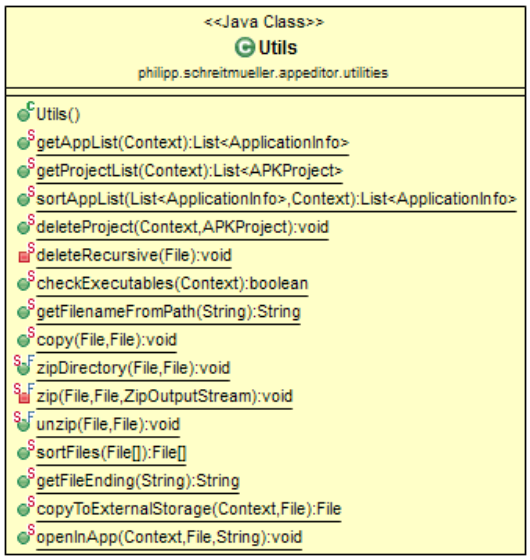


Abbildung 5.14: Übersicht der Methoden von `Utils.java`.

5.4.5 *.fragments

- **`FileListFragment.java`**: In diesem `Fragment` wird die Darstellung des Filebrowsers implementiert. `FileListFragment` bekommt von der Eltern-Activity das `APKProject` übergeben, für welches der Filebrowser gefüllt werden soll. Dazu verwaltet das Fragment ein `TreeStateManager<String>` Objekt, welches bei Orientierungsänderungen des Bildschirms mit der Methode `private void onSaveInstanceState(Bundle)` gecached wird. In der `public View onCreateView( LayoutInflater, ViewGroup , Bundle)` wird dann überprüft, ob ein `TreeStateManager<String>` im `Bundle`-Objekt gespeichert wurde, und gegebenenfalls aus dem Speicher geladen. Mit einem `TreeBuilder<String>` Objekt werden dann alle Files rekursiv in den Baum eingefügt. Letztendlich wird dann noch ein `SimpleStandardAdapter` instanziiert und dem `TreeView` Objekt, welches aus dem Layout referenziert wurde, zugewiesen.
- **`SettingsFragment.java`**: Hier war sehr wenig Eigenarbeit zu leisten, weil `SettingsFragment` von `PreferenceFragment` abgeleitet ist. Mit dem Aufruf `addPreferencesFromResource(R.xml.preferences);` wird die Preference-Spezifikation eingelesen. Der Darstellung der Einstellungen sowie die Speicherung erfolgen automatisch. Aktuell kann eingestellt werden ob Ressourcen dekompiliert werden sollen, ob die verschiedenen App-Icons in der Übersichtsliste angezeigt werden sollen und ob XML und Smali-Syntaxhighlighting durchgeführt werden

sollen.

- **`FileDetailFragment.java`**: Repräsentiert im Wesentlichen den File-Editor. Dem Fragment wird die darzustellende Datei und das `APKProject` übergeben. Abhängig von der Dateiendung wird entweder `R.layout.fragment_file_img`, `R.layout.fragment_file_txt` oder `R.layout.fragment_file_na`, für nicht-unterstützte Filetypen geladen. Danach wird im Falle eines text-basierten Files die Datei asynchron geladen und je nach Einstellung Syntax-Highlighting durchgeführt. Außerdem wird ein `TextWatcher` initialisiert, der Undo/Redo Aktionen ermöglicht und den `hasChanged` Boolean Wert gegebenenfalls auf `true` setzt. Der Ladevorgang des Bildbetrachters unterscheidet sich grundsätzlich. Jedoch gibt es auch kleinere Unterschiede zwischen dem Smali und dem XML-Modus.

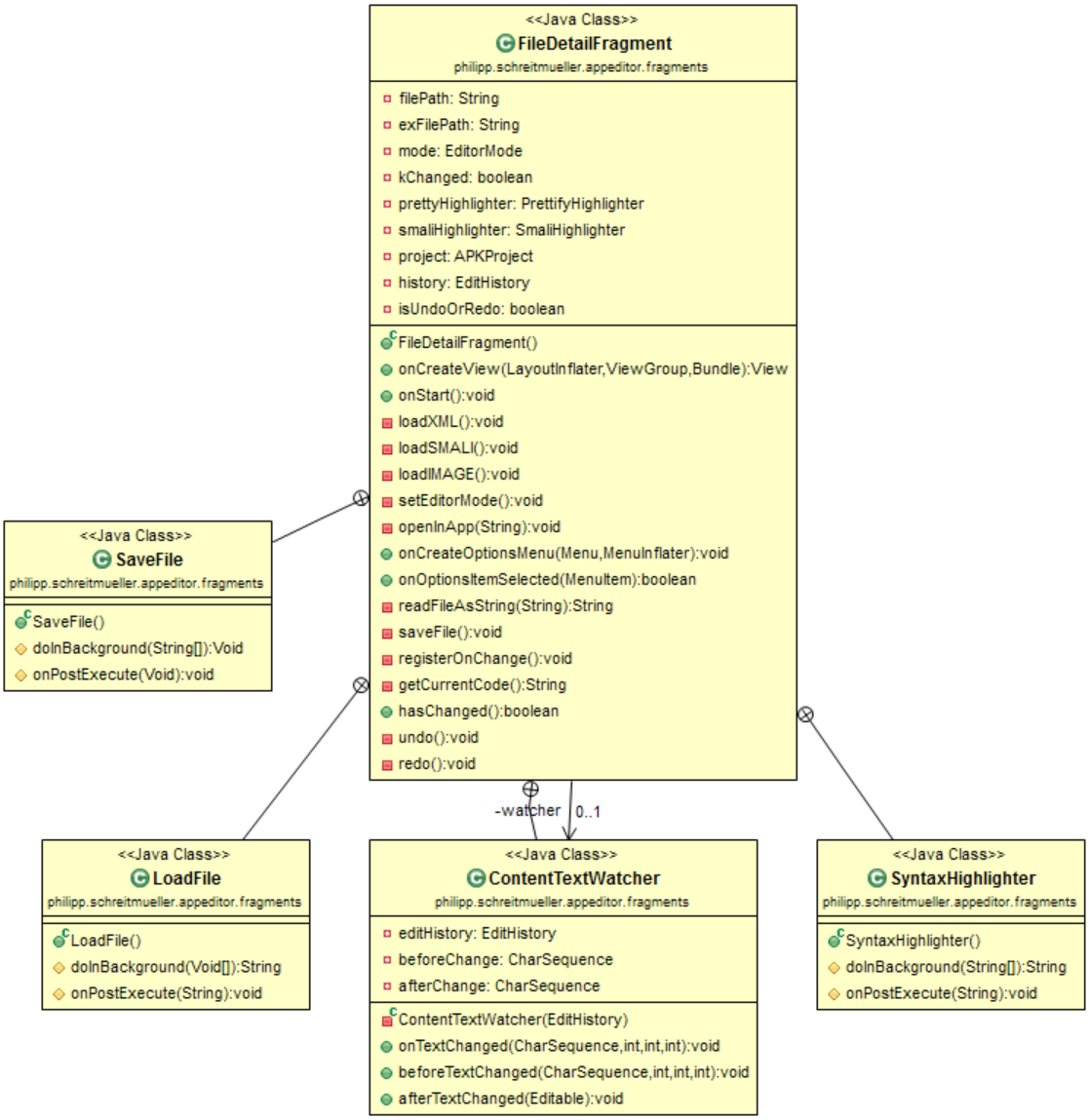


Abbildung 5.15: Klassendiagramm von `FileDetailFragment.java`.

6 Qualitätsanalyse

In diesem Kapitel wird der Testplan von *AppEditor* präsentiert. Zum Testen wurde eine App mit Namen *AppEditor Tests* entwickelt. Anhand *AppEditor Tests* und der App *JDiary*, die im Rahmen des *Android Praktikums* bei Herrn Kannengießer im Sommersemester 2013 entwickelt wurde, wird getestet, ob alle Funktionen von *AppEditor* ordnungsgemäß funktionieren. Wie bereits in Kapitel 2.2 näher besprochen, sind beim Reengineering immer moralische und rechtliche Aspekte zu berücksichtigen. In den vorgestellten Testfällen wurden die zu dekompilierende Apps vom Autor der Arbeit entwickelt. Deshalb gibt es hier keinerlei rechtlichen Probleme. Am Schluss wird noch ein Blick auf die Performance von *AppEditor* geworfen.

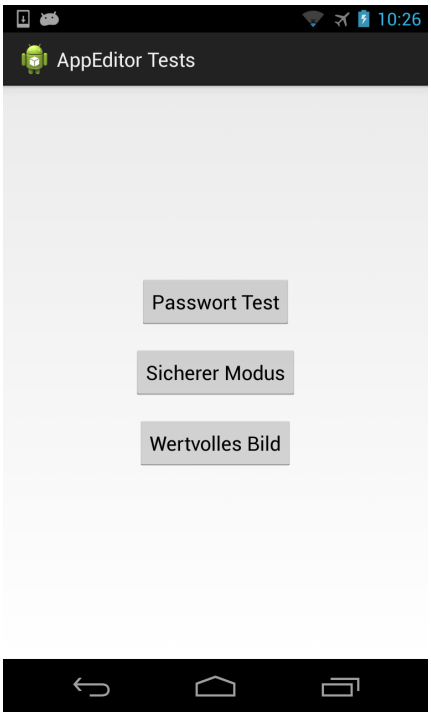


Abbildung 6.1: Übersichts-Activity von *AppEditor Tests*.

6.1 Bearbeitung von Smali Quellcode

Im ersten Beispiel soll gezeigt werden, wie leicht eine Passwort-Abfrage umgangen werden kann, wenn naiv programmiert wurde. In der Activity wird nach einem Passwort gefragt: Je nachdem, ob das eingegebene Passwort richtig ist, wird ein entsprechender `Toast` angezeigt.

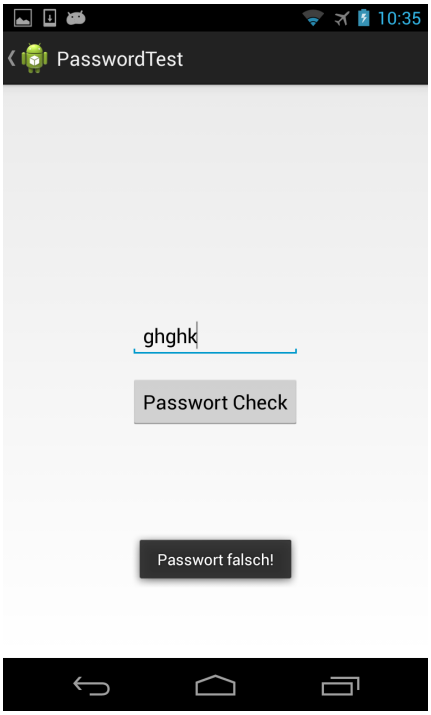


Abbildung 6.2: Passworttest in *AppEditor Tests*.

Der zugehörige Java Quellcode sieht folgendermaßen aus:

```
1  package philipp.schreitmueller.appeditor.tests;
2
3  import android.os.Bundle;
4  import android.app.Activity;
5  import android.view.MenuItem;
6  import android.view.View;
7  import android.widget.EditText;
8  import android.widget.Toast;
9  import android.support.v4.app.NavUtils;
10
11 public class PasswordTest extends Activity {
12   private String password = "pass123";
13
14   @Override
15   protected void onCreate(Bundle savedInstanceState) {
16     super.onCreate(savedInstanceState);
17     setContentView(R.layout.activity_password_test);
18     getActionBar().setDisplayHomeAsUpEnabled(true);
19     findViewById(R.id.passCheck).setOnClickListener(
20         new View.OnClickListener() {
21
22           @Override
23           public void onClick(View v) {
24             checkPassword();
25
```

```
26              }
27          });
28    }
29
30    @Override
31    public boolean onOptionsItemSelected(MenuItem item) {
32      switch (item.getItemId()) {
33      case android.R.id.home:
34        NavUtils.navigateUpFromSameTask(this);
35        return true;
36      }
37      return super.onOptionsItemSelected(item);
38    }
39
40    private void checkPasswort() {
41      if (((EditText) findViewById(R.id.passInput)).getText().toString()
42          .equals(password))
43        Toast.makeText(getApplicationContext(), "Passwort richtig!",
44            Toast.LENGTH_LONG).show();
45      else
46        Toast.makeText(getApplicationContext(), "Passwort falsch!",
47            Toast.LENGTH_LONG).show();
48    }
49  }
```

Listing 6.1: PasswordTest.java

Der Passwort wurde hier im Klartext als Objekt Eigenschaft gespeichert. Jetzt wird die App mittels *AppEditor* auf dem Gerät dekompiliert und der dekompilierte Code untersucht.

```
1  .class public Lphilipp/schreitmueller/appeditor/tests/PasswordTest;
2  .super Landroid/app/Activity;
3  .source "PasswordTest.java"
4
5
6  # instance fields
7  .field private password:Ljava/lang/String;
8
9
10 # direct methods
11 .method public constructor <init>()V
12    .locals 1
13
14    .prologue
15    .line 11
16    invoke-direct {p0}, Landroid/app/Activity;-><init>()V
17
18    .line 12
19    const-string v0, "pass123"
20
21    iput-object v0, p0, Lphilipp/schreitmueller/appeditor/tests/
      PasswordTest;->password:Ljava/lang/String;
22
```

```
23     .line 11
24     return-void
25 .end method
26 
27 .method static synthetic access$0(Lphilipp/schreitmueller/appeditor/tests
       /PasswordTest;)V
28     .locals 0
29     .parameter
30 
31     .prologue
32     .line 37
33     invoke-direct {p0}, Lphilipp/schreitmueller/appeditor/tests/
       PasswordTest;->checkPasswort()V
34 
35     return-void
36 .end method
37 
38 # Hier ist die Methode die geaendert werden muss, um den Passwortschutz
       auszutricksen
39 .method private checkPasswort()V
40     .locals 3
41 
42     .prologue
43     const/4 v2, 0x1
44 
45     .line 38
46     const/high16 v0, 0x7f08
47 
48     invoke-virtual {p0, v0}, Lphilipp/schreitmueller/appeditor/tests/
       PasswordTest;->findViewById(I)Landroid/view/View;
49 
50     move-result-object v0
51 
52     check-cast v0, Landroid/widget/EditText;
53 
54     invoke-virtual {v0}, Landroid/widget/EditText;->getText()Landroid/text
       /Editable;
55 
56     move-result-object v0
57 
58     invoke-interface {v0}, Landroid/text/Editable;->toString()Ljava/lang/
       String;
59 
60     move-result-object v0
61 
62     iget-object v1, p0, Lphilipp/schreitmueller/appeditor/tests/
       PasswordTest;->password:Ljava/lang/String;
63 
64       # Aufruf der equals Methode mit v0 (Benutzereingabe) und v1 (
       password Member)
65     invoke-virtual {v0, v1}, Ljava/lang/String;->equals(Ljava/lang/Object
       ;)Z
66 
```

```
67      move-result v0
68
69        # Das Ergebnis des Vergleichs liegt in v0. Es wird Gleichheit mit
        Null (falsches Eingabe) ueberprueft und gegebenenfalls nach :cond_0
        gesprungen
70      if-eqz v0, :cond_0
71
72        # Wird dieser CodePfad ausgefuehrt, war die Eingabe richtig.
73      .line 39
74      invoke-virtual {p0}, Lphilipp/schreitmueller/appeditor/tests/
        PasswordTest;->getApplicationContext()Landroid/content/Context;
75
76      move-result-object v0
77
78      const-string v1, "Passwort richtig!"
79
80      invoke-static {v0, v1, v2}, Landroid/widget/Toast;->makeText(Landroid/
        content/Context;Ljava/lang/CharSequence;I)Landroid/widget/Toast;
81
82      move-result-object v0
83
84      invoke-virtual {v0}, Landroid/widget/Toast;->show()V
85
86      .line 42
87      :goto_0
88      return-void
89
90      .line 41
91      :cond_0
92      invoke-virtual {p0}, Lphilipp/schreitmueller/appeditor/tests/
        PasswordTest;->getApplicationContext()Landroid/content/Context;
93
94      move-result-object v0
95
96      const-string v1, "Passwort falsch!"
97
98      invoke-static {v0, v1, v2}, Landroid/widget/Toast;->makeText(Landroid/
        content/Context;Ljava/lang/CharSequence;I)Landroid/widget/Toast;
99
100     move-result-object v0
101
102     invoke-virtual {v0}, Landroid/widget/Toast;->show()V
103
104     goto :goto_0
105 .end method
106
107 # Zur Uebersichtlichkeit wurde der restliche Quellcode hier weggelassen
```

Listing 6.2: PasswordTest.smali

Wie man sieht, ist im Smali-Code das Passwort direkt einsehbar. Dadurch kann man beispielsweise das Passwort im Code ändern, die App erneut kompilieren und dann die ur-

sprüngliche App mit neuem Passwort nutzen. Eine neue Version der *AppEditor Tests* ist in Abbildung 6.3 zu sehen. Man kann auch den Opcode `if-eqz` („if equal zero“) in Zeile 70 in `if-nez` („if not equal zero“) ändern. In dieser Zeile wird das Ergebnis des Stringvergleichs, welches in `v0` gespeichert ist, mit Null verglichen. Durch das Umkehren der Logik werden alle Passwörter akzeptiert, die ungleich dem ursprünglich vorgesehenen Passwort sind. Eine andere Möglichkeit wäre nach dem Aufruf der `equals`-Methode, den Wert von `v1` fest auf 1 zu setzen.

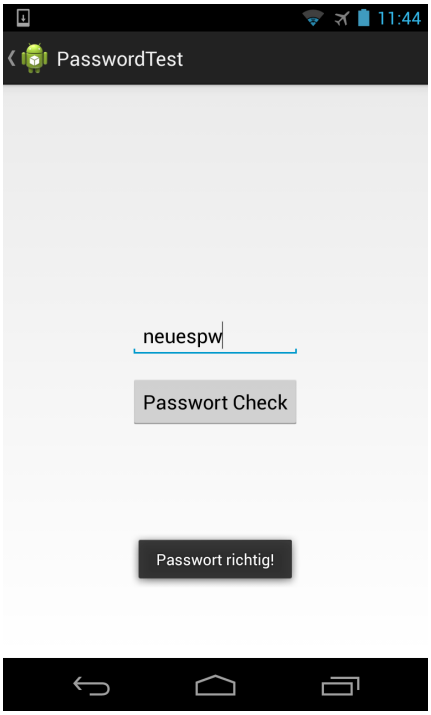


Abbildung 6.3: Geändertes Passwort in *AppEditor Tests*.

In *AppEditor Tests* ist auch ein Testfall implementiert, bei dem ein sicherer Modus simuliert wird. Abhängig von einem internen Boolean-Wert wird der gesicherte Bereich oder ein öffentlicher Bereich angezeigt. Der Java Code der Activity sieht wie folgt aus:

```
1  package philipp.schreitmueller.appeditor.tests;
2
3  import android.os.Bundle;
4  import android.app.Activity;
5  import android.view.MenuItem;
6  import android.widget.TextView;
7  import android.support.v4.app.NavUtils;
8
9  public class SecureMode extends Activity {
10
11   private boolean secureMode=false;
12
13   @Override
14   protected void onCreate(Bundle savedInstanceState) {
```

```
15      super.onCreate(savedInstanceState);
16      setContentView(R.layout.activity_secure_mode);
17      // Show the Up button in the action bar.
18      getActionBar().setDisplayHomeAsUpEnabled(true);
19      if(secureMode)
20         ((TextView)findViewById(R.id.secureMessage)).setText("Sie sind im
     gesicherten Modus.\n Hier eine Geheimzahl:17121991.");
21      else
22         ((TextView)findViewById(R.id.secureMessage)).setText("Sie befinden
     sich aktuell nicht im sicheren Modus.");
23
24    }
25    @Override
26    public boolean onOptionsItemSelected(MenuItem item) {
27      switch (item.getItemId()) {
28      case android.R.id.home:
29
30        NavUtils.navigateUpFromSameTask(this);
31        return true;
32      }
33      return super.onOptionsItemSelected(item);
34    }
35
36  }
```

Listing 6.3: SecureMode.java

Wenn man die korrespondierende `SecureMode.smali` so bearbeitet, dass statt einer 0 eine 1 im Member `secureMode` gespeichert wird, kann auf den gesicherten Bereich zugegriffen werden.

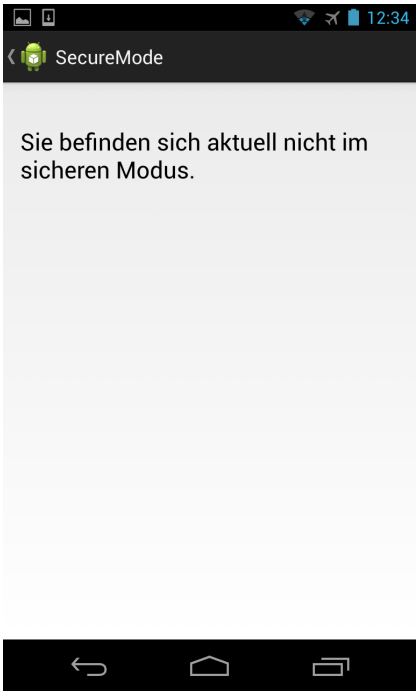


Abbildung 6.4: `SecureMode.java` im öffentlichen Modus.

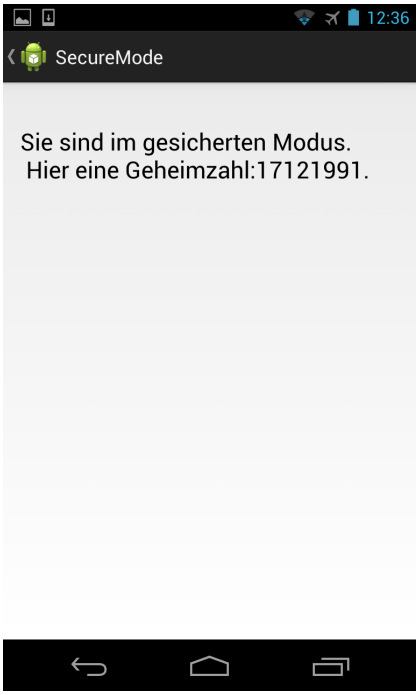


Abbildung 6.5: `SecureMode.java` im gesicherten Modus.

Die Vorgehensweise kann auch auf die im Praktikum entwickelte App *JDiary* angewendet werden. Dort ist eine Serveradresse fest im Code hinterlegt.

6.2 Bearbeiten von Ressourcen

Neben der Bearbeitung von Smali Code, soll hier auch noch die Ressourcen-Bearbeitung getestet werden. Es wurde eine neue Version der vorgestellten *AppEditor Tests* kompiliert, bei der der Name der App und Button-Beschriftungen geändert wurden. Außerdem wurde ein neuer Button ins Layout eingefügt.

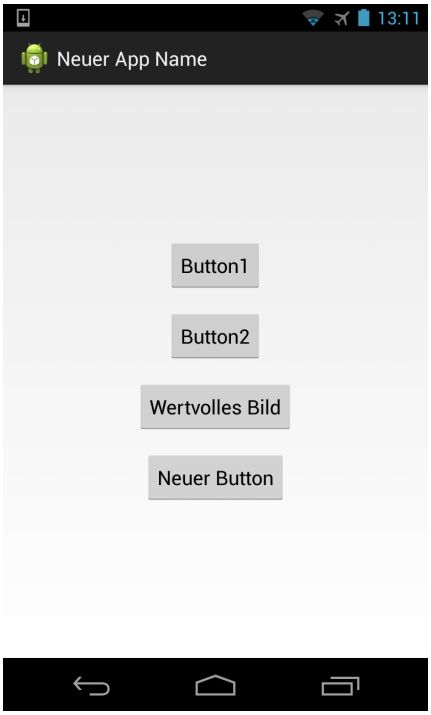


Abbildung 6.6: Geändertes Layout in *AppEditor Tests*.

Mit diesem Test soll nur eine kleine Demonstration der Möglichkeiten gegeben werden: Prinzipiell lassen sich neue Layouts, Texte und damit die App komplett umgestalten.

6.3 Speichern von Ressourcen

Als letzter Test soll noch demonstriert werden, wie eine Bild-Ressource, die in einer App integriert ist, mit Hilfe des *AppEditors* in voller Qualität exportiert werden kann. Dazu wurde in *AppEditor Tests* die Klasse `SecureImage` eingeführt, welche ein Bild aus den Ressourcen anzeigt. Der in *AppEditor* integrierte Bildebetrachter ermöglicht den Export der entsprechenden Datei (siehe Abbildung 6.7).

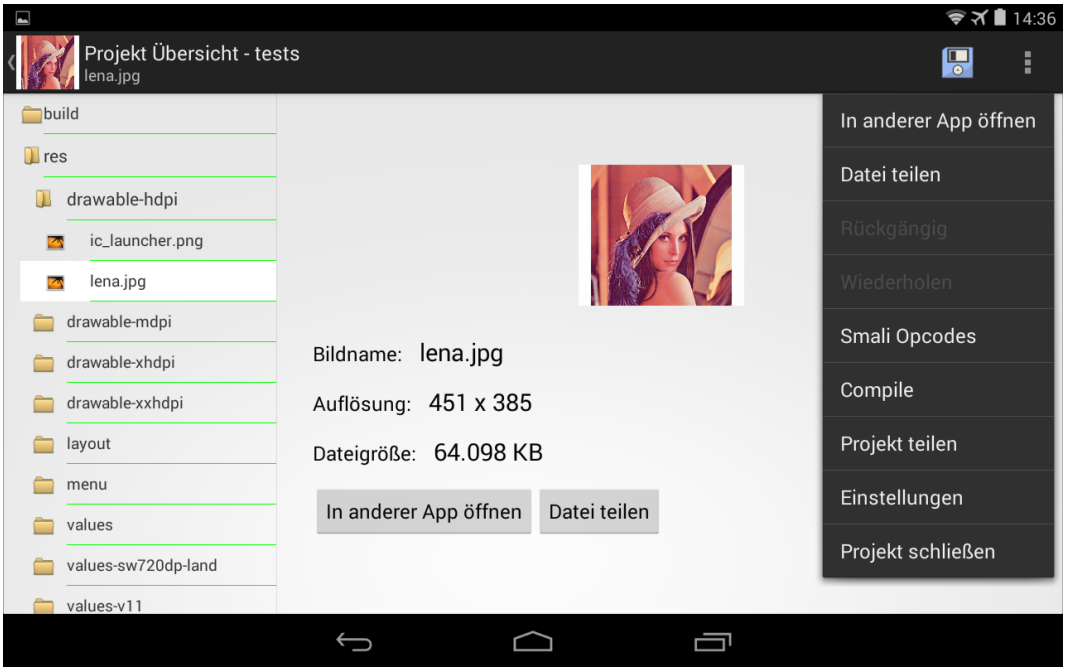


Abbildung 6.7: Export der Original Bilddatei aus *AppEditor Tests* auf dem Nexus 7.

6.4 Performance

Decompiling und Compiling sind relativ speicher- und CPU-hungrige Vorgänge. Abbildung 6.8 zeigt die Anteile an der CPU Auslastung während des Dekompilierens von *AppEditor Tests*. Insgesamt beträgt der Anteil an der CPU-Auslastung ca. 75%. Auch die Memory-Auslastung ist selbst für kleinere Apps relativ groß. Deshalb wird von einem Dekompilieren von Apps, die größer als 4 MB sind, derzeit abgeraten. Die langsame Performance ist wahrscheinlich auf das apktool in der Kombination mit der DVM zurückzuführen. Im Speziellen wirken sich große Bildressourcen auf die Performance aus. Das zeigen auch die Testmessungen:

App	Gerät	Decompiling	Compiling
AppEditor Tests	Nexus 4	38,861s	59,564s
JDiary	Nexus 4	28,537s	3.473s
AppEditor Tests	Nexus 7	38,921s	56,292s
JDiary	Nexus 7	27,782s	2,588s

Tabelle 6.1: Performance Messungen von *AppEditor*.

Zukünftig dürften diese Probleme aber immer weniger ins Gewicht fallen, nachdem jedes Jahre neue, schnellere Prozessoren für Smartphones entwickelt werden. Auf normalen Desktops dauern die Operationen deutlich kürzer, nämlich wenige Sekunden.

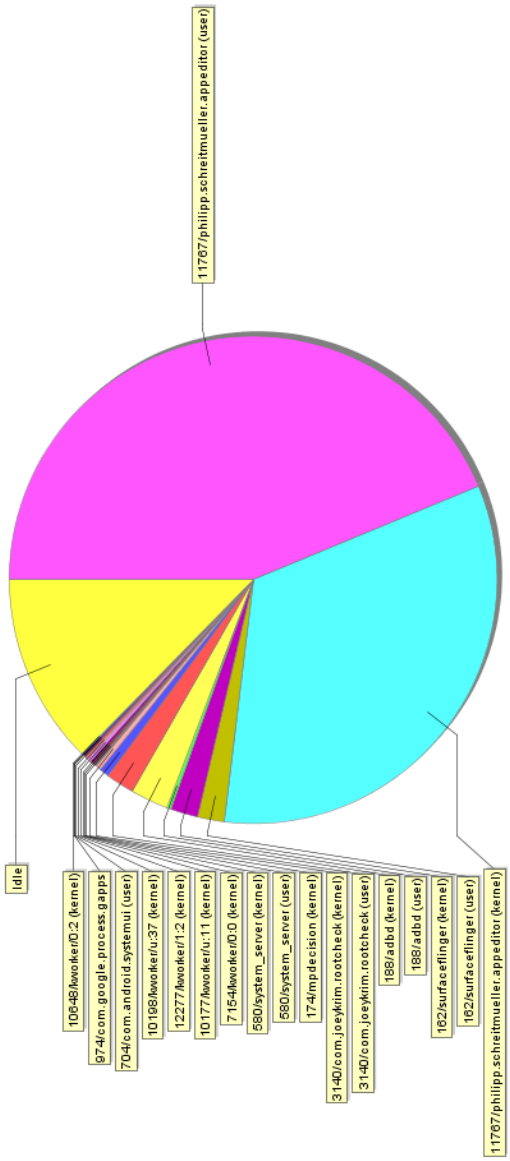


Abbildung 6.8: Verteilung der CPU Last beim Decompiling von *AppEditor Tests* auf dem Nexus 4.

7 Ergebnis

Es wird ein Blick in die Zukunft des Android App-Format geworfen, die Ergebnisse zusammengefasst sowie Verbesserungen, die in der Zukunft noch implementiert werden können, angesprochen.

7.1 Future Android: Android ART

In Android 4.4 wird ART experimentell eingeführt. Dabei handelt es sich um eine neue Laufzeitumgebung, welche die DVM ersetzt [11]. ART lässt sich bisher ausschließlich in den Entwickleroptionen auswählen. Standard ist weiterhin die DVM. Jedoch sollten Entwickler bereits die Möglichkeit bekommen das neue Laufzeitsystem auszuprobieren. Anders als bei der DVM wird hier Ahead-of-Time Compiling betrieben. Das bedeutet, dass die Apps bereits vor dem Öffnen in Maschinencode vorliegen. Dadurch soll Performance-Gewinn erreicht werden, da ja bereits vorher Zeit für die Kompilierung in den Maschinencode aufgewendet wurde. Bei der DVM wird Just-in-time Kompilierung verwendet, welche flexibler, aber dafür nicht ganz so schnell ist.

In der Realität wurden aber weder in [20] noch in [14] signifikante Geschwindigkeitsvorteile festgestellt. Im Gegenteil: Die Apps belegen in der ART-Version deutlich mehr Speicherplatz, weil ja Maschinen- statt interpretiertem Code gespeichert werden muss.

Trotzdem wird ART, an dem mittlerweile auch schon Jahre gearbeitet wurde, wahrscheinlich in einer zukünftigen Android Version die DVM ersetzen. Bisher gibt es noch keinen angepassten Compiler, um direkt ART Apps zu erzeugen. Apps können bisher ausschließlich auf dem Gerät angepasst werden. Das bedeutet, dass die DVM-kompatible Version weiterhin auf dem Gerät vorhanden sein muss. Bei Apps, welche nur in optimiertem Maschinencode vorliegen, ist das Decompiling beziehungsweise Compiling schwer bis nicht vollständig möglich. Dadurch wären Apps grundsätzlich besser gegen (ungewolltes) Decompiling geschützt. Noch gibt es seitens Google noch keine ausführlicheren Informationen als sie in [11] zu finden sind. Die Zukunft des Android App-Formats bleibt also spannend.

7.2 Ausblick

In der Arbeit wurde die Android App *AppEditor* vorgestellt, welche das Dekompilieren, Bearbeiten und erneute Kompilieren von Android Apps direkt auf dem Gerät ermöglicht.

Eine solche App gibt es aktuell nicht im Play Store. Deshalb kann durchaus davon gesprochen werden, dass hier etwas Neues entwickelt wurde. Die App verwendet lediglich Dritt-Bibliotheken, welche unter *Open-Source* Lizenzen stehen. Dadurch wäre eine Veröffentlichung der App durchaus möglich. Leider würde wohl auch hier, ähnlich wie bei der in Kapitel 4.2 vorgestelllten App *AppGuard*, eine schnelle Entfernung seitens Google drohen. Deshalb und auch aufgrund von rechtlichen Unsicherheiten, welche den Entwickler betreffen, ist eine Veröffentlichung im Play Store aktuell nicht geplant. Trotzdem wird die Arbeit an *AppEditor* fortgesetzt, weil das Projekt großes Potenzial hat. Erweiterungen sind im Bereich des Texteditor vorstellbar. Außerdem wäre eine Implementierung der `APKToolOnline.java` , um performance-kritische Vorgänge auf Wunsch auf schnellere Server auszulagern, ein weiteres Kriterium für zukünftige Arbeit an dem Projekt. Eine Veröffentlichung des Quellcodes auf einer offenen Onlineplattform ist geplant.

Anhang

Quellcode

Auf diesem Datenträger befindet sich der Quellcode aller beschriebenen Anwendungen.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Literaturverzeichnis

[1] android.com. Android security overview. http://source.android.com/devices/tech/security/index.html. [Online; abgerufen am 19. Februar 2014].

[2] backes:SRT. Appguard. http://www.srt-appguard.com/de/. [Online; abgerufen am 21. Februar 2014].

[3] Arno Becker. Die architektur von android. http://www.heise.de/ct/artikel/Innenansichten-eines-Smartphone-Betriebssystems-1901647.html, March 2011. [Online; abgerufen am 26. Januar 2014].

[4] Arno Becker and Marcus Pant. *Android 2: Grundlagen und Programmierung*. dpunkt. verlag, 2011.

[5] Nicola Carlo. Einblick in die dalvik virtual machine. *IMVS Fokus Report*, 3:7, 2009.

[6] Cristina Cifuentes. History of decompilation (1960-1979). http://www.program-transformation.org/Transform/HistoryOfDecompilation1, 1998. [Online; abgerufen am 12. Februar 2014].

[7] connor.tumbleson. Apktooloptions. https://code.google.com/p/android-apktool/wiki/ApktoolOptions. [Online; abgerufen am 21. Februar 2014].

[8] Google Developers. App manifest. http://developer.android.com/guide/topics/manifest/manifest-intro.html. [Online; abgerufen am 18. Februar 2014].

[9] Google Developers. Canvas and drawables - nine-patch. http://developer.android.com/guide/topics/graphics/2d-graphics.html#nine-patch. [Online; abgerufen am 24. Februar 2014].

[10] Google Developers. Fragments. http://developer.android.com/guide/components/fragments.html. [Online; abgerufen am 24. Februar 2014].

[11] Google Developers. Introducing art. https://source.android.com/devices/tech/dalvik/art.html. [Online; abgerufen am 26. Februar 2014].

[12] Carsten Drees. Idc: Android nun bei 81 prozent marktanteil bei den smartphones. http://www.mobilegeeks.de/idc-android-nun-bei-81-prozent-marktanteil-bei-den-smartphones, November 2013. [Online; abgerufen am 4. Februar 2014].

[13] Frank Erdle. Android: Die geschichte des erfolgs. http://www.connect.de/ratgeber/android-geschichte-des-erfolgs-1491130.html, May 2013. [Online; abgerufen am 5. Februar 2014].

[14] Lukas Funk. Art: Alternative android-runtime im akku-benchmark. http://www.androidnext.de/news/

art-alternative-android-runtime-im-akku-benchmark. [Online; abgerufen am 26. Februar 2014].

[15] Google. Android. http://www.android.com, January 2014. [Online; abgerufen am 5. Februar 2014].

[16] Google. Dashboards. http://developer.android.com/about/dashboards/index.html, January 2014. [Online; abgerufen am 10. Februar 2014].

[17] Anmol Misra and Abhishek Dubey. *Android Security: Attacks and Defenses*. Auerbach Pub, 2013.

[18] Godfrey Nolan. Decompiler implementation. In *Decompiling Android*. Springer, 2012.

[19] smali. Typesmethodsandfields. https://code.google.com/p/smali/wiki/TypesMethodsAndFields. [Online; abgerufen am 21. Februar 2014].

[20] Jakob Straub. Art: Der turbo von android 4.4 kitkat. http://www.androidpit.de/dalvik-nachfolger-art-der-turbo-von-kitkat. [Online; abgerufen am 26. Februar 2014].