IT-Security

Erkennung von Android-Bibliotheken

Mittels statistischen Verfahrens

Diplomarbeit

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

eingereicht von

Arnold Dumas, BSc

is151505

im Rahmen des

Studienganges IT-Security an der Fachhochschule St. Pölten

Betreuung

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

Mitwirkung: FH-Prof. Dipl.-Ing. Dr. Paul Tavolato

St. Pölten, 15. Oktober 2017 _____ _____

(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*

Danksagungen

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich während der Anfertigung dieser Masterarbeit unterstützt und motiviert haben.

- Zuerst gebührt mein Dank Herrn FH-Prof. Dipl.-Ing. Dr. Sebastian Schrittwieser, Bakk., der meine Masterarbeit betreut und begutachtet hat. Für die konstruktive Kritik möchte ich mich bedanken.
- Ein besonderer Dank gilt Mag. Elisabeth Genser, die mir wertvolle Tipps bei der Erfassung dieser Diplomarbeit gegeben hat.
- Abschließend möchte ich mich bei meinen Eltern Dr. Brigitte Poumerol-Dumas und Dr. Jean-Bernard Dumas bedanken, die mir mein Studium in Österreich ermöglicht haben und mich immer unterstützt haben.

Kurzfassung

Android ist heutzutage ein weltweit verbreitetes Mobil-Betriebssystem, welches von sehr vielen Personen täglich verwendet wird, sei es bewusst auf ihren Handys oder unbewusst auf Geräten in Einkaufszentren oder im Kino. Für diese Plattform können Millionen Applikationen gekauft und benutzt werden. Es kommt heutzutage kaum eine Applikation ohne Verwendung von bereits existierenden Bibliotheken aus. Diese werden von Entwicklern eingebunden, um an die verschiedensten Funktionen zu kommen, sei es der Datenbank-Zugriff, soziale Netzwerke oder Werbung. Diese Bibliotheken sind jedoch des Öfteren veraltet und werden von den Applikation-Entwicklern selten aktualisiert, was ein Sicherheitsrisiko für den Endbenutzer darstellt.

Ziel dieser Masterarbeit ist es, bekannte Bibliotheken in einer bestehenden Applikation zu erkennen. Anhand dieser Informationen, welche Sicherheitslücken sich in den angewendeten Bibliotheken befinden, können dann Applikationen möglicherweise ausgenutzt werden.

Diese Arbeit unterteilt sich in verschiedene Teile. Zum einen soll ein Überblick über das Android-Ökosystem geschaffen und zum anderen ein Konzept entwickelt werden, welches ein solches Erkennungsverfahren erlaubt. Dieses Erkennungsverfahren wird dann in Java implementiert, diese Implementierung wird ausführlich kommentiert und erklärt, damit sie eine Basis weiterer Arbeiten darstellen kann. Im letzten Kapitel werden die Ergebnisse und deren Genauigkeit präsentiert.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation 1

1.2 Android-Applikationen 2

1.3 Bibliotheken 3

1.4 Fragestellung 3

2 Android-Ökosystem **6**

2.1 Aufbau einer Bibliotheken 6

2.2 Inhalt eines APK-Files 8

2.3 Code Obfuskation 11

2.4 Extrahieren des Quellcodes 13

3 Konzept **18**

3.1 Code-Cloning-Detection 18

3.2 Definition einer Filtering-Methode 21

3.3 Komplettes Erkennungsverfahren 24

3.4 Analyse der Bibliotheken 25

3.5 Füllung der Datenbank 26

3.6 Analyse der Applikation 27

3.6.1 Berechnung der Matching-Tabelle 28

3.6.2 Optimierung der Ergebnisse 29

3.6.3 Code-Cloning-Detection 30

3.7 Auswertung der Ergebnisse 34

3.8 Erstellung des Reports 35

4 Implementierung **36**

4.1 Technische Umgebung 36

4.2 Code-Architektur 37

4.3 Libparser 40
4.3.1 Extrahieren 41
4.3.2 Parsing und Analyse 42
4.3.3 Speichern der Statistiken 46
4.4 SQL-Datenbank 47
4.5 Apkparser 50
4.5.1 Vom APK-File zu Java 50
4.5.2 Vergleich 53
4.5.3 Optimierungen 55
4.6 Code-Cloning 57
4.6.1 Einbindung in das Projekt 57
4.6.2 Auswertung der Ergebnisse 61
4.7 Erstellung des Reports 62

5 Tests und Ergebnisse 65
5.1 Erste Testphase 65
5.2 Zweite Testphase 66
5.3 Dritte Testphase 68

6 Verfügbarkeit und Anwendbarkeit 70
6.1 Einrichtung der SQL-Datenbank 70
6.2 Konfiguration von LibParser 70
6.3 Konfiguration von ApkParser 71

7 Schlußfolgerung 73

Abbildungsverzeichnis 74

Tabellenverzeichnis 75

Literatur 78

1 Einleitung

1.1 Motivation

Im März 2017 ist Android das weltweit meistverwendete Betriebssystem geworden, mit mehr als 1,6 Milliarden aktiven Geräten [1]. Jedoch leidet Android unter verschiedenen Problemen. Das eine Problem ist die sogenannte Fragmentierung, das bedeutet, dass sehr viele verschiedene Versionen von Android zum selben Zeitpunkt betrieben werden.

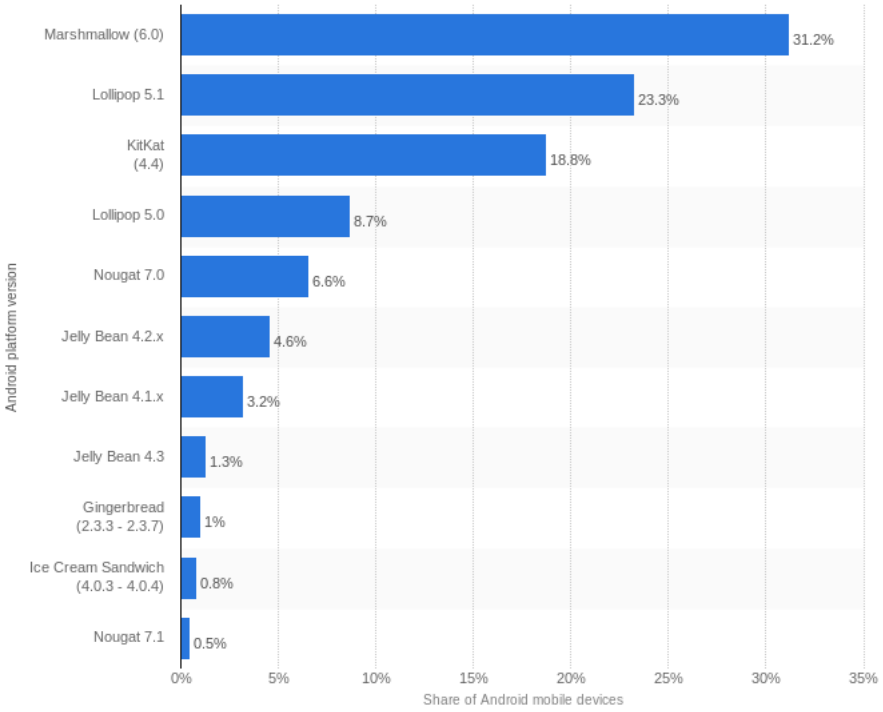


Abbildung 1.1: Android-Fragmentierung

Überstehend ist ein Diagramm abgebildet, welches die Fragmentierung der verschiedenen Android-Versionen schildert. Ich möchte an dieser Stelle ein kleines Beispiel einbringen. Samsung, der weltweit führende Android-Handys-Hersteller, bietet zurzeit 13 Handy-Modelle an, welche weltweit von rund

[1] https://www.forbes.com/sites/leemathews/2017/02/20/this-is-how-google-keeps-1-6-billion-android-devices-safe/

200 Anbietern verkauft werden. Jeder Telekom-Anbieter liefert eine an seine Bedürfnisse angepasste Android-Version, welche marktspezifische Applikationen beinhaltet. Oft ist es so, dass vom Verkaufspreis die Support-Periode abhängt, weil es für den Anbieter aufwändig ist, die neuen Android-Versionen anzupassen. [2]

Generell gesehen können Android-betreffende Sicherheitlücken in drei Kategorien unterteilt werden:

- Zum einen kommen die normalen 0-day Exploits im Linux-Kernel. Da Android den Linux-Kernel als Grundbaustein hat, können diese Linux-spezifischen Exploits ausgenutzt werden [9] [10]. Letztes Jahr wurde zum Beispiel eine Sicherheitslücke im Kernel von der Firma Perception Point gefunden [3], von der 66% aller Android-Geräte betroffen waren.

- Dem Android-Betriebssystem müssen Treiber zur Verfügung gestellt werden, damit es mit der Handy-Hardware überhaupt kommunizieren kann. Diese Treiber werden von den verschiedenen Handy-Herstellern entwickelt und stellen einen weiteren Angriffsvektor dar.

- Ein anderer Angriffsvektor bildet aber auch das Applikation-Ökosystem. Android-Applikationen bauen auf Bausteinen auf, welche Bibliotheken genannt sind. Diese werden von den Entwicklern ausgesucht, um die angestrebten Funktionen umsetzen zu können. In diesen Bausteinen befinden sich manchmal bekannte Sicherheitslücken, welche dann von Angreifern im laufenden Betrieb ausgenutzt werden können.

1.2 Android-Applikationen

Heutzutage befinden sich im Google Play Store und anderen Märkten zahlreiche Applikationen, die nahezu jeden Zweck erfüllen können. Obwohl all diese Applikationen auf Android-Handys beziehungsweise Tablets laufen, werden diese mit den verschiedensten Technologien und Programmiersprachen entwickelt. Dem Endbenutzer ist es im Prinzip egal, in welcher Programmiersprache die Applikation entwickelt wurde, weil er das kaum bis gar nicht bemerkt.

Am Anfang der Android-Ära wurden alle Android-Applikationen in der Programmiersprache Java geschrieben, welche von Google bevorzugt wird. Jedoch wurden in den letzten Jahren verschiedene Technologien entwickelt, die einem die Möglichkeit geben, mit beinahe jeder Programmiersprache Android-Applikationen entwickeln zu können. Nennenswert sind auf jeden Fall folgende Programmiersprachen: C, C++, C#, HTML5/CSS3 und JavaScript. Die von Google bevorzugte Programmiersprache ist Java, mit welcher die Entwicklung sogenannter Native Apps ermöglicht wird. Obwohl solche Applikationen

[2] https://www.wired.com/2017/03/good-news-androids-huge-security-problem-getting-less-huge/

[3] http://perception-point.io/2016/01/14/analysis-and-exploitation-of-a-linux-kernel-vulnerability-cve-2016-0728/

eigentlich normale Java-Programme sind, werden diese aber in einer von Google entwickelten virtuellen Maschine ausgeführt, *Dalvik*. Wie oben schon erwähnt, stellen C und C++ eine sehr interessante Alternative zu Java dar, insofern, dass sehr viele populäre externe Bibliotheken in C++ geschrieben sind und diese von Google offiziell unterstützt werden, etwa durch die NDK [4] (Native Development Kit). Es muss dann keine Umschreibung von C/C++-Code in Java stattfinden. Eine weitere Möglichkeit bilden HTML5/CSS3 und JavaScript, mit welchen sogenannte hybride Applikationen zustande gebracht werden. Im Prinzip sind diese kleinen Webanwendungen, mit denen aber, durch einen Compatibility-Layer, der Zugriff auf die Handy-Hardware gewährleistet werden kann.
Der Inhalt meiner Masterarbeit beschränkt sich allerdings auf die in Java geschriebenen Applikationen, also um diese *Native Apps*. Diese können als klassische Java-Programme betrachtet werden, obwohl sie natürlich durch die von Google zur Verfügung gestellte SDK extra Klassen verfügen, etwa wenn es um die Kamera, das Mikrofon oder den Location-Sensor geht. Wie in der klassischen Java-Welt stehen Entwicklern in der Android-Welt tausende Bibliotheken zur Verfügung, auch jene, die nicht für Android konzipiert sind, können eingebunden werden.

1.3 Bibliotheken

Aus der Sicht eines Entwicklers, der sein Konzept in Form einer Applikation auf den Markt bringen möchte, ist es am wichtigsten, die Idee hinter der Applikation umzusetzen. Eine Verbindung zu den Facebook-Servern herzustellen, oder animierte Knöpfe selber zu programmieren ist eine reine Zeitverschwendung, insofern, dass sich schon sehr viele Entwickler mit diesen Problemen auseinandergesetzt haben. Um das Rad nicht neu erfinden zu müssen, werden sogenannte Bibliotheken eingebunden, welche Entwicklern gereiften und getesteten Codes zur Verfügung stellen.
Bei Java-Applikationen werden in Java-geschriebene Bibliotheken eingebunden. In der Regel ist der Code so dokumentiert, dass sich mit Hilfe vom javadoc eine hilfreiche und brauchbare Dokumentation generieren lässt. Bei nativen Apps, also in C oder C++ geschrieben, werden ebenfalls in C oder C++ geschriebene Bibliotheken verwendet.

1.4 Fragestellung

Am Anfang wurden sehr viele Applikationen von Hobbyisten entwickelt. So ist es aber längst nicht mehr und viele Firmen erkämpfen sich den Markt. Da sind natürlich Time-To-Market und Kostenbegriffe von höchster Bedeutung geworden. Der Spruch *If it ain't broke, don't fix it* wird geprägt. In gutem Deutsch,

[4] https://developer.android.com/ndk/index.html

solange die Applikation reibungslos funktioniert, wird nichts korrigiert, weil die Applikation eben gut funktioniert. Vor allem auf diesem Markt ist Sicherheit keine Funktion, die verkauft werden könnte. Es ist aber so, dass Sicherheitslücken in diesen Bibliotheken nach deren Entdeckung veröffentlicht werden. Angenommen, dass die Applikation MeinTollesSpiel auf der Bibliothek MeineTolleBibliothek aufbaut, und dass diese Bibliothek eine Schwachstelle aufweist, dann könnte die Applikation über die Schwachstelle der Bibliothek ausgenutzt werden. In diesem Fall müsste also keine an die Applikation angepasste Arbeit geleistet werden. Es könnte nur noch nach bekannten Bibliotheken gesucht werden, die bestimmte Schwachstellen aufweisen.
Ziel dieser Masterarbeit ist es, anhand einer bestehenden Applikation herauszufinden, welche Bibliotheken samt deren Version verwendet wurden. Die Information, dass eine Applikation auf einer spezifischen Bibliothek Aufbaut, könnte von höchster Priorität sein, wenn für diese Bibliothek Sicherheitslücken ans Tageslicht kommen. In dieser Arbeit werden nur die in Java geschriebenen Applikationen in Betracht gezogen, weil keine allgemeinen Methoden entwickelt werden, die mehrere oder alle Technologien abdecken könnten. Außerdem wird die überwiegende Mehrheit aller im Google Play zur Verfügung gestellte Applikationen in Java geschrieben.
Ähnliche Funktionen werden von zwei Produkten angeboten:

- Zum einen wurde von chinesischen Forschern ein Programm namens LibRadar entwickelt, welches bekannte Bibliotheken in einer fertigen Applikation finden kann. Die Implementierung wurde im [15] beschrieben. Dessen Ansatz basiert auf dem Smali-Code und arbeitet auf einer Maschinennahen Ebene, weil Smali-Code ASM Code sehr ähnlich ist. Aus diesem Grund ist die Suche extrem schnell und wird auch über eine Website als Service angeboten [5]. Die Python-Implementierung wurde auf Github zur Verfügung gestellt [6].

- Zum anderen bieten verschiedene Firmen einen funktionsmäßig gleichen Service an. Die Firma ADN-Source ist einer der am Markt vertretenden Anbieter.

Keine dieser beiden Lösungen ist aber befriedigend. Von Firmen angebotene Lösungen sind kostenpflichtig und deren Quellcode wird nicht offengelegt. Diese können also nicht von Dritten weiterentwickelt beziehungsweise angepasst werden. LibRadar ist ebenfalls nicht zufrieden stellend, weil die Bibliotheken kompiliert werden müssen. Das heißt, für jede Bibliothek, die dann erkannt werden soll, muss eine Applikation kompiliert werden, die eben auf dieser Bibliothek in einer spezifischen Version aufbaut. Es soll ein Lösungansatz entwickelt werden, welcher keine Kompilierung erfodert und welcher erweitert beziehungsweise angepasst werden kann.

[5] http://pkuos.org/
[6] https://github.com/pkumza/LibRadar

Diese Masterarbeit gliedert sich in fünf Teile. Zuerst werden im ersten Teil Android-spezifische Begriffe anhand von Beispielen erläutert. Es wird ebenfalls der Kompilierungsprozess einer Android-Applikation vorgestellt. In einem zweiten Teil wird ein Lösungsansatz offeriert und detailliert beschrieben, welcher zur Bibliothekenerkennung dient. In einem dritten Teil wird dem Leser dessen Java-Implementierung und die technische Umgebung näher gebracht. In einem weiteren Kapitel werden Ergebnisse präsentiert und deren Genauigkeit geprüft. Am Ende wird beschrieben, wie die Applikation installiert und weiterentwickelt werden könnte.

2 Android-Ökosystem

Bevor ein Konzept zurstande kommen kann, müssen noch die wichtigsten Teile des Android-Ökosystems ausführlich beschrieben werden. Zuerst werden Bibliotheken vorgestellt, sowohl aus einer technischen Perspektive als aus der Sicht eines Entwicklers. Dann wird der Kompilierungsprozess einer in Java-geschriebenen Applikation beschrieben, weil es im Endeffekt nur darum geht, diesen Weg bis hin zum Quellcode zurückzugehen. Dies deswegen, da mit dem Quellcode der Applikation im Prinzip gesagt werden kann, auf welchen Komponenten die Applikation aufbaut. Es werden auch die vom Entwickler eingesetzten Techniken vorgestellt, die dazu dienen, dass der Weg unumkehrbar gemacht wird.

2.1 Aufbau einer Bibliotheken

Aus technischer Perspektive können Bibliotheken als eine Sammlung verschiedener Klassen (hier Java-Klassen) betrachtet werden. Diese können auf mehreren Wegen eingebunden werden: als ZIP-Archive im Projekt-Verzeichnis, im Build-System oder in einem 3rd-Party-Verzeichnis. Bei Android-Entwicklern sind zwei Build-Systems besonders populär: Maven und Gradle. Das Letzte ist das de facto Build-System in der von Google veröffentlichten Programmierungsumgebung Android Studio.
Android-Bibliotheken stellen sehr viele verschiedene Funktionen zur Verfügung: von der Integration zu sozialen Netzwerken über Werbungen bis hin zu Datenbank-Zugriff. Unter dieser riesigen Anzahl an Bibliotheken ist es so, dass sich ein paar Bibliotheken in sehr vielen Applikationen befinden. Die nächste Tabelle stammt aus dem LibRadar Projekt, welches präsentiert wurde und stellt, nach Analyse von mehr als hunderttausenden Anwendungen, die zehn populärsten Bibliotheken vor:

Tabelle 2.1: 10 populärste Android-Bibliotheken

Name	Package	Typ
Android Support v4	android/support/v4	Development Aid
Google Ads	com/google/ads	Advertisement
Google GMS	com/google/android/gms	Development Aid
Facebook	com/facebook	Social Network
Google Analytics	com/google/analytics	Mobile Analytics
Google Gson	com/google/gson	Development Aid
Google Play	com/android/vending	App Market
Apache Http	org/apache/http	Development Aid
Apache Common	org/apache/commons	Development Aid
Google GCM	com/google/android/gcm	Development Aid

Angenommen, dass ein Entwickler die Bibliothek Gson in der Version 2.8.0 in sein Projekt einbinden will, dann muss bei Maven das pom.xml-File folgendermaßen aussehen:

```
<!-- https://mvnrepository.com/artifact/com.google.code.gson/gson -->
<dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.8.0</version>
</dependency>
```

Wird dennoch Gradle verwendet, dann im build.gradle:

```
dependencies {
    // https://mvnrepository.com/artifact/com.google.code.gson/gson
    compile group: 'com.google.code.gson', name: 'gson', version: '2.8.0'
}
```

Eine weitere Möglichkeit wäre jene, das Jar-File direkt in das Projektverzeichnis zu kopieren. Jedoch sind immer drei Elemente erkennbar, unabhängig von der gewählten Methode:

- der Packagename ("com.google.code.gson")
- die Version ("2.8.0")

- der Name ("Gson")

Diese drei Elemente bilden einen Kenner, der eine bestimmte Bibliothek in einer bestimmten Version kennzeichnet. Es sind genau diese Informationen, auf die wir aus einer fertigen Applikation kommen wollen. Es ist eigentlich so, dass es mit diesen drei Werten möglich ist, nach von Sicherheitsforschern entdeckten Schwachstellen suchen zu können, falls jemals eine gefunden wird.

2.2 Inhalt eines APK-Files

Klassische Java-Applikationen werden oft als ausführbare JAR-Dateien veröffentlicht oder als .dmg, .exe, manchmal auch in Form eines .sh-Skript, je nach gezieltem Betriebssystem. Bei Android ist es so, dass Applikationen als APK-Files zur Verfügung gestellt werden (Dateinamenserweiterung .apk oder auch .xapk und MIME-Type application/vnd.android.package-archive). Das APK-Format basiert auf dem Jar-Format, welches selber eine Erweiterung des ZIP-Formates ist. Ein APK-File ist also einem ZIP-Archiv gleichzustellen und beinhaltet unter anderem folgende Files und Verzeichnisse [5]:

- AndroidManifest.xml: Ein zusätzliches AndroidManifest.xml-File in welchem der Namen, die Berechtigungen, die API-Version stehen
- lib/: In diesem Verzeichnis liegt der kompilierte Code in Form von .so Files.
- classes.dex: Enthält den kompilierten Java-Code der Applikation.
- META-INF/: Dieses Verzeichnis enthält folgende Files CERT.RSA, CERT.SF, and MANIFEST.MF, welche die digitalen Signaturen der Applikation sind.

Das für uns interessanteste File ist das classes.dex Files. Wie schon oben erwähnt, sind Android-Applikationen an sich Java-Programme, welche dann logischerweise aus Java-Klassen und manchmal auch aus externen Jar-Dateien (etwa wenn externe Bibliotheken verwendet werden) stammen. Bei einem ganz klassischen Java-Programm werden Java-Klassen mit dem Tool javac in .class-Files (Bytecode) kompiliert. Diese werden dann in eine ausführbare Jar-Datei vereint.
Bei Android-Applikationen beginnt es nach demselben Schema, jedoch verwendet Android (beziehungsweise die Dalvik Maschine) ein anderes Instruktion-Set, Dalvik. Aus diesem Grund werden dann diese .class-Files und externe Bibliotheken in ein Dex-File umgewandelt. Dieses Dex-File enthält keinen nativen Code, sondern ausschließlich Bytecode. Mit der Dalvik virtuellen Maschine wird der Bytecode *just-in-time* in den nativen Code kompiliert. Dieser ganze Prozess wird im [3] im ersten Kapitel detailliert beschrieben.

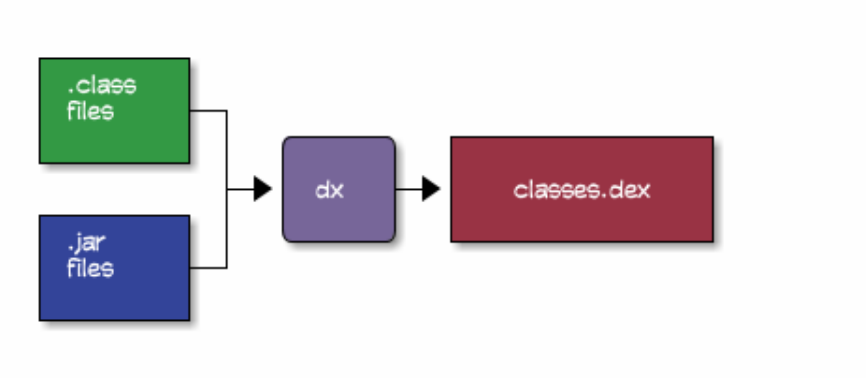


Abbildung 2.1: Generierung des classes.dex-Files

Aus diesem classes.dex-Files wird dann das fertige Applikation-File generiert, nämlich das APK-File:

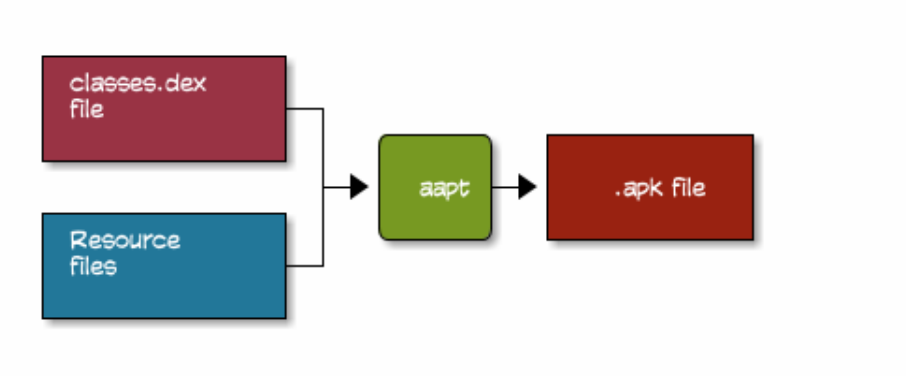


Abbildung 2.2: Generieung des APK-Files

Ab Android 5.0 (Lollipop) wurde Davik durch die Android Runtime (kurz ART) ersetzt. Im Gegenteil zu Dalvi wird bei ART der Bytecode bei der Installierung der Applikation in einen nativen Code kompiliert [8]. Dies bringt einige Vorteile mit sich:

- Geringe CPU-Last, weil der Bytecode nur einmal kompiliert werden muss, was einen geringeren Energieverbrauch bedeutet.
- Die Applikation wirkt für den User schneller, weil der Code schon kompiliert ist.

Nachteilig fällt aber ART dadurch auf, dass die Installation einer neuen Applikation beziehungsweise einer Aktualisierung länger dauert, weil der Bytecode aus der Dex-Datei in einen nativen Code kompiliert wird.

Will der Entwickler seine Applikation im Google Play zur Verfügung stellen, muss dann aus Sicherheitsgründen das APK-File signiert werden. Dies erfolgt basierend auf einer vom Inhalt des APK-Files

abhängigen Prüfsumme und einem Schlüssel, der im Voraus generiert werden muss [7]. Damit eine einfache Handhabung für den Entwickler garantiert ist, stellt das Android-Studio dem Entwickler eine GUI zur Erzeugung des Schlüssels her. Dafür werden nur einige Informationen über den Entwickler und die Applikation benötigt.

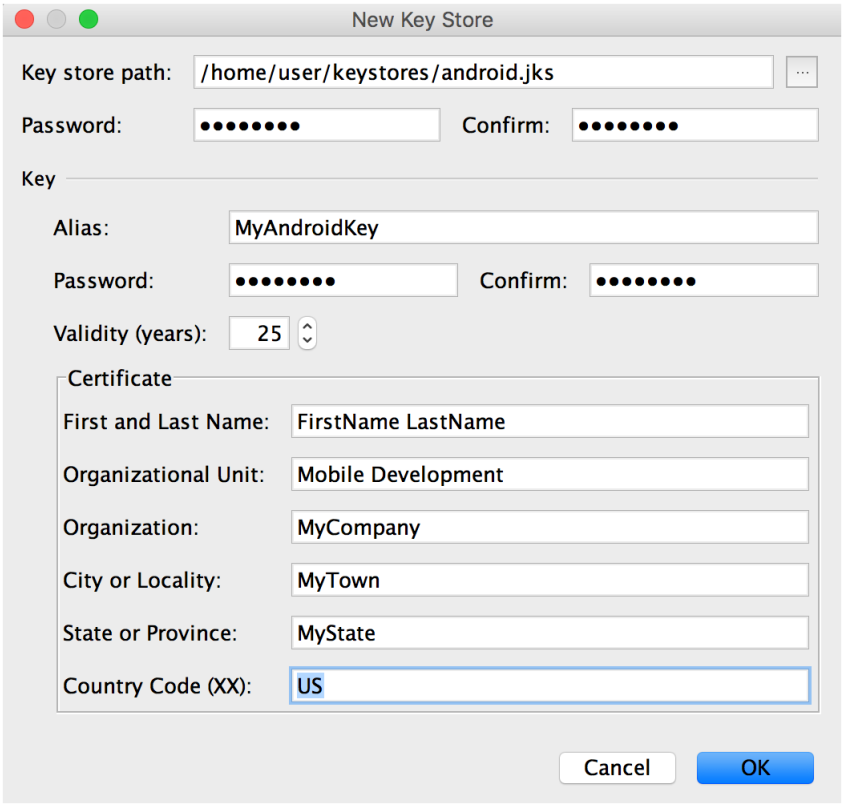


Abbildung 2.3: Erzeugung eines neuen Key

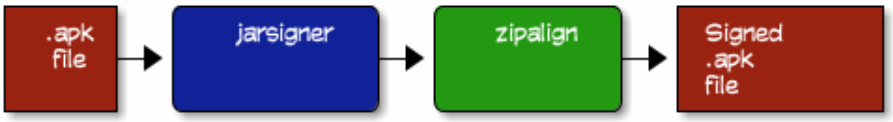


Abbildung 2.4: Signieren des APK-Files

Hier wird ein aus der Java-Welt stammendes Tool angewendet, nämlich jarsigner. Dieses ist Teil des Oracle's Java Development Kit und wird häufig verwendet, um Jar-Files zu signieren. Ein besonderes Merkmal bei Android ist aber, dass das signierte APK-File von zipalign gehandelt werden muss, um das Alignment zu prüfen beziehungsweise zu korrigieren. Dadurch, dass das Alignment optimiert wurde,

wird sichergestellt, dass der RAM-Verbrauch während der Ausführung der Applikation so gering wie möglich bleibt.

2.3 Code Obfuskation

Eine große Gefahr im Hinblick auf die Sicherheit bei üblichen Java-Programmen besteht darin, dass es relativ einfach ist, an den originalen Code zu gelangen. Bei Android-Applikationen ist es nicht anders. Dass der originale Code aus dem APK-File extrahiert werden kann, ist für den Entwickler ein großes Problem, insofern, dass es für Angreifer dann leichter ist, maßgeschneiderte Angriffe vorzubereiten oder nachSicherheitslücken zu suchen. Es ist für den Angreifer dann auch möglich, nach bekannten Sicherheitslücken zu suchen, die in den verwendeten Bibliotheken bekanntgegeben wurden. Mit Bad-Publicity und möglichen Verlusten durch verstärkte Konkurrenz ist dann zu rechnen. Um dem Vorzubeugen, ist der Einsatz eines Obfuskator der traditionelle und effektivere Lösungsansatz.
Im Prinzip geht es darum, den Code absichtlich so zu modifizieren, dass er dann für Menschen schwer verständlich beziehungsweise rückgewinnbar wird. Ein Programm P verschleiern heißt, dieses Programm in ein anderes Programm P' umzuwandeln, aus dem sich Informationen viel schwerer extrahieren lassen [17]. Obfuskation ist kein neues Thema und wurde schon in vielen wissenschaftlichen Papers behandelt und diskutiert. Um das Ziel zu erreichen, können verschiedene Wege eingegangen werden, unter anderem Folgende [18]:

- Verändern des Kontrollflusses, es werden zum Beispiel if-Branches eingefügt.
- Einfügen von redundantem Code, es wird unnötiger Code eingefügt, um den Leser zu täuschen.
- Variablensubstitution, hier werden Variablennamen einfach durch zufallsgenerierte Namen ersetzt, my_counter wird zum Beispiel o12jda.

Manche Funktionen kann man sich leicht vorstellen, zum Beispiel, wenn es um Variablensubstitution geht. Da wird eine Variable namens mein_zähler durch eine eine andere Variable namens IOdksdj in der ganzen Codebase ersetzt. Eine solche Änderung führt natürlich zu keiner Änderung der Funktionsweise, solange dieser Variablenname noch nicht verwendet wurde. Es werden auch komplexere Ersetzungen angewendet wie Opaque Predicates (undurchsichtige Prädikate), wo eine einfache Zuweisung durch eine andere Instruktion ersetzt wird. Der Wert der ersetzenden Instruktion ist dann nicht einfach vorauszusagen. Nun folgt ein Beispiel:

```
1 // Originaler Code
2 boolean flag = true;
```

```
3
4 // Nach Obfuskation
5 boolean flag = System.currentTimeMillis() > 0;
```

Um zu verstehen, welche Auswirkungen dieses Programm unter anderem hat, sehen wir uns folgendes Beispiel an. Hier wurde eine einfache Java-Klasse geschrieben, die von einer Bibliothek abhängt. Diese ist Teil einer kleinen Android-Applikation, die mit Proguard obfuscated wurde. Der Code ist recht einfach, es wird nur eine Klasse definiert, welche eine einzige Methode implementiert, in der zwei Objekte erzeugt werden. Methoden von diesen Objekten werden an verschiedenen Stellen aufgerufen.
Der Original-Code ist:

```
1  package fhstp.ac.at.gson_2_8_test_no_obfuscation;
2
3  import android.os.Bundle;
4  import android.support.v7.app.AppCompatActivity;
5  import com.google.gson.GsonBuilder;
6
7  public class MainActivity extends AppCompatActivity {
8      protected void onCreate(Bundle savedInstanceState) {
9          super.onCreate(savedInstanceState);
10         setContentView((int) R.layout.activity_main);
11         Object albums = new Albums();
12         albums.title = "Free Music Archive - Albums";
13         albums.message = "";
14         albums.total = "11259";
15         albums.total_pages = 2252;
16         albums.page = 1;
17         albums.limit = "5";
18         System.out.println(new GsonBuilder().create().toJson(albums));
19     }
20 }
```

und jetzt der mit Proguard obfuscated Code:

```
1  package fhstp.ac.at.gson_2_8_test_no_obfuscation;
2
3  import a.a.a.g;
4  import android.os.Bundle;
5  import android.support.v7.app.c;
6
7  public class MainActivity extends c {
8      protected void onCreate(Bundle bundle) {
9          super.onCreate(bundle);
10         setContentView((int) R.layout.activity_main);
11         Object aVar = new a();
```

```
12          aVar.a = "Free Music Archive - Albums";
13          aVar.b = "";
14          aVar.d = "11259";
15          aVar.e = 2252;
16          aVar.f = 1;
17          aVar.g = "5";
18          System.out.println(new g().a().a(aVar));
19      }
20  }
```

Anhand dieses Beispiels kann man feststellen, dass folgende Elemente umbenannt wurden:

- die Klassennamen, die Klasse AppCompatActivity heißt jetzt c.
- die Methodennamen, die Methode create() heißt jetzt a() und toJson() ebenfalls a().
- die Variablennamen, die Variable albums heißt in der modifizierten Version aVar.
- die Import-Statements – da die Klassen umbenannt wurden, müssen auch die Import-Statements umgeschrieben werden. Es ist auch zu anzumerken, dass ebenso die Reihenfolge der Import-Statements geändert wurde.

Technisch gesehen bleibt nach Anwendung eines solchen Programmes das Extrahieren des Codes möglich und einfach. Aber dadurch, dass der Code so sehr umgeschrieben wurde, lassen sich die Aufgaben der Klassen beziehungsweise der Methoden nicht mehr von ihren Namen ableiten. Es ist natürlich schwer, die Rolle einer Methode nachzuvollziehen, wenn diese zum Beispiel aaaa() statt toJson() heißt. Auch bei Klassen ist es nicht mehr möglich, deren Rollen anhand deren Namen zu ermitteln.
Im Android-Ökosystem wird sehr oft Proguard[1] eingesetzt. Proguard ist sehr populär und gilt als de facto Standard, seitdem er Teil des Android-SDK geworden ist. Dieser lässt sich mit einer Konfigurationsdatei an den Bedarf des Entwicklers anpassen. Mit Android Studio und Gradle ist die Integration mittlerweile tief und vereinfacht den ganzen Prozess für den Entwickler. Die Konfiguration erfolgt nämlich direkt im build.gradle und/oder in einer separaten Datei, proguard-rules.pro.

2.4 Extrahieren des Quellcodes

Wie bereits ausführlich im Abschnitt 2.2 geschildert, enthält jedes APK-File einen Java-Code, jedoch in einer kompilierten Form. Damit diese Merkmale überhaupt berechnet werden können, müss zuerst der Java-Code aus dem APK-File extrahiert werden. Es ist nämlich so, dass wir die zu analysierenden

[1] https://www.guardsquare.com/en/proguard

Applikationen in Form eines APK-Files bekommen. Aus dem APK-File lässt sich der Code in zwei Formen extrahieren. Mit dem Tool namens apktool[2] kann der Smali-Code extrahiert werden. Hierbei handelt sich um einen ASM-ähnlichen Code, der für einen Java-Entwickler schwer lesbar ist.

Folgendermaßen kann der Smali-Code aus einem APK-File extrahiert werden:

```
arnold@acapulco: java -jar apktool_2.2.4.jar d test_application.apk
I: Using Apktool 2.2.4 on test_application.apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file:
 ↪  /home/arnold/.local/share/apktool/framework/1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...
```

Nach Ausführung dieses Befehls liegt ein Smali-Verzeichnis bereit, welches alle Smali-Files enthält beziehungsweise Unterverzeichnisse.

Untenstehend ist ein Beispiel eines solchen Smali-Files:

```
.class public Lcom/packageName/example;
.super Ljava/lang/Object;
.source "example.java"

.field public final someInt:I

.field public final someBool:Z

.method public constructor <init>(ZLjava/lang/String;I)V
        .locals 6

        .parameter "someBool"
        .parameter "someInt"
        .parameter "exampleString"

        .prologue
        .line 10
        invoke-direct {p0}, Ljava/lang/Object;-><init>()V

        const-string v0, "i will not fear. fear is the mind-killer."

```

[2] https://ibotpeaches.github.io/Apktool/

```
22          const/4 v0, 0xF
23
24          new-instance v1, Ljava/lang/StringBuilder;
25          const-string v2, "the spice must flow"
26          invoke-direct {v1, v2},
             ↪  Ljava/lang/StringBuilder;-><init>(Ljava/lang/String;)V
27
28          invoke-virtual {v1, p1},
             ↪  Ljava/lang/StringBuilder;->append(Z)Ljava/lang/StringBuilder;
29          move-result-object v1
30
31          const-string v2, "some random string"
32          invoke-virtual {v1, v2},
             ↪  Ljava/lang/StringBuilder;->append(Ljava/lang/String;)Ljava/lang/StringBuilder;
33          move-result-object v1
34
35          invoke-virtual {v1}, Ljava/lang/StringBuilder;->toString()Ljava/lang/String;
36          move-result-object v1
37
38          const-string v0, "Tag"
39          invoke-static {v0, v1},
             ↪  Landroid/util/Log;->d(Ljava/lang/String;Ljava/lang/String;)I
40          move-result v0
41
42          invoke-static {}, Ljava/lang/System;->currentTimeMillis()J
43          move-result-wide v2
44          const-wide/16 v4, 0x300
45          div-long/2addr v2, v4
46          long-to-int v2, v2
47
48  .line 12
```

In dieser Arbeit ist für uns der Smali-Code ziemlich irrelevant, weil er zu sehr vom ursprünglichen Quellcode der jeweiligen Bibliotheken abweicht. Zwar gibt es in der Forschung Konzepte, die entweder auf Smali beziehungsweise Low-level Code [22] oder Byte-Code basieren [23] [24] basieren, aber dafür müsste jede Bibliothek vor dem Erkennungsverfahren kompiliert werden. Die zweite Form, in der wir den Code bekommen können, ist der kompilierte Java-Code. Wie ausführlich beschrieben, wird die Applikation als ein Java-Programm kompiliert, jedoch mit einem anderen Instruktion-Set.

Um diesen Java-Code zu bekommen, wird ein mehrstufiges Verfahren benötigt. Zum einen wird das APK-File mit dem unzip-Tool entpackt, als ob es sich um ein ganz normales ZIP-Archiv handeln würde. In diesen extrahierten Dateien suchen wir nach dem classes.dex-File, welches den Code enthält [4].

Es wurde schon erklärt, dass Android-Applikationen innerhalb einer virtuellen Maschine, Dalvik, ausgeführt werden. Aus diesem Byte-Code ist es möglich, den korrespondierenden Java-Code, durch einen Dekompilierungsprozess herauszubekommen. Dank diesem kann auf den originalen Code zugegriffen

werden. Da sich sowohl eine Vielzahl von Entwicklern als auch die Wissenschaft für das Java-Ökosystem interessiert, wurden schon einige Tools zur Dekompilierung von Java-Anwendungen entwickelt. Alle Tools folgen im Prinzip demselben theoretischen Schema, welches sich in sechs Stufen unterteilt:

1. Read opcodes: Am Anfang werden die Opcodes, also die Bytecode-Instruktionen gelesen.
2. Interpret opcode behavior: Deren Verfahren wird interpretiert.
3. Identify Java-language *idioms*: Danach wird nach bekannten Java-Idioms gesucht.
4. Identify patterns of control-flow: Es wird nach Loops- oder If-Strukturen gesucht.
5. Generate Java-language statements: An dieser Stelle werden die zu den gelesenen Bytecode-Opcodes passenden Java-Zeilen generiert.
6. Format and output to file: Diese Java-Zeilen werden dann in Files geschrieben. An dieser Stelle werden auch die passenden Verzeichnisse erstellt.

Bei Android-Applikationen ist der Startpunkt des Dekompilierungsprozesses das classes.dex-File. Aus diesem File können mehrere Tools die ganzen Java-Files und deren Baumstruktur rekonstruieren und darstellen. Jadx [3] ist eines davon, kann sowohl als GUI-Applikation als auch in der Shell über Kommandobefehle ausgeführt werden. Obwohl es von einem kleinen Team entwickelt und gewartet wird, unterstützt es die gängigsten. Der Quellcode, der unter der Apache Lizenz veröffentlich wird, und der User Guide werden auf Github [4] gehostet.
Um aus einem Dex-File die Java-Files im Verzeichnis mein_verzeichnis zu bekommen, lautet der Befehl:

```
1 jadx -d mein_verzeichnis classes.dex
```

Je nachdem, ob die Applikation obfuscated wurde, kann die originale Baumstruktur aller Files abgebildet werden, wie auf dem folgenden Bild zu sehen ist. Die Namen der Java-Files werden ebenfalls beibehalten, wenn die Applikation nicht obfuscated wurde.

[3] https://skylot.github.io/jadx/
[4] https://github.com/skylot/jadx

Abbildung 2.5: Kompilierte Applikation ohne Obfuskation in jadx-gui geöffnet

Hiert handelt es sich um eine reine Test-Applikation, die nur auf einer externen Bibliothek aufbaut, nämlich Google Gson in Version 2.8. Deren Quellcode befindet sich auf der CD am Ende dieser Arbeit Wurde die Applikation jedoch obfuscated, kann man sich ein ähnliches Ergebnis erwarten:

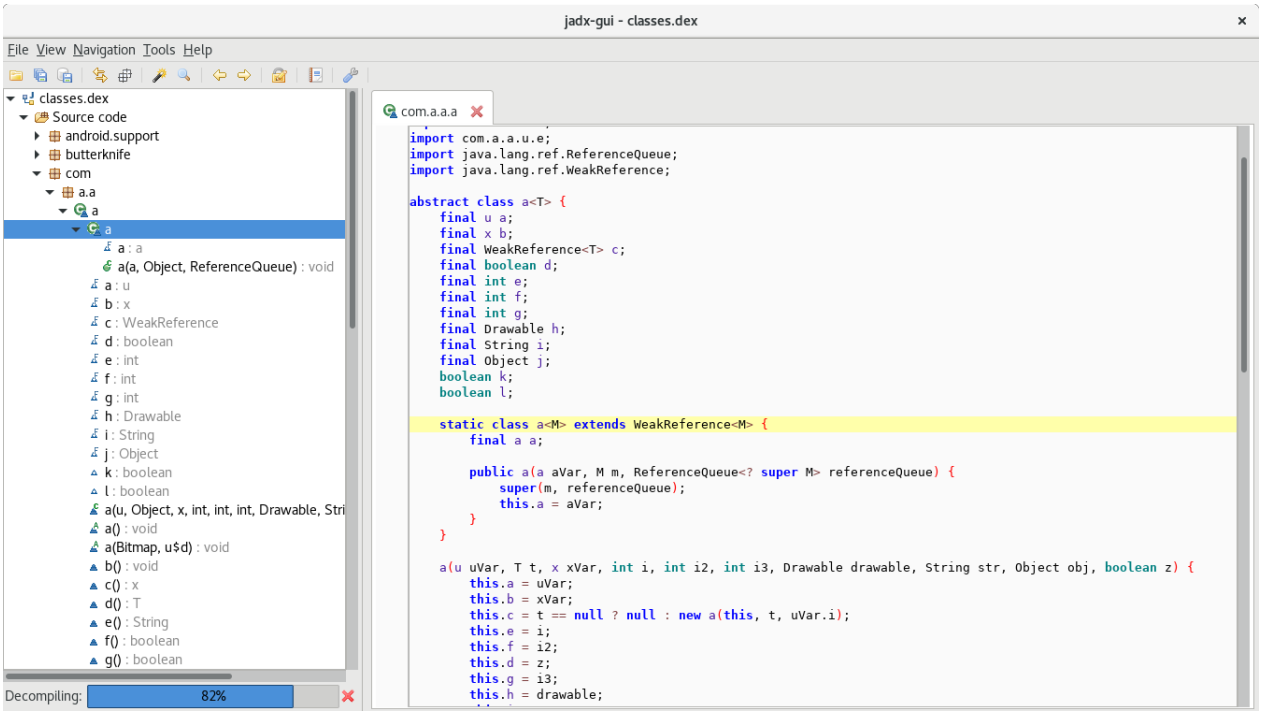


Abbildung 2.6: Kompilierte Applikation mit Obfuskation in jadx-gui geöffnet

3 Konzept

Wie im Abschnitt 2.4 beschrieben, kann für jede Applikation der Java-Code extrahiert werden. Ich möchte an dieser Stelle auch noch anmerken, dass dieser Java-Code nicht zu hundert Prozent dem originalen Quellcode gleichzustellen ist. Und dies aus mehreren Gründen. Zum einen kann der Java-Kompiler bestimmte Funktionen, die nie aufgerufen werden, einfach ignorieren. Und zum anderen wurde sehr wahrscheinlich der Code obfuscated. Jedoch stellt dieser Quellcode den Ausgangspunkt unseres Erkennungsverfahrens dar. Alle Bibliotheken, die wir in fertigen Applikationen erkennen wollen, sind ebenfalls in Java geschrieben. Im Prinzip geht es also um den Vergleich zwischen verschiedenen Quellcode-Files, hier Java-Klassen.

3.1 Code-Cloning-Detection

Der Vergleich von verschiedenen Codeabschnitten nennt sich in der Literatur *Code-Cloning-Detection* und ist an sich kein neuartiges Thema, sondern ein seit vielen Jahren diskutiertes Forschungsthema. Im Prinzip geht es darum, den Ähnlichkeitsgrad zwischen zwei Codeabschnitten zu berechnen. Jedes Tool liefert dann einen Wert zurück, welcher besagt, wie ähnlich zwei Codeabschnitte zueinander sind. Dieses praxisnahe Thema findet prinzipiell zwei verschiedene Anwendungsfelder. Zum einen kann *Code-Cloning-Detection* in der Softwareentwicklungswelt hilfreich sein, wenn es um Codededuplizierung geht. Aufgrund eines schwachen beziehungsweise zu schnell entworfenen Klassendiagramms kann es zu der Situation kommen, in der sich eine Funktion beziehungsweise eine Klasse an verschiedenen Stellen in der Codebasis befindet. Solche Funktionen werden in manchen kommerziellen Entwicklungsumgebungen angeboten, zum Beispiel in Intellij Idea[1] oder ReSharper[2] von der tschechischen Firma Jetbrains s.r.o. Ein weiteres Anwendungsfeld von *Code-Cloning-Detection* ist die Suche nach einem propriäteren Code beziehungsweise bestimmten Funktionen, Klassen oder auch Bibliotheken. Hier kann man sich folgende Situation leicht vorstellen: Eine Softwareentwicklungsfirma hat vor Kurzem einen ganz einzigartigen Algorithmus in einem zukunftsträchtigen Bereich entworfen und implementiert. Ein

[1] https://www.jetbrains.com/idea/

[2] https://www.jetbrains.com/resharper/

paar Monate später kommt ein sehr ähnliches Produkt auf den Markt, das von einem Mitbewerber verkauft wird. Wenn die rechtlichen Rahmenbedingungen gegeben sind, könnte die erste Firma das Produkt des anderen Konzerns analysieren und versuchen, herauszufinden, ob dieses das eigene Algorithmus beinhaltet.
Code-Cloning-Detection wurde schon sehr viel diskutiert, im Folgenden stütze ich mich auf folgende Papers:
Wir schon oben erwähnt, ist *Code-Cloning-Detection* bereits seit mehreren Jahren ein erarbeites Forschungsthema. Es wurden mit der Zeit mehrere Vorgehensweise entwickelt. Untersuchungen von Roy, Chanchal K., James R. Cordy und Rainer Koschke ([2]) weisen nach, dass all diese Methoden grundsätzlich in fünf Kategorien eingeteilt werden können:

- Text-basierend: Hierbei wird der originale Quellcode vor der tatsächlichen Analyse kaum bis gar nicht geändert. Dann werden die zwei zu vergleichenden Files Wort für Wort verglichen.
- Token-basierend: Wie bei Kompilers wird eine lexikalische Analyse des Quellcodes durchgeführt. Die daraus gesammelten Daten werden dann analysiert und es wird nach gemeinsamen Sequenzen gesucht.
- Tree-basierend: Bei solchen Methoden wird der abstrakte Syntaxbaum (Abstract Syntax Tree - AST) für jedes File konstruiert. Erst danach werden diese Bäume miteinander Blatt für Blatt verglichen.
- Metrics-basierend: Hierbei wird, wie bei Tree-basierenden Methoden, der abstrakte Syntaxbaum konstruiert. Jedoch werden danach verschiedene Werte (Metrics) für jeden Baum berechnet. Diese Werte, und nicht mehr die Bäume selbst werden im Endeffekt verglichen.
- Graphen-basierend: Da werden PDG (Programm Data Graphs) gebildet und verglichen. Es werden also Abhängigkeiten innerhalb des Programms untersucht.

Diese verschiedenen Methoden schließen sich einander nicht aus und können kombiniert werden. In diesem Fall wird von hybriden Methoden die Rede sein. Obwohl alle Methoden einem einzigen Ziel folgen, nämlich aus zwei oder mehreren Codeabschnitten den Ähnlichkeitsgrad zu berechnen, eignet sich in manchen Fällen eher die Methode A und in anderen Anwendungsfällen eher die Methode B. Roy, Chanchal K., James R. Cordy und Rainer Koschke haben in ihrer Arbeit über *Code-Cloning-Detection* diese Methoden in verschiedenen Anwendungsfällen angewendet.
An erster Stelle definieren sie vier Szenarien, inwieweit ein Codeabschnitt vom Original abweichen kann:

- Type-1: Der zweite Codeabschnitt bleibt gleich, nur Leerzeichen oder Kommentare werden hinzugefügt.
- Type-2: Auf einer syntaktischen Ebene wird der Code nicht geändert, bis auf folgende Elemente: Identifiers, Literals, Typen, Leerzeichen, Code-Layout und Kommentare.
- Type-3: Da wird der Code modifiziert, es verstehen sich folgende Änderungen: gelöschte oder hinzugefügte Statements, umbenannte Identifiers, Literals, Typen, Leerzeichen, Code-Layout und Kommentare.
- Type-4: Zwei Codefragemente, die dieselbe Aufgabe erledigen, welche jedoch auf zweierlei Art und Weise implementiert wurden.

Für jeden Änderungstypen hat sich diese Forschungsgruppe verschiedene Szenarien ausgedacht und hat verglichen, welche Ergebnisse da jede Methode liefern kann. Jede Methode wird von zwischen vier und elf Implementierungen vertreten, je nach Methode. Das Szenario 1 entspricht dem Typ-1 und so weiter. Je nach Implementierung sind mit unterschiedlichen Ergebnissen zu rechnen, jedoch kann generell gesagt werden, dass:

- bei dem Szenario 1 grundsätzlich alle Methoden befriedigende Ergebnisse liefern. Jedoch heben sich Text-basierende, Token-basierende und Tree-basierende Methoden hervor.
- bei dem Szenario 2 alle Methoden außer der Text-basierenden Methode punkten können.
- bei dem Szenario 3 alle Methoden punkten können, je nach Implementierung. Manche Programme wie Duplix, Gemini oder Deckard können mäßig befridiegende Erebnisse berechnen. Generell hebt sich jedoch keine Methode hervor.
- Bei dem Szenario 4 nur 2 Methoden überhaupt Ergebnisse zurückliefern können, die Metrics-basierende und die Graph-basierende Methode. Die drei anderen Methoden können keinen Ähnlichkeitsgrad berechnen, bis auf ein einziges Programm, Marcus, das eine Text-basierende Methode implementiert.

Zurück zu unserem Fall. Bei Android-Applikation ist es so, dass der Java-Code, der später verglichen wird, vom Obfuskator geändert wurde. Dies wurde in Abschnitt 2.3 ausführlich beschrieben. Diese Situation entspricht also eher dem Type-3. Kommentare spielen da keine Rolle, aber es werden sicher einige Methoden hinzugefügt beziehungsweise gelöscht. Außerdem werden alle Variablennamen während der Obfuskation geändert.

Code-Cloning-Detection wird also eingesetzt, wenn man den Ähnlichkeitsgrad zwischen zwei Codeabschnitten herausfinden möchte. In dieser Diplomarbeit kann jedoch Code-Cloning allein keine Lösung sein, es wird aber ohne Zweifel ein Teil der Lösung sein. Nehmen wir an, die Applikation besteht aus 200 Files (eher im Fall einer kleinen Demo-App) und die Anzahl an aus Bibliotheken extrahierten Files beträgt 1000 (drei normale Android-Bibliotheken). Da würden wir bis zu 200 000 Code-Cloning-Vergleiche ausführen müssen. Nehmen wir dann an, dass eine solche Operation eine halbe Sekunde dauert, dann würde der ganze Prozess einen ganzen Tag dauern, was einfach zu lang ist. Würden wir dann normale Applikationen und mehr beziehungsweise größere Bibliotheken analysieren, dann würde die Ausführungszeit drastisch zunehmen.
Diese kleine Rechnung wirft folgende Frage auf: Wie kann eine Vorauswahl der zu analysierenden Files getroffen werden, sodass nicht relevante Files frühzeitig einfach ausgeschlossen werden können. Diese Frage lässt sich mit einer Art Filtering beantworten, das nach dem Extrahieren und vor der Code-Cloning-Detection stattfindet. Die Idee ist, vor dem Code-Cloning sagen zu können, mit welchen Files in der Datenbank dieses File aus der Applikation matchen könnte, damit unnötige Operationen erspart werden: Dieses Filtering sollte im Idealfall folgende Kriterien erfüllen:

- Flexibilität: An dieser Stelle sei es nochmal hervorgehoben, dass der zu analysierende Code dekompiliert wurde. Es heißt also, dass das Filtering keine Prüfsumme oder Ähnliches sein kann.
- Ausführungszeit: Ziel dieser Filtering-Methode ist es ja, den ganzen Prozess zu beschleunigen. Das Filtering muss also schneller als die Code-Cloning-Detection sein.
- Genauigkeit: Es müssen nur irrelevante Files ausgeschlossen werden. Der Filteringprozess muss flexibel und genau genug sein, um die Randeffekte des Obfuscation-Verfahrens umgehen zu können.
- Berechenbarkeit: Die zu erkennenden Bibliotheken werden uns in Form von Java-Code zur Verfügung gestellt. Am besten kann man das Filtering direkt auf dem Java-Code aufbauen.

Um all diese Kriterien zu erfüllen, wird eine Art Prüfsumme entwickelt, welche auf den verschiedenen Merkmalen vom Java-Code basiert. Es wird jedoch auf einer logischen Ebene gearbeitet.

3.2 Definition einer Filtering-Methode

In einem von Sun Microsystems publierten White Paper ([11]) wurde die Programmiersprache Java und deren Ziele beschrieben. Im ersten Teil des Dokuments werden sie alle im Detail beschrieben. Hauptsächlich strebt Java folgende Ziele an:

- einfach, objektorientiert und verteilt
- robust und sicher
- architekturneutral und portabel
- leistungsfähig
- interpretierbar, parallelisierbar und dynamisch

Da eine Android Applikation also als ein reines Java Projekt gesehen werden kann, wird unsere Prüfsumme eine Kombination aus verschiedenen Java-Merkmalen sein, welche schnell und auf zuverlässige Weise berechnet werden können. Man kann sich fragen, was hier mit Java-Merkmal gemeint ist. In diesem Zusammenhang ist ein Merkmal ein Wert, der sich für jeden Java-Codeabschnitt berechnen lässt. Folgende Merkmale fallen einem diesbezüglich als Erstes ein:

- Wie viele Klassen/*Interfaces* definiert sind
- Wie viele Methoden werden dokumentiert beziehungsweise wie viele Zeile Kommentare sind
- Wie viele Methoden aufgerufen werden
- Wie viele Methoden definiert werden
- Wie viele Ausnahmen behandelt werden
- Wie viele Klassen-Instanz erzeugt werden (dynamische Allozierung)
- Wie viele Schleifen benutzt werden

Es können weitere Merkmale aufgelistet werden, aber die schon oben erwähnten Kriterien können einen Codeabschnitt bereits präzis definieren. Damit wir uns auf die oben erwähnten Merkmale einigen können, folgt ein Beispiel. Hier ist eine ganz triviale Klasse zu sehen:

```
1  package tld.domain.project
2
3  import tld.domain.project.core.Engine;
4  import java.lang.String;
5
6  public class Vehicle {
7
8          protected Engine engine;
9          protected String name;
10
```

```
11          public Vehicle(Engine engine, String name) {
12
13                  this.engine = engine;
14                  this.name = name;
15          }
16
17          public String getName() {
18
19                  return this.name;
20          }
21
22          public boolean isEngineStarted() {
23
24                  return this.engine.isStarted();
25          }
26
27          public boolean startEngine() {
28
29                  if (!isEngineStarted()) {
30
31                          this.engine.start();
32                  }
33          }
34  }
```

Nach kurzer Beobachtung kann man feststellen, dass in diesem Abschnitt nur eine Klasse namens Vehicle definiert wird. Die Klasse hat einen einzigen Konstruktor und verfügt über drei Methoden, getName(), isEngineStarted() und startEngine(). In der Methode startEngine() gibt es eine If-Anweisung. Ist isEngineStart() gleich true, dann wird die Methode start() aufgerufen. In diesem Codeabschnitt wurde keine Schleife definiert und es befindet sich kein einziger Kommentar.
Dann ergibt sich folgende Tabelle:

Tabelle 3.1: Statistiken aus einem Java-File

Klassen	Methoden-Definition	Methoden-Aufrufe	if-Anweisungen
1	4	3	1

Es können viele andere Kriterien beobachtet werden, aber in dieser Arbeit wird ausschließlich mit folgenden Merkmalen gearbeitet:

- Wie viele Klassen/*Interfaces* definiert werden
- Wie viele Methoden im ganzen File aufgerufen werden

- Wie viele Methoden definiert werden
- Wie viele if-Anweisungen zu finden sind
- Wie viele Ausnahmen (*Exceptions*) behandelt werden

Hier darf man nicht vergessen, dass jedes File, das aus der Applikation extrahiert und dann analysiert wird, von Proguard (oder Ähnlichem) obfuscated wurde, von javac-Programm kompiliert wurde und anschließend von einem Dekompiler dekompiliert wurde. Das heißt also, dass sich manche Merkmale an dieser Stelle für unsere Arbeit nicht eignen. Auch wenn sich Kommentare in den originalen Files befinden, werden sie nach dem Kompilierungsprozess nicht mehr zu finden sein. Die Anzahl an definierten und aufgerufenen Methoden sind gute Merkmale. Auch wenn der Obfuskator wahrscheinlich alle Methoden umbenannt und ein paar hinzugefügt hat, bleibt dieser Wert trotzdem relativ stabil.
Jetzt sind unsere zur Analyse ausgewählte Kriterien definiert. Als Erstes wird zum Beispiel das File HttpResponse.java aus einer zu erkennenden Bibliothek analysiert. Für dieses File wurden vom Parser folgende Merkmale festgestellt:

Tabelle 3.2: Analyse des AIODJ.java-Files

Klassen-Anzahl	Methoden-Definition	Methoden-Aufrufe	if-Anweisungen	Ausnahmen
1	3	6	1	2

Mit diesen Merkmalen werden wahrscheinlich unterschiedliche Codeabschnitte als doch ähnlich gesehen. Aber an dieser Stelle geht es nur darum, vom Inhalt her unpassende Files aus der *Code-Cloning-Detection* auszuschließen.

3.3 Komplettes Erkennungsverfahren

Nun wurde die Filtering-Methode definiert und dazu kommt eine Code-Cloning-Stufe. Allerdings ist dies nur ein kleiner Teil eines größeren Programms, welches eine fertige Applikation als Input nimmt und eine Report als Output generiert.
Generell zu der gesamten Struktur: Das erarbeitete Konzept baut auf zwei Programmen und einer SQL-Datenbank auf. Das erste Programm beschäftigt sich nur mit den zu findenden Bibliotheken, das andere nur mit der auszuwertenden Android-Applikation. Beide Programme kommunizieren mit einer SQL-Datenbank, welche die Java-Merkmale speichert. Der generelle Ablauf befindet sich unterstehend abgebildet:

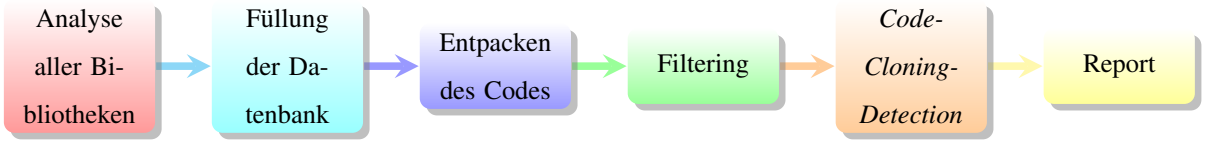


Abbildung 3.1: Abbildung des gesamten Erkennungsverfahrens

Ein einmaliges Verfahren bildet die Analyse aller Bibliotheken, die später dann erkannt werden sollen. Es wird der Quellcode jeder Bibliothek heruntergeladen beziehungsweise auf dem Filesystem gelesen, und von einem Parser analysiert. Hier wird auf jedes Java-File ein an unsere Situation angepasster Java-Parser ausgeführt. Dieser Java-Parser wird für jedes File nach den oben erröteten Merkmalen suchen. Welche Kriterien tatsächlich ausgewählt wurden, wird später ausführlich beschrieben und begründet. Die vom Parser gesammelten Daten werden dann in einer SQL-Datenbank gespeichert. Somit sind wir nun am Ende des ersten Schrittes. Die Analyse aller zu findenden Bibliotheken muss nur einmal stattfinden, weil die Ergebnisse gespeichert und vom zweiten Programm wiederverwendet werden. Dieses Programm wurde LibParser genannt.
Nach der Erstellung der Datenbank und deren Füllung kann die tatsächliche Analyse der Android-Applikation beginnen. Dafür wurde ein Programm namens ApkParser entwickelt. ApkParser folgt im Wesentlichen zwei Zielen an, zum einen muss man aus einer fertigen Applikation auf deren Merkmale kommen, zum anderen muss das Programm die berechneten Merkmale mit denen in der Datenbank vergleichen. Zuerst wird also der Code aus dem APK-File extrahiert, dann wird dieser von einem Parser analysiert. Beide Parser, der für die Bibliotheken und der für den aus dem APK-extrahierten Code basieren auf einem gemeinsamen Kern. Die daraus resultierenden Daten werden mit den Daten in der Datenbank verglichen. Nach dieser Etappe befinden sich die Zwischenergebnisse in einer Art Tabelle, einer Matching-Tabelle, deren Eigenheiten im Folgenden erläutert werden. Diese Zwischenergebnisse müssen dann raffiniert werden. Danach kommt das Code-Cloning-Detection und weitere Filteringmethoden. Nun ist das Analyseverfahren am Ende und das Report kann generiert werden.

3.4 Analyse der Bibliotheken

Die Analyse aller zu findenden Bibliotheken bildet die erste Stufe des gesamten Konzepts.
Android-Bibliotheken sind technisch betrachtet ganz normale Java-Bibliotheken und werden von den jeweiligen Entwicklern sehr häufig in Form von ZIP- beziehungsweise JAR- Dateien zur Verfügung gestellt. Diese Archiv-Files bilden den Startpunkt des Analyseverfahrens.
Bevor der Analyseprozess überhaupt gestartet werden kann, muss eine XML-Datei geschrieben werden,

welche alle zu analysierenden Bibliotheken beschreibt. Im Prinzip müssen für jede Bibliothek folgende Informationen bekanntgegeben werden: Absoluter Pfad zum ZIP-Archiv, Packagenamen und Version. Dann wird dieses XML-File gelesen und jede Bibliothek wird analysiert. Um an den Quellcode zu gelangen, werden diese Archiv-Dateien entpackt und deren Inhalt wird aufgelistet und gefiltert. In solchen Archiv-Files befinden sich manchmal für uns nicht relevante Dateien, etwa Files wie: README, LICENCE, HTML-Dokumentation oder auch Beispiele. Bei dem Entpacken werden diese Files einfach ignoriert, da sie keine Information mitbringen. Alle Java-Files liegen nun auf dem Filesystem.
Das Ziel dieser ersten Stufe ist es, die Merkmale aller aus den Bibliotheken kommenden Files in der SQL-Datenbank liegen zu haben. Zu diesem Zweck werden alle extrahierte Files von einem dafür geschriebenen Parser, LibParser, durchsucht und analysiert. In Kapitel 3.2 wurden alle relevante Kriterien aufgelistet und erklärt. Genau diese Merkmale wird der Parser für jedes Java-File berechnen. Die Ergebnisse werden dann in der SQL-Datenbank mit den aus der XML-Datei gelesenen Informationen abgespeichert.

3.5 Füllung der Datenbank

Die Füllung der Datenbank erfolgt nur einmal, mit der Annahme, dass sich das Set von zu erkennenden Bibliotheken nicht ändert, egal wie viele Applikationen analysiert werden müssen. Sollen jedoch zusätzliche Bibliotheken erkannt werden, dann muss die Datenbank entsprechend ergänzt werden.
Das Schema der SQL-Datenbank spiegelt die ausgewählten Java-Merkmale wider. Es wird nur eine Tabelle verwendet, die alle Merkmale enthält.

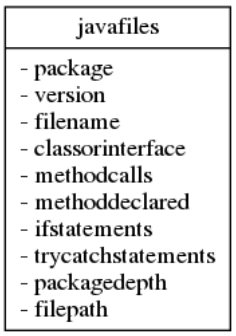


Abbildung 3.2: Schema der SQL-Datenbank

Danach kann nach Files gesucht werden, die ähnliche Merkmale aufweisen. Nehmen wir an, dass wir nach einem File suchen, in dem fünf Methoden aufgerufen werden. Ein solcher Befehl sieht in Pseudocode folgendermaßen aus:

```
1 SELECT package, version, filename WHERE methodcalls == 5
```

Jedoch sei hier hervorgehoben, dass der aus dem APK extrahierte Code kompiliert und obfuscated wurde. Weswegen muss hier ein Deltawert eingeführt werden, mit dem sich die Nebenwirkungen des Obfusaction-Prozesses umgehen lassen. Anstatt dessen, nach einem genauen Wert zu suchen, wird eher innerhalb eines Bereiches gesucht. Nennen wir die gesucht Anzahl an aufgerufenen Methoden a und den Deltawert Δ. Dann muss zwischen zwei Situationen unterschieden werden:

- $a > \Delta$, dann wird zwischen $a - \Delta$ und $a + \Delta$ gesucht
- $a \leqslant \Delta$, dann wird zwischen 0 und $a + \Delta$ gesucht

In diesem Fall würden wir nach einem File suchen, in dem fünf Methoden aufgerufen werden, mit einem Deltawert von 2. Der Pseudocode dieses Befehls lautet in diesem Fall:

```
1 SELECT package, version, filename WHERE 3 <= methodcalls <= 7
```

Die Auswirkung dieses Deltawertes auf die Genauigkeit und die Geschwindigkeit des Erkennungsverfahrens wird dann in der Testphase im Detail beleuchtet.

3.6 Analyse der Applikation

Nun befinden sich alle Merkmale der Bibliotheken samt deren Informationen in der SQL-Datenbank. Diese Merkmale lassen sich auch leicht abfragen. Es kann nun die Analyse der fertigen Applikation beginnen. Die Applikation wird in Form eines APK-Files zur Verfügung gestellt. Nun wissen wir, dass ein APK-File einem ZIP-File gleichzustellen ist. Dieses wird dann entpackt und das classes.dex-File wird dem Dekompiler übergeben. Der Dekompiler kann aus diesem classes.dex-File den Java-Code extrahieren, welcher Code die Basis der Analyse darstellt.
Nachdem sich der Java-Code im Dateisystem befindet, kann jetzt mit dem tatsächlichen Analyseverfahren begonnen werden. Für jede Java-Datei werden verschiedene Java-Merkmale von einem Parser berechnet. Alle beobachteten Merkmale wurden in Kapitel Abschnitt 3.2 geschildert.
Nach jeder Analyse eines Java-Files wird in der SQL-Datenbank nach Dateien gesucht, die dieselben Merkmale aufweisen. An dieser Stelle sei nochmal erwähnt, dass alle Java-Files, die aus der APK-Datei extrahiert wurden, einen langen Weg hinter sich haben. Der originale Code wurde zumal von Proguard (oder Ähnlichem) obfuscated, von javac-Programm kompiliert und anschließend aus dem APK-File extrahiert und dann dekompiliert. Die Merkmale, die gerade eben berechnet wurden, werden sich also

definitiv nicht eins zu eins in der Datenbank befinden, weil durch diese verschiedenen Prozesse Informationen verloren gingen beziehungsweise Noise hinzugefügt wurde. Hier wird also nach Files gesucht, die ungefähr dieselben Merkmale haben. Dies wird mittels SQL-Kommandos realisiert. Mehr dazu im nächsten Kapitel, in welchem die gesamte Implementierung detailliert und die technischen Entscheidungen begründet werden.

3.6.1 Berechnung der Matching-Tabelle

Wenn alle zu analysierenden Files geparsed wurden und es danach in der Datenbank gesucht wurde, kommt eine Matching-Tabelle zustande, die Folgendermaßen abgebildet werden kann:

Tabelle 3.3: Abbildung einer fiktiven Matching-Tabelle

File im APK	Packagename	Version	File
ABC.java	com.google.gson	2.6	Gson.java
ABC.java	org.jmrtd	4.5	PassportService.java
XYZ.java	com.google.gson	2.5	Json.java
XYZ.java	es.dmoral.toasty	1.0	ToastyUtils.java
XYZ.java	net.sf.scuba	1.5	DG1.java

Diese Tabelle besagt im Prinzip, welches File aus der Applikation jenem File aus allen analysierten Bibliotheken am ähnlichsten ist. Um eine bessere Lesbarkeit zu gewährleisten, wurden hier nur zwei Files aus einer fiktiven analysierten Applikation abgebildet. Das erste File ist ABC.java. Nach dessen Analyse und mehreren Vergleichen mit der Datenbank ergibt sich, dass es sich um zwei bekannte Files handeln könnte: entweder um Gson.java aus der Bibliothek com.google.gson in Version 2.6 oder um PassportServices.java aus der Bibliothek org.jmrtd in Version 4.5. Die zweite Datei ist XYZ.java, welche eines der drei folgenden Files sein konnte: Json.java aus der Bibliothek com.google.gson in Version 2.5, ToastyUtils.java aus der Bibliothek es.dmoral.toasty in Version 1.0 oder DG1.java aus der Bibliothek net.sf.scuba in der Version 1.5.

Hier sei noch einmal hervorgehoben, dass diese Tabelle ein großer Hinweis darauf ist, auf welchen Bibliotheken die Applikation aufbaut. Jedoch kann diese Tabelle alleine kein Erkennungsverfahren darstellen, weil zwei sehr unterschiedliche Files vielleicht in manchen Fällen ähnliche Merkmale aufweisen. Aus diesem Grund folgt eben *Code-Cloning-Detection*, damit diese Zwischenergebnisse bestätigt beziehungsweise abgelehnt werden können.

3.6.2 Optimierung der Ergebnisse

Je mehr Bibliotheken erkannt werden sollen und je größer die zu analysierende Applikation ist, umso länger wird die oben abgebildete Tabelle. Eine längere Matching-Tabelle bedeutet im Endeffekt mehr *Code-Cloning-Detection* und eine längere Ausführungszeit. Daher die Notwendigkeit, um Rechenleistung zu sparen und um die Laufzeit relativ kurz zu halten, diese Tabelle zu verkürzen, in dem bestimmte Elemente gelöscht werden. Es müssen also verschiedene Optimierungsstrategien entwickelt und erarbeitet werden, welche es ermöglichen, die Tabelle kurz zu halten, wodurch das gesamte Erkennungsverfahren beschleunigt wird. Diese Optimierungsstrategien verstehen sich hier unabhängig von der zur Implementierung gewählten Programmiersprache und auf einer theoretischen Ebene.

Ähnliche Versionen

In der Datenbank, die alle Merkmale der Bibliotheken enthält, werden viele der Bibliotheken in verschiedenen Versionen abgebildet. Bei größeren Bibliotheken ist es oft so, dass sie aus verschiedenen Modulen bestehen. Module, die etwas ältere Technologien oder Protokolle unterstützen, werden kaum aktualisiert, weil sie sehr stabil und schon über alle benötigte Funktionen verfügen. Weil manche Files zwischen zwei aufeinanderfolgenden Versionen nicht geändert wurden, taucht die unterstehend abgebildete Situation häufig auf:

Tabelle 3.4: Typisches Double-Match Beispiel

File im APK	Packagename	Version	File
ABC.java	com.package.my	2.6	Circle.java
ABC.java	com.package.my	2.5	Circle.java

Nehmen wir an, jetzt wird das aus dem APK-File extrahierte ABC.java-File mit Files in der Datenbank verglichen. Wenn dieses Files mit dem File Circle.java aus dem Package com.package.my in Version 2.6 ähnlich ist, dann ist er ebenfalls dem File Circle.java aus demselben Package in der Version 2.7 ähnlich, weil das File Circle.java wahrscheinlich zwischen diesen zwei Revisionen nicht geändert wurde. Da wäre es eine reine Zeitverschwendung, beide Files vom Code-Cloning-Tool analysieren zu lassen. Obwohl es sich theoretisch um zwei verschiedene Files handelt, weil aus zwei unterschiedlichen Versionen kommend, haben beide denselben oder einen sehr ähnlichen Inhalt. Aus diesem Grund kann Zeit erspart werden, in dem nur eines der beiden Files analysiert wird.
In dieser Stufe werden also alle Files, die sich mehrfach in der Tabelle befinden, durch ein einziges File

ersetzt. Dies stört keinesfalls das Analyseverfahren, weil diese Files nach unseren Kriterien genau dieselben sind. Später findet eine weitere Stufe des Erkennungsverfahrens heraus (*Code-Cloning-Detection*), um welche der beiden Versionen es sich wirklich handelt.

Packagedepth

Es wurde schon erklärt, wie der Obfuskator die Packagenamen umschreibt. Jedoch bleibt die Verzeichnisstruktur gleich. Dies wird wollen wir zu Nutzen ziehen, in dem die Tiefe eines Files innerhalb der Verzeichnisstruktur berechnet und gespeichert wird. Mit diesem neuen Merkmal können schnell und auf zuverlässiger Weise Files aus dem Erkennungsverfahrens ausgeschlossen. Dieser Wert wird in den weiteren Teilen der Arbeit als ein ganz normales Merkmal gesehen.

3.6.3 Code-Cloning-Detection

Integration in das Projekt

Diese Matching-Tabelle weist zwar darauf hin, von welchen Bibliotheken die Applikation abhängt, kann jedoch nicht per se als Ergebnis betrachtet werden. Der Dekompilierungprozess kann manchmal nicht erfolgreich abgeschlossen werden, weil der Dekompiler mit dem von javac generierten Code nicht zurecht kommt. Es kann auch passieren, dass sich die von der Applikation angewendete Bibliothek gar nicht in der Datenbank befindet. Es muss also eine zweite Stufe folgen, in der diese Hinweise bestätigt oder abgelehnt werden können. In dieser Stufe werden extrahierte Files aus der Applikation mit originellen Files aus den jeweiligen Bibliotheken verglichen. Es ist im Prinzip so, dass jede in der Matching-Tabelle vemutete Beziehung nun anhand eines externen Tools geprüft wird.
Wenn die Applikation nicht obfuscated wurde, ist es natürlich sehr einfach, festzustellen, ob es sich um dasselbe File handelt, weil in den Files dann nichts geändert wurde. Jedoch werden immer mehr und mehr Applikationen obfuscated, weil es mittlerweile sehr einfach geworden ist, es zu tun. Es muss also definiert werden, was hier mit *Code-Cloning-Detection* gemeint ist.

Code-Cloning-Detection Beispiel

Nehmen wir an, MathHelper.java ist unser Reference-File:

```
1 package test;
2
3 public class MathHelper {
4
5     public static int factorial(int n) {
6         if(n == 0) {
```

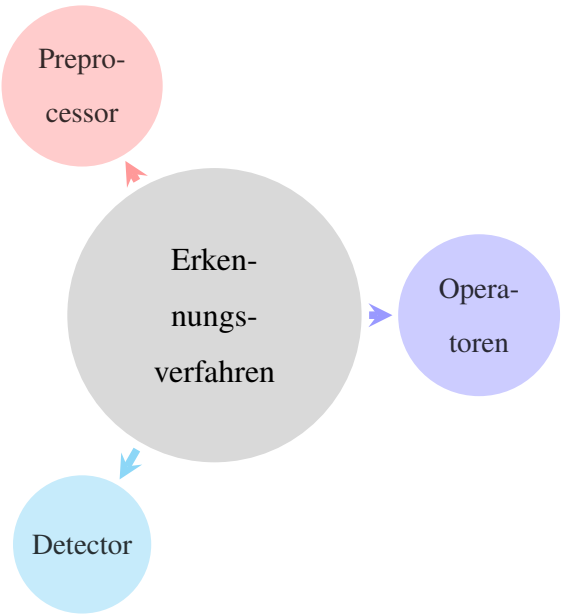
```
7               return 1;
8           } else{
9               return n * factorial(n-1);
10          }
11      }
12
13      public static int gcdOne(int a, int b) {
14        while (b != 0) {
15            if (a > b) {
16                a = a - b;
17            } else {
18                b = b - a;
19            }
20        }
21        return a;
22      }
23  }
```

Das von der Matching-Tabelle vorgeschlagene File hat dann folgenden Inhalt:

```
1   package test;
2
3   public class TestFileTwo {
4
5       public int factorial(int n) {
6           if(n == 0) {
7               return 1;
8           } else {
9               return n * factorial(n-1);
10          }
11      }
12
13      public int gcdTwo(int c, int d) {
14          while (d != 0) {
15              if (c > d) {
16                  c = c - d;
17              } else {
18                  d = d - c;
19              }
20          }
21          return c;
22      }
23  }
```

Werfen wir nun einen Blick auf die zwei eingeblendeten Files. Nach kurzer Beobachtung fällt es einem leicht, zu erkennen, dass diese kaum voneinander abweichen. Es wurden nur die Methoden beziehungsweise die Variablen umbenannt. Die erste Funktion wurde eins-zu-eins übernommen, wobei bei der zweiten Funktion einige Änderungen vorgenommen wurden.

Es existieren schon mehrere Strategien, um den Match zwischen zwei Files zu berechnen (wie in der Einleitung dieses Kapitels erwähnt). Je nach Fall und Programmiersprache ist die eine oder andere Vorgehensweise am interessantesten. In dieser Arbeit wird das Erkennungsverfahren auf dem AST aufbauen. Dieser Baum bildet die logische Struktur des Codeabschnittes beziehungsweise der Klasse ab. Wenn zwei Klassen verglichen werden, werden im Endeffekt nur beide Bäume verglichen [19].



Das gesamte Erkennungsverfahren kann, wie obenstehend, als eine Pipeline abgebildet werden.

- Preprocessor: Ein Preprocessor ändert den Abstraktbaum, löscht oder fügt Elemente hinzu.
- Detector: Ein Detector implementiert eine der existierenden Methode.
- Operator: Eine Operation, die vom Processor ausgeführt wird.

Operatoren führen dazu, dass bestimmte Elemente der Bäume im Suchprozess ignoriert werden. Es sind schon viele Operatoren implementiert, aber benutzerdefinierte Operatoren lassen sich ebenfalls auf einfache Art und Weise in das *Pipeline* einbinden. An dieser Stelle möchte ich nur einige Operatoren erwähnen und deren Funktionsweise mit Beispielen in Pseudocode erläutern. Die Liste wird jedoch kurz behalten, damit sich der Leser einen Überblick verschaffen kann (http://jccd.sourceforge.net/operators.html):

- GeneralizeMethodArgumentTypes: Types von Methoden-Argumenten werden weggenommen. Dann wird diese Funktion computeFactorial(int x) durch computeFactorial(x) ersetzt.
- GeneralizeMethodDeclarationNames: Methoden-Namen werden im Vergleichsverfahren nicht mehr in Betracht gezogen. Dieser zählt im Android Usecase zu einem der wichtigsten Operatoren, weil

davon ausgegangen werden muss, dass sehr viele Methoden (wenn nicht alle) vom Obfuskator umbenannt werden.

- GeneralizeMethodReturnTypes: Der Rückgabewert-Type wird gelöscht, int getMemberVariable() wird zu getMemberVariable().
- GeneralizeVariableNames: Alle Variablennamen werden gelöscht, int factorial(int x) wird int factorial(int).
- RemoveGenericTypes: HashMap<String, MeineKlasse> wird HashMap. Jegliche Typ-Informationen werden nicht mehr betrachtet.

Alle weitere Operatoren, mit denen das Erkennungsverfahren optimiert werden kann, werden im nächsten Kapitel vorgestellt.
Dieses Tool liefert dann eine Datenstruktur zurück, welche all die ähnlichen Bereiche enthält. Deren Implementierung wird später beschrieben. Aus dieser Datenstruktur wird ein Wert berechnet, welcher dann besagt, wie ähnlich die zwei Files tatsächlich sind. Je höher ist der Wert, desto ähnlicher sind beide Files. Der Wert ist eine natürliche Zahl, die zwischen 0 und der Anzahl an Code-Zeilen liegt. Dieser Wert sagt nämlich genau, wie viele Code-Zeilen das Tool für ähnlich sieht. In der Matching-Tabelle stehen für jedes analysierte File mehrere mögliche Treffer. Jeder Treffer wird mit dem aus der APK kommenden File verglichen und der Ähnlichkeitsfaktor wird zwischengespeichert.
Es werden also alle Files, die sich in der Matching-Tabelle befinden, verglichen. Für jedes aus dem APK-Archiv extrahierte File wird dann das beste Match zwischengespeichert. Die Idee ist dann, zu zählen, wie oft jede Bibliothek vorkommt und in welcher Version.
Kommen wir nun zu unserem Beispiel zurück. Früher wurde gemerkt, dass beide Files ähnlich erschienen. Um das nun von diesem Tool prüfen zu lassen, brauchen wir nun mal einige Operatoren, um das Pipeline zu konfigurieren.
Für unser oben eingeblendetes Beispiel werden folgende Operatoren benötigt:

- GeneralizeMethodDeclarationNames: der Name der zweiten Funktion wurde geändert.
- GeneralizeVariableNames: die Variablen wurden ebenfalls umbenannt.

Dann kann das Programm ausgeführt und der Wert berechnet werden: 17. Das Tool hat zwei ähnliche Bereiche gefunden. Der erste Bereich ist die erste Funktion (Beginn des Blockes Zeile X und Ende Zeile Y) und der zweite Bereich (Beginn des Blockes Zeile X und Ende Zeile Y) ist die zweite Methode.

Für jedes analysierte File wird eine solche Tabelle erzeugt, welche schildert, wie groß mit welchem File der Faktor ist. Am Ende steht also klar, mit welchem File aus welcher Bibliothek das analysierte File am Nähesten ist. Es wird hier nur der höchste Werte beibehalten und das dazupassende File:

Tabelle 3.5: Berechnete Merkmale für das File ABC.java

File aus der App	File aus der Bibliothek	Faktor
ABC.java	org.package.mylib:Auto.java	10
ABC.java	org.package2.mylib2:LaserBeam.java	20
ABC.java	org.package3.mylib3:Banana.java	3

Diese Tabelle sagt also, dass für das File ABC.java (aus dem APK extrahiert) drei verschiedene Files passen würden. Hier sprechen sich aber die Faktoren dazu aus, dass ABC.java eigentlich LaserBeam.java ist. Dieser Faktor, wir vorher schon erklärt, ist eigentlich die Anzahl an gemeinsamen Zeilen (die zumindest dem Code-Cloning-Verfahren nach als solche erkannt werden).

3.7 Auswertung der Ergebnisse

Nun haben die zwei Stufen stattgefunden, die Berechnung der Merkmale samt Suche in der SQL-Datenbank und das Code-Cloning-Verfahren.
In der letzten Phase (*Code-Cloning-Detection*) wird folgende Tabelle generiert:

Tabelle 3.6: Ergebnisse von Code-Cloning-Detection

File aus der App	Package	File aus der Bibliothek
ABC.java	org.package.mylib	Auto.java
DQOK.java	org.package2.mylib2	LaserBeam.java
DCNBS.java	org.package3.mylib3	Banana.java

Dazu wird für jedes analysierte File das passende File mit dem höchsten Ähnlichkeitsgrad genommen. Es kann dann gezählt werden, wie oft jede gefundene Bibliothek auftaucht. Diese Zahlen werden dann mit dem in letzten Phase berechneten Faktor kombiniert. Je öfter eine Bibliothek gefunden wurde und je höher der Faktor ist, desto wahrscheinlicher ist es, dass sich diese Bibliothek tatsächlich in der ausgewerteten Applikation befindet.

An dieser Stelle wissen wir für jedes aus dem APK extrahierte und analysierte File, aus welcher Bibliothek es kommen sollte. Da ist natürlich mit Fehlern und *False-Positives* zu rechnen, weil der Obfuskation-Prozess unterschiedlich konfiguriert sein kann. Außerdem kann der Dekompiler nicht jedes File perfekt dekompilieren und es tauchen manchmal Fehler auf. Hier müssen statistische Verfahren angewendet werden, um die Aussagekräftigkeit unseres Erkennungsverfahrens zu messen.
An dieser Stelle ist nun das Analyseverfahren abgeschlossen und die gefundenen Bibliotheken samt deren Versionen können aufgelistet werden.

3.8 Erstellung des Reports

Nach dem vorherigen Schritt ist das Analyseverfahren abgeschlossen. Es bleibt nur die Erstellung eines Reports übrig, auf welchem dann die unterschiedlichen Abhängigkeiten zu sehen sind.
Neben jeder vermutlich angewendeten Bibliothek stehen folgende Informationen:

- Wie viele Files analysiert wurden.
- Wie viele Files, und mit welchen Dateien,erkannt wurden.
- Für jedes erkannte Files den Ähnlichkeitsgrad.

Das Report wird in Form von einer .txt-Datei beziehungsweise .pdf, wenn eine LaTeX-Umgebung vorhanden ist.

4 Implementierung

In diesem Teil wird die Umsetzung des vorher erwähnten Konzepts detailliert beschrieben. Es werden sowohl technische als auch Architekturentscheidungen begründet.

4.1 Technische Umgebung

Die technische Umsetzung des in Kapitel 3 beschriebenen Konzepts baut auf folgenden Technologien und Programmiersprachen auf:

- Java SE [1]
- MariaDB [2]
- javaparser [3]
- jccd [4]
- jadx [5]

Generell zu der Implementierung sei gesagt, dass die beiden Parser und die SQL-Infrastruktur in Java geschrieben sind. Mit dieser Programmiersprache entwickelte Applikationen können mühelos auf den gängigsten Betriebssystemen ausgeführt werden. Meinerseits waren auch Java-Kenntnisse vorhanden, erworben durch verschiedene Lehrveranstaltungen an meiner Universität in Frankreich und an der Fachhochschule St. Pölten. Diese Programmiersprache ermöglicht es, ein plattformunabhängiges Programm zu entwickeln, welches sowohl auf kleinen Rechnern als auch auf Großrechnern ausgeführt werden kann. Außerdem ist das Programm dann einfacher zu veröffentlichen, weil der Endbenutzer nur die ausführbare Jar-Datei braucht, welche sowohl das Programm als auch alle Abhängigkeiten beinhaltet. Ebenfalls

[1] https://www.oracle.com/java/index.html
[2] https://mariadb.org/
[3] http://javaparser.org/
[4] http://jccd.sourceforge.net/
[5] https://skylot.github.io/jadx/

ist die Bibliothek javaparser in Java geschrieben. Auf dieser Bibliothek bauen beide Parser, die alle Java-Files analysieren werden. Diese Bibliothek lässt sich einfach in bestehende Programme einbinden und ermöglicht ein schnelles und an unsere Bedürfnisse angepasstes Parsen.
In einer SQL-Datenbank werden alle vom Parser berechneten Merkmale gespeichert. Die aktuelle Implementierung basiert auf MariaDB, welche aufgrund rechtlicher Probleme mit Oracle als Ersatz für MySQL entwickelt wurde. Es könnten jedoch auch andere Varianten eingesetzt werden, da keine MariaDB-spezifischen Befehle ausgeführt werden. Es sollte also in Ordnung sein, eine MySQL- oder PostgreSQL-Instanz anzuwenden. Nach Bedarf ließe sich auch eine nicht auf SQL-basierende Datenbank einbinden, es müssten dann nur die Klasse umgeschrieben werden, die tatsächlich mit dieser kommunizieren.
Der Kompilierungsprozess, im dem aus dem Code ein fertiges APK-File wird, wurde in Abschnitt 2.4 beschrieben. Genau das Gegenteil macht ein Decompiler. Hier kommt jadx zum Einsatz. Jadx ist, laut der Projekt-Homepage, *a command line and GUI tools for produce Java source code from Android Dex and Apk files*. Es existieren leider nur wenige open-source Android-spezifische Decompilers. Weil dieser noch entwickelt und betreut wird, kann davon ausgegangen werden, dass diese auch neue Android-Instruktionen unterstützen. Das ist zum jetzigen Zeitpunkt eine zukunftstaugliche Wahl.

4.2 Code-Architektur

Untenstehend befindet sich das Klassendiagramm der gesamten Applikation, also von LibParser und ApkParser:

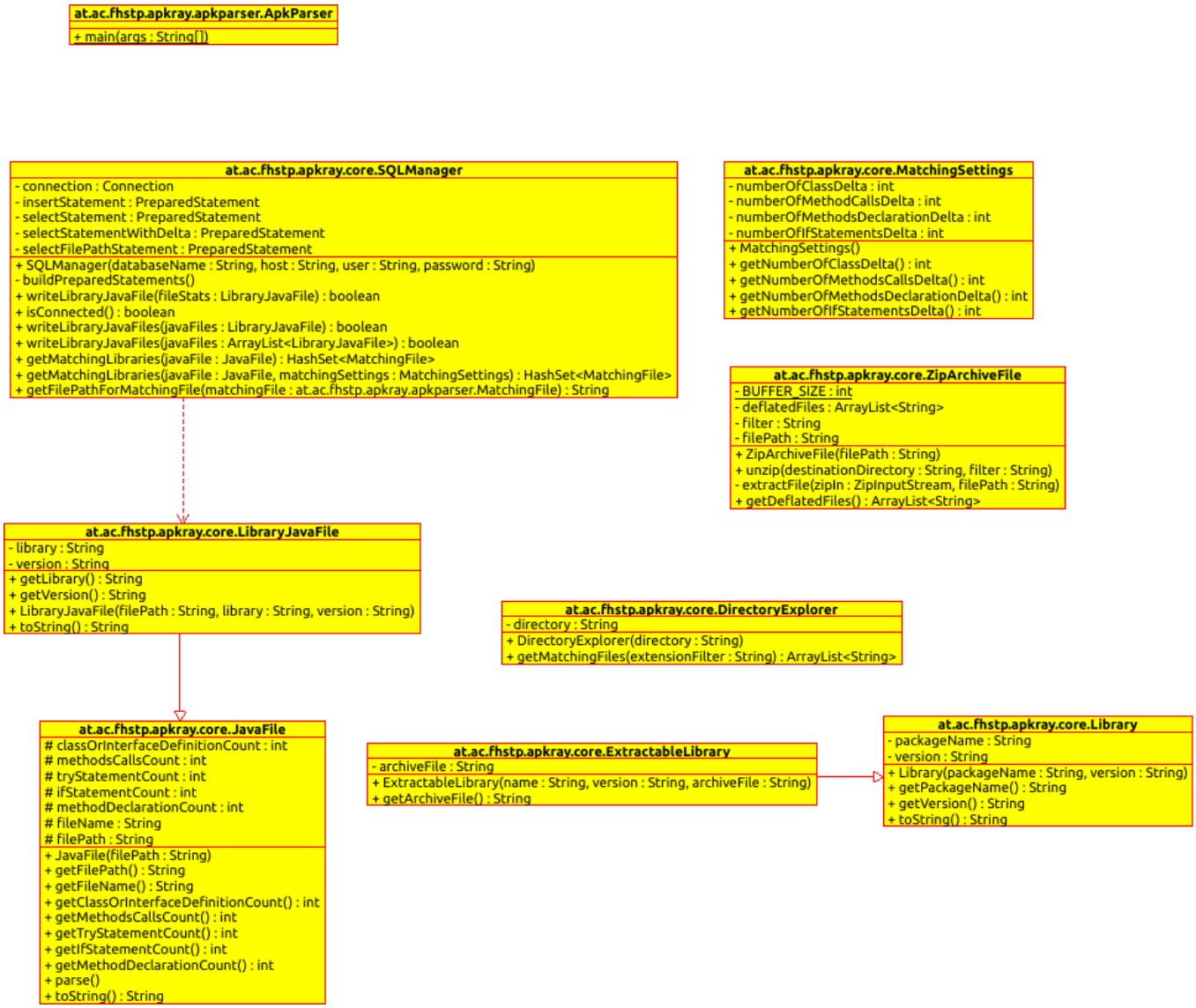


Abbildung 4.1: UML-Diagramm des Core-Packages

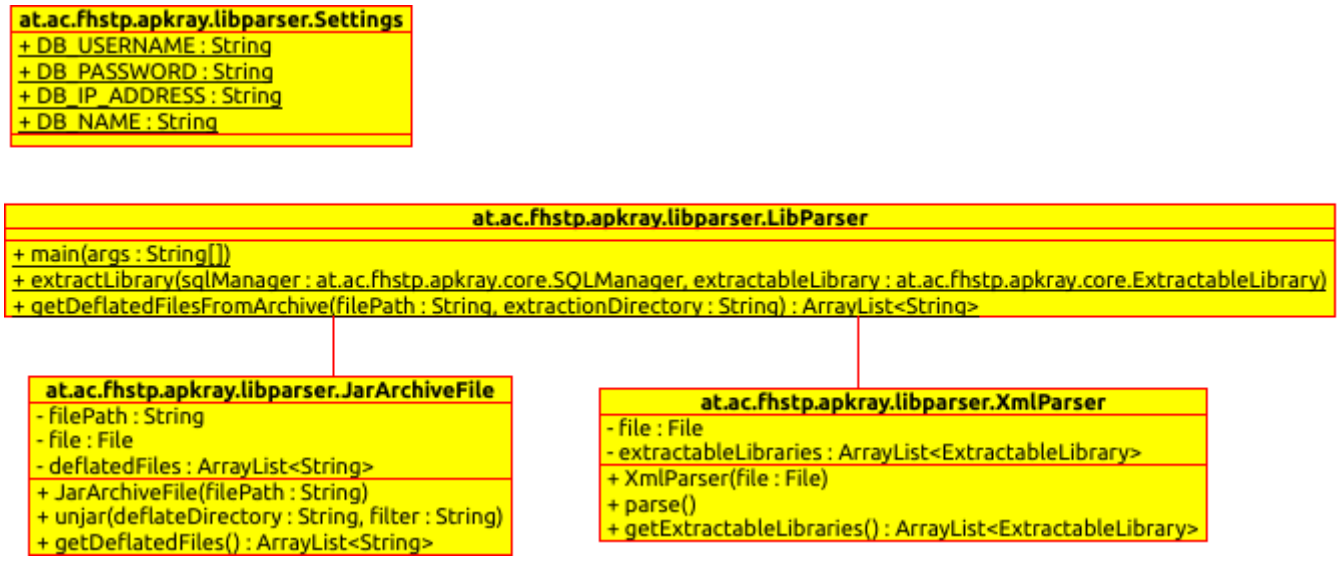


Abbildung 4.2: UML-Diagramm des LibParser-Packages

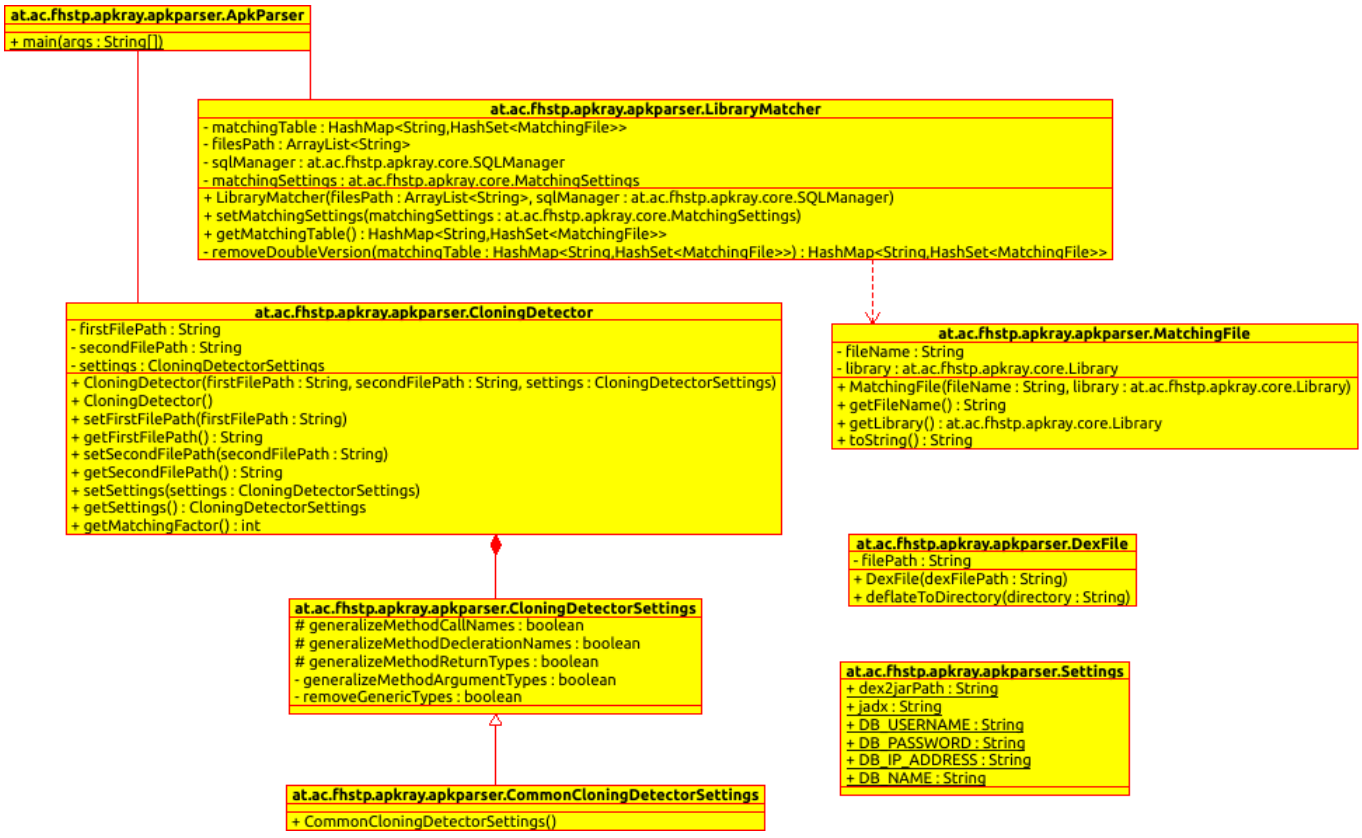


Abbildung 4.3: UML-Diagramm des ApkParser-Packages

Rein organisatorisch gesehen wurde der Quellcode in drei Verzeichnisse unterteilt. Jedes Verzeichnis ist aus Sicht von Java ein Package. Alle Klasse sind folgendermaßen in drei Packages unterteilt:

- Im Package at.ac.fhstp.apkray.core befinden sich alle Klassen, die sowohl vom LibParser als auch vom ApkRay gebraucht werden, etwa Klassen, welche die Verbindung zu der SQL-Datenbank herstellen oder Klassen, die den Umgang mit Files und Verzeichnissen erleichtern.
- Im Package at.ac.fhstp.apkray.libparser befinden sich alle Klassen zum Programm LibParser. In diesem Package befindet sich eine Main-Funktion, welche der Entry-Point vom Programm LibParser ist.
- Im Package at.ac.fhstp.apkray.apkray befinden sich alle Klassen zum Programm ApkRay. In diesem Package ist ebenfalls eine Main-Funktion vorhanden, welche der Entry-Point vom Programm ApkParser ist.

4.3 Libparser

LibParser ist ein für diese Arbeit geschriebenes Programm, das die Datenbank, in der sich die Merkmale aller zu erkennenden Bibliotheken befinden, erstellt und füllt. All die zu erkennenden Bibliotheken werden in einem XML-File vor der Ausführung beschrieben. Dieses File wird dann vom LibParser-Tool gelesen. In diesem File steht sowohl jede Bibliothek, die analysiert werden muss, als auch deren Eigenschaften (der Packagename, die Versionsnummer und der absolute Pfad zum Archiv-File):

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <libraries>
3     <library>
4        <package>org.apache.common.io</package>
5        <version>2.5.1</version>
6        <source>/home/arnold/DA/Files/commons-io-2.5-src.zip</source>
7     </library>
8     <library>
9        <package>com.google.gson</package>
10       <version>2.4</version>
11       <source>/home/arnold/DA/Files/gson-gson-parent-2.4.zip</source>
12    </library>
13 </libraries>
```

Für jede Bibliothek werden drei Tags benötigt:

- <package>: der Packagename der zu analysierenden Bibliothek
- <version>: die Versionsnummer der zu analysierenden Bibliothek
- <source>: der absolute Pfad zum Archiv der zu analysierenden Bibliothek. Dieses File enthält alle Java-Files und kann auf der Homepage des jeweiligen Projekts heruntergeladen werden.

Jede Bibliothek wird dann extrahiert und jedes File nach unseren festgestellten Merkmalen analysiert. Dies wird der Reihe nach erfolgen. Für jede im File beschriebene Bibliothek wird das entsprechende Archiv-File entpackt und all die darin enthaltenen Java Files werden vom Parser gelesen. Diese Zwischenergebnisse werden dann in der Datenbank abgebildet.

4.3.1 Extrahieren

Für jede Bibliothek steht das passende Archiv-File im XML-File. Es kann sich um ein ZIP oder ein Jar-Archive handeln. Beide Typen werden von den entsprechenden Klassen JarArchiveFile und ZipArchiveFile unterstützt.

In den zu analysierenden Archive-Files befinden sich oft Dateien, die im Zuge dieser Arbeit ignoriert gehören, etwa HTML-Dokumentation, TODO-Listen, AUTHORS Files oder Ähnliches. Diese werden vom LibParser-Programm ignoriert, in dem wir dem Extracter einen String geben, welcher als Filter fungiert. Es werden nur die Files extrahiert, die diesem Filter entsprechen. Da sollten nur Files mit .java Extension in Betracht gezogen werden:

```
1  public static ArrayList<String> getDeflatedFilesFromArchive(String filePath, String
    ↪  extractionDirectory) {
2
3      String archiveExtension = FilenameUtils.getExtension(filePath);
4      String javaFileExtension = "java";
5
6      if (archiveExtension.equals("zip")) {
7
8          ZipArchiveFile zipArchiveFile = new ZipArchiveFile(filePath);
9          zipArchiveFile.unzip(extractionDirectory, javaFileExtension);
10
11         return zipArchiveFile.getDeflatedFiles();
12
13     } else if (archiveExtension.equals("jar")) {
14
15         JarArchiveFile jarArchiveFile = new JarArchiveFile(filePath);
16         jarArchiveFile.unjar(extractionDirectory, javaFileExtension);
17
18         return jarArchiveFile.getDeflatedFiles();
19     }
20
21     return null;
22 }
```

ZipArchiveFile und JarArchiveFile bauen jeweils auf der ZipInputStream - beziehungsweise JarFile-Klasse, beide im Package java.util, auf. Mit diesen beiden Klassen lassen sich alle im Archive beinhal-

tene Files auflisten und nach deren Dateiendung sortieren. Nur die Files, die mit .java enden, werden in Betracht gezogen und später vom Parse analysiert.

4.3.2 Parsing und Analyse

JavaParser ist eine open-source Bibliothek, unter LGPL-Lizenz zur Verfügung gestellt, welche sehr viele Operationen rund um Java-Code ermöglicht. Diese Bibliothek wurde besonders für folgende Anwendungsfälle konzipiert:

- Parsing: Da ist es möglich, aus einem Java-Code den AST (*Abstract Syntax Tree*) zu extrahieren. Dies wird zum Beispiel angewendet, wenn die Struktur einer Klasse grafisch abgebildet werden soll.
- Analyse von bestehendem Java-Code: Wie viele Variablen sind in diesen zwei Klassen *protected*. So wird der JavaParser in dieser Arbeit eingesetzt.
- Refactoring: Alle Methoden, die mit get beginnen, werden nun getMember heißen. Das macht zum Beispiel innerhalb einer Entwicklungsumgebung Sinn.
- Generierung: Eine Klasse generieren, die eine calculateValue-Methode hat und zwei Variablen. Wenn zum Beispiel aus einer textuellen Beschreibung oder einem UML-Klassendiagramm Ein kompilierbarer Code generiert werden soll.

Diese Bibliothek wird intensiv betreut und weiterentwickelt, die letzte Version von Java, Java 9, und deren Neuerungen sind schon bestens unterstützt. Aus diesem Grund eignet sich hier JavaParser sehr gut.
Eine der Basis-Klassen dieser Bibliothek ist die VoidVisitorAdapter-Klasse, welche ein Visitor im technischen Kontext ist. Davon erbt die CustomVisitor-Subklasse. Die Mutter-Klasse definiert eine Reihe an Methoden, die jedesmal aufgerufen werden, wenn ein entsprechendes Element im Java-Code gefunden wird. In der VoidVisitorAdapter- Implementierung werden alle Elemente des jeweiligen Java-Codes durchgegangen.
Das ist das Ziel, für jedes gefundene Merkmal, das in Abschnitt 3.2 definiert wurde, den entsprechenden Zähler um eins zu erhöhen. Dafür werden alle passenden Methoden überladen. Diese überladenen Methoden tun hier nichts anderes, als den passenden Zähler zu erhöhen und die Basis-Implementierung aufzurufen.
So sieht die CustomVisitor-Klasse aus:

```java
package at.ac.fhstp.apkray.core;

import com.github.javaparser.ast.*;

public class CustomVisitor extends VoidVisitorAdapter<Object> {

    private int methodCallsCount = 0;
    private int classOrInterfaceDefinitionCount = 0;
    private int tryStatementCount = 0;
    private int ifStatementCount = 0;
    private int methodDeclarationCount = 0;

    @Override
    public void visit(MethodCallExpr n, Object arg) {

        super.visit(n, arg);
        this.methodCallsCount += 1;
    }

    @Override
    public void visit(ClassOrInterfaceDeclaration n, Object arg) {

        super.visit(n, arg);
        this.classOrInterfaceDefinitionCount += 1;
    }

    @Override
    public void visit(TryStmt n, Object arg) {

        super.visit(n, arg);
        this.tryStatementCount += 1;
    }

    @Override
    public void visit(IfStmt n, Object arg) {

        super.visit(n, arg);
        this.ifStatementCount += 1;
    }

    @Override
    public void visit(MethodDeclaration n, Object arg) {

        super.visit(n, arg);
        this.methodDeclarationCount += 1;
    }

    @Override
    public void visit(CompilationUnit n, Object arg) {

```

```
51          super.visit(n, arg);
52          String packageName = n.getPackage().getName().toString();
53
54          int count = 0;
55          for(char c : packageName.toCharArray()) {
56              if('.' == c) {
57                  count++;
58              }
59          }
60          this.packageDepth = count;
61      }
62  }
```

Die letzte Methode ist ein bisschen komplizierter als die Anderen. Sie wird nur einmal pro File aufgerufen mit dem Inhalt des ganzen Files. Um die Tiefe der Ordnerstruktur zu berechnen werden nur die Punkten gezählt. Zum Beispiel für die Bibliothek mylibrary, mit Packagenamen com.mydomain.mylibrary. Da wird eine Tiefe von zwei berechnet. Unterstehend befindet sich eine Tabelle, welche allen anderen überladenen Methoden eine Erläuterung gibt:

Tabelle 4.1: Auflistung aller gesuchten Elemente

Name	Gesuchtes Element
MethodCallExpr	wie viele Methode aufgerufen werden
ClassOrInterfaceDeclaration	wie viele Klassen/Interfaces definiert werden
TryStmt	wie viele Ausnahmen behandelt werden
IfStmt	wie viele if-Anweisungen verwendet werden
MethodDeclaration	wie viele Methode deklariert werden

Zu jedem beobachteten Merkmal wird auch eine get-Methode bereitgestellt, damit diese von anderen Klassen gelesen werden.

Weil alle Merkmale auf einer in beiden Programmen einzigen Weise berechnet werden sollen, wird diese CustomObserver-Klasse an verschiedenen Stellen eingebunden. Diese kommt sowohl im LibParser als auch im ApkParser zum Einsatz – in der JavaFile-Klasse eingebettet. Diese dient als Interface zwischen dem Parser und dem Rest des Codes. Ein JavaFile ist ein File, welches Java-Codes enthält. Jede Instanz dieser Klasse stellt ein anderes File dar und kennt die Eigenschaften des entsprechendes Quellcodes. Diese Klasse stellt eine parse-Methode dar, deren Implementierung untenstehend zu sehen ist:

```
1  public class JavaFile {
2
```

```
3       // ...
4
5       public void parse() {
6
7           CustomVisitor visitor = new CustomVisitor();
8
9           try {
10              File file = new File(this.filePath);
11              visitor.visit(JavaParser.parse(file), null);
12
13          } catch (ParseException | IOException e) {
14
15             new RuntimeException(e);
16          }
17
18          this.classOrInterfaceDefinitionCount =
             ↪  visitor.getClassOrInterfaceDefinitionCount();
19          this.methodsCallsCount = visitor.getMethodCallsCount();
20          this.tryStatementCount = visitor.getTryStatementCount();
21          this.methodDeclarationCount = visitor.getMethodDeclarationCount();
22          this.ifStatementCount = visitor.getIfStatementCount();
23          this.packageDepth = visitor.getPackageDepth();
24      }
25  }
```

Die JavaParser-Klasse wird als *public final* definiert und hilft beim Tokenisieren eines Java-Files. Mit der parse-Methode werden alle Tokens einer bestimmten Java-Datei zurückgeliefert. Diese Token werden dann unserem CustomVisitor übergeben.

Nun sind alle aus den Archiv-Files kommenden Files extrahiert und deren Statistiken berechnet und befinden sich in JavaFile-Instanzen. Diese werden dann in der LibParser-Klasse der Reihe nach vom Parser gelesen und analysiert. Dieser Codeabschnitt eignet sich zum Analysieren einer einzigen Bibliothek. Er befindet sich also innerhalb einer Schleife, die alle zu analysierende Bibliotheken durchgeht.

```
1   ArrayList<String> deflatedFiles = getDeflatedFilesFromArchive(filePath,
     ↪  extractionDirectory);
2
3   String libraryName = extractableLibrary.getPackageName();
4   String libraryVersion = extractableLibrary.getVersion();
5
6   for (String file : deflatedFiles) {
7
8       LibraryJavaFile libraryJavaFile = new LibraryJavaFile(file, libraryName,
         ↪  libraryVersion);
9       libraryJavaFile.parse();
10  }
```

4.3.3 Speichern der Statistiken

Jetzt sind wir soweit, dass wir den Code einer Java-Bibliothek aus einem Jar- beziehungsweise Zip-Archiv extrahieren und analysieren können. Es fehlt nur noch die dauerhafte Speicherung dieser erworbenen Daten. Dazu kommt eine MariaDB-Instanz, eine Fork der bekannten MySQL Datenbank, zum Einsatz.

Jede JavaFile-Instanz kennt, nachdem die parse-Methode aufgerufen wurde, alle Statistiken. Jedoch weiß keine JavaFile-Instanz, aus welcher Bibliothek das jeweilige File herkommt. Aus diesem Grund wird eine weitere Klasse definiert: LibraryJavaFile. Diese erbt von der JavaFile und besitzt zwei weitere private Felder:

```
1  public class LibraryJavaFile extends JavaFile {
2
3      private String library;
4      private String version;
5
6      public String getLibrary() {
7
8          return this.library;
9      }
10
11     public String getVersion() {
12
13         return this.version;
14     }
15
16     public LibraryJavaFile(String filePath, String library, String version) {
17
18         super(filePath);
19
20         this.library = library;
21         this.version = version;
22     }
23
24     // ...
25 }
```

Jede LibraryJavaFile-Instanz verfügt über alle Informationen, die gebraucht werden, um die Statistiken samt Informationen über die jeweilige Bibliothek in die SQL-Datenbank zu speichern. LibraryJavaFile-Instanzen werden einfach an die Klasse übergeben, die in einer direkten Verbindung mit der SQL-Datenbank ist:

```
1  sqlManager = new SQLManager(Settings.DB_NAME, Settings.DB_IP_ADDRESS,
    ↪  Settings.DB_USERNAME, Settings.DB_PASSWORD);
```

```
2
3  // Extrahieren...
4
5  sqlManager.writeLibraryJavaFile(libraryJavaFile);
```

4.4 SQL-Datenbank

Alle Daten werden in einer SQL Datenbank gespeichert. Zum aktuellen Zustand wurde eine MariaDB Datenbank eingesetzt, welche aber durch eine andere SQL-Datenbank ersetzt werden kann.
Die einzige Tabelle der Datenbank wird mit folgendem Befehl erzeugt:

```
1  CREATE TABLE libfile (
2          name varchar(20),
3          version varchar(10),
4          filename varchar(100),
5          classoridef int,
6          methodcalls int,
7          trycount int,
8          packagedepth int,
9          filepath varchar(200)
10 );
```

name ist der Name der Bibliothek, also ein String mit bis zu 20 Buchstaben. *version* ist die Versionsnummer, auch ein String mit bis zu zehn Buchstaben, damit auch längere Versionsnummer wie 1.0.0-rc1 keine Probleme darstellen. *filename* ist der Name des Files, ebenfalls ein String mit bis zu 100 Buchstaben. Dann kommen die Merkmale, *classoridef*, *methodcalls*, *trycount*, *packagedepth*, alle als Ganzzahlen (*int*) gespeichert. Das letzt Feld, *filepath*, ist der absolute Pfad zum File auf der Festplatte und auch ein String mit bis zu 200 Buchstaben.
Es gibt drei Befehle, einen INSERT- und zwei SELECT-Befehle. Wenn die Statistiken eines neuen Java-Files gespeichert werden sollen, typischerweise nach dem Aufruf der parse-Methode einer LibraryJavaFile-Instanz, dann wird folgender SQL-Befehl ausgeführt:

```
1  try {
2
3      // Prepare the INSERT statement
4      insertStatement = connection.prepareStatement("INSERT INTO javafiles "
5              + "(package, version, filename, classorinterface, methodcalls,
                 ↪  methoddeclared, ifstatements, trycatchstatements, packagedepth,
                 ↪  filepath) "
6              + "VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)");
7
```

```
8  } catch (SQLException e) {
9
10     e.printStackTrace();
11 }
```

Dieser SQL-Befehl wird in der writeLibraryJavaFile-Methode ausgeführt. In dieser Methode werden nur die verschiedenen Felder ein aus der LibraryJavaFile-Instanz kommenden Wertes zugewiesen:

```
1  public boolean writeLibraryJavaFile(LibraryJavaFile fileStats) {
2
3      try {
4
5          insertStatement.setString(1, fileStats.getLibrary());
6          insertStatement.setString(2, fileStats.getVersion());
7          insertStatement.setString(3, fileStats.getFileName());
8          insertStatement.setInt(4, fileStats.getClassOrInterfaceDefinitionCount());
9          insertStatement.setInt(5, fileStats.getMethodsCallsCount());
10         insertStatement.setInt(6, fileStats.getMethodDeclarationCount());
11         insertStatement.setInt(7, fileStats.getIfStatementCount());
12         insertStatement.setInt(8, fileStats.getTryStatementCount());
13         insertStatement.setString(9, fileStats.getFilePath());
14
15         return insertStatement.execute();
16
17     } catch (SQLException e) {
18
19         e.printStackTrace();
20     }
21
22     return false;
23 }
```

Dazu kommen noch zwei auf SELECT basierende Befehle. Es können mehrere Files ohne Delta gesucht werden, wenn nach genau einem JavaFile gesucht werden soll. Es kann aber auch nach mehreren Files mit Deltawert gesucht werden, wenn ein aus dem APK extrahierter File analysiert wurde und nach ähnlichen Dateien gesucht wird. Die Theorie zum Deltawert ist in **??** zu finden. Weil der Obfuscator den Code von der Semantik her leicht ändert, sollte nun nach Files gesucht werden, in denen zwischen ein und fünf Methoden aufgerufen werden und in denen sich zwischen 0 und 2 Try-Catch befinden sollen. Die Abweichung wird mit der Klasse MatchingSettings materialisiert:

```
1  package at.ac.fhstp.apkray.core;
2
3  public class MatchingSettings {
4
```

```
5       private int numberOfClassDelta;
6       private int numberOfMethodCallsDelta;
7       private int numberOfMethodsDeclarationDelta;
8       private int numberOfIfStatementsDelta;
9
10      public MatchingSettings() {
11
12          this.numberOfClassDelta = 2;
13          this.numberOfMethodCallsDelta = 2;
14          this.numberOfMethodsDeclarationDelta = 2;
15          this.numberOfIfStatementsDelta = 2;
16      }
17
18      public int getNumberOfClassDelta() {
19
20          return this.numberOfClassDelta;
21      }
22
23      public int getNumberOfMethodsCallsDelta() {
24
25          return this.numberOfMethodCallsDelta;
26      }
27
28      public int getNumberOfMethodsDeclarationDelta() {
29
30          return this.numberOfMethodsDeclarationDelta;
31      }
32
33      public int getNumberOfIfStatementsDelta() {
34
35          return this.numberOfIfStatementsDelta;
36      }
37  }
```

Der Deltawert kann für jedes Merkmal einzeln angepasst werden. Generell sei an dieser Stelle gesagt, dass, je höher der Deltawert, desto mehr Files im vom SELECT-Befehl zurückgelieferten Set zu finden werden. Dafür wird aber generell die Genauigkeit des Erkennungsverfahrens verbessert, weil mehr Files in die *Code-Cloning-Detection* gehen. Es können entweder die schon angepassten Settings-Klassen angewendet oder eigene Werte definiert werden.

Die zu diesem Schema passenden SQL-Befehle sind in der Klasse SQLManager gekapselt. Alle SQL-Befehle sind mit PreparedStatements implementiert. Mit dieser Klasse wird der Ausführungsplan nur einmal von der jeweiligen Datenbank berechnet, und zwar, wenn der SQL-Befehl initialisiert wird. Dies garantiert eine schnelle Ausführung der SQL-Befehle, weil sie schon übersetzt wurden und zur Ausführungszeit in einem optimalen Format vorliegen [13]. Der *selectStatementWithDelta* zieht Nutzen aus dieser MatchingSettings-Klasse.

```
1  private PreparedStatement insertStatement = null;
2  // Prepare the SELECT statement
3  selectStatement = connection.prepareStatement("SELECT package, version, filename "
4          + "FROM javafiles WHERE classorinterface = ? AND methodcalls = ? AND
           ↪  methoddeclared = ? AND ifstatements = ?");
5
6  private PreparedStatement selectStatement = null;
7  // Prepare the SELECT statement
8  selectStatementWithDelta = connection.prepareStatement("SELECT package, version,
    ↪  filename "
9          + "FROM javafiles WHERE classorinterface BETWEEN ? AND ? AND methodcalls
           ↪  BETWEEN ? AND ? "
10         + "AND methoddeclared BETWEEN ? AND ? AND ifstatements BETWEEN ? AND ?");
11
12 private PreparedStatement selectStatementWithDelta = null;
13 // Prepare the SELECT statement
14 selectFilePathStatement = connection.prepareStatement("SELECT filepath FROM
    ↪  javafiles "
15         + "WHERE filename LIKE ? AND package LIKE ? AND version LIKE ?");
```

Nun wurden also alle zu erkennenden Bibliotheken extrahiert und analysiert. Deren Merkmale befinden sich in der SQL-Datenbank abgespeichert und es kann mit der Analyse der fertigen Applikation begonnen werden.

4.5 Apkparser

ApkParser ist ein in Java geschriebenes Tool, welches auf der von LibParser gebildeten Datenbank und der JavaParser-Bibliothek aufbaut. Die zu analysierende Applikation wird in einer ersten Phase in lesbaren Java-Quellcode konvertiert. Die daraus entstehenden Files werden dann analysiert und mit jenen in der Datenbank verglichen.
Für jedes File aus der Applikation wird versucht, ähnliche Files in der Datenbank zu finden. In einer weiteren Phase werden sie dann genauer verglichen.

4.5.1 Vom APK-File zu Java

Startpunkt des Erkennungsverfahrens ist das fertige APK-File, das zu analysieren ist. Aus diesem File soll das classes.dex File extrahiert und in ein separates Verzeichnis abgelegt werden. Das Extrahieren wird mit der Klasse ZipArchiveFile durchgeführt, da das APK-Fileformat eigentlich ein ZIP-Archiv ist. Wir nutzen die Gelegenheit, hier einen Patern eingeben zu können. Damit werden alle Files, die nicht mit .dex enden, im Prozess einfach ignoriert.

```
ApkFile apkFile = new ApkFile(apkFilePath);
ArrayList<String> dexClasses = new ArrayList<>();

// Default working directory
String workingDirectory = System.getProperty("user.home") + File.separator +
 ↪  "ApkRay" + File.separator + apkFile.getAppName();

if (args.length == 2) {
    if (Utils.directoryExists(args[1])) {
        workingDirectory = args[1] + File.separator + apkFile.getAppName();
    }
}
```

Wobei die ApkFile-Klasse eigentlich nur ein Wrapper um die ZipArchiveFile ist:

```
public class ApkFile {

    private String filePath;
    private String appName;

    public ApkFile(String filePath) {

        this.filePath = filePath;

        // Compute app name
        File tmpFile = new File(this.filePath);
        String fileName = tmpFile.getName();
        fileName.replace(' ', '_');
        this.appName = FilenameUtils.removeExtension(fileName);
    }

    public String getAppName() {

        return this.appName;
    }

    public ArrayList<String> extractDexClasses(String extractionDirectory) throws
     ↪  IOException {

        ZipArchiveFile zipArchiveFile = new ZipArchiveFile(this.filePath);
        zipArchiveFile.unzip(extractionDirectory, "dex");

        return zipArchiveFile.getDeflatedFiles();
    }
}
```

In dieser ersten Stufe wurde das classes.dex-File extrahiert und liegt nun im Filesystem zur Verfügung.

Es wurde vorher erklärt, wie aus Java-Files die Merkmale berechnet werden. Dazu werden eben Java-Files benötigt und keine classes.dex-Files. Die nächste Stufe des Erkennungsverfahrens ist also die Konvertierung dieses Files in zumindest für eine Maschine lesbaren Java-Code, welcher dann vom Java-Parser gelesen und analysiert werden kann. An dieser Stelle wird ein Dekompiler gebraucht, weil der sich im classes.dex-File befindede Code eigentlich ein kompilierter Java-Code ist (.class-Files).
Die Java-Dekompilierung ist wie *Code-Cloning-Detection* kein neuartiges Thema, sondern ein Forschungsthema, das auch in der Industrie Echo findet. Unter den verschiedenen am Markt verfügbaren Dekompilern wurde jadx ausgewählt. Dieser verfügt unter anderem über folgende Funktionen:

- Kann direkt mit dem .dex-File arbeiten, es wird kein manuelles Extrahieren der .class-Files benötigt
- open-source, von daher erweiterbar und anpassbar
- relativ gut betreut und weiterentwickelt

Jadx wurde schon in 2.2 (Seite 16) erwähnt und dessen Benutzeroberfläche kurz vorgestellt. Mithilfe dieses Tools ist es dann möglich, den Quellcode der fertigen Applikation in einem Verzeichnis extrahiert zu bekommen. Jadx wird gleich im Java-Code auf transparente Art und Weise aufgerufen:

```
1  public class DexFile {
2
3      private String filePath;
4
5      public DexFile(String dexFilePath) {
6
7          this.filePath = dexFilePath;
8      }
9
10     public void deflateToDirectory(String directory) {
11
12         try {
13
14             String command = Settings.jadx + " -d " + directory + " " +
                ↪  this.filePath;
15             Process jadxProcess = Runtime.getRuntime().exec(command);
16             jadxProcess.waitFor();
17
18         } catch (IOException | InterruptedException e) {
19
20             e.printStackTrace();
21         }
22     }
23 }
```

Die waitFor()-Methode ist in Java eine sogenannte blocking-Methode. Das heißt, dass die Ausführung des Programms erst nach der Rückgabe der Methode fortgesetzt wird. Dieses Mechanismus garantiert uns, dass alle Klassen und Files erfolgreich extrahiert wurden, bevor die Code-Analyse gestartet wird. Obwohl jadx ein gereiftes Tool ist, bereiten ihm manche Codeabschnitte Probleme. Das heißt also, dass in den dekompilierten Java-Klassen manchmal unbrauchbare Codezeile zu finden sind. Dies ist leider in keinster Weise umgehbar, es muss also im gesamten Prozess damit gerechnetwerden.

4.5.2 Vergleich

Es ist nun so weit, dass der aus dem Dex-File extrahierte Java-Code im Filesystem liegt und bereit zur Auswertung ist.
Es können nicht alle extrahierte Files betrachtet werden, das wäre ein viel zu langer Prozess. Die Idee ist also, nur einen Teil davon zu analysieren. Innerhalb der Klasse ApkParser wird die komplette Liste geliefert:

```
ArrayList<String> matchingFiles = new
 ↪  ArrayList<>(directoryExplorer.getMatchingFiles("java"));
```

Diese Liste enthält alle Java-Files, die analysiert werden könnten. Jedoch können nicht alle Files analysiert werden, sonst könnte das Erkennungsverfahren eine Ewigkeit dauern. Es wurde eine Methode entwickelt, die in dieser List eine bestimmte Anzahl an Dateien nimmt, welche eine definierte Länge haben. Diese Funktion schließt auch alle Files aus, die sich in einem `android` Verzeichnis befinden. In diesem Verzeichnis befinden sich die v4/v7/v13 Bibliotheken, welche von Google entwickelt werden.

```
public static ArrayList<String> selectFilesLongerThan(ArrayList<String> filePaths,
 ↪  int count, int minimalLength) {

    ArrayList<String> selectedFiles = new ArrayList<>();
    Collections.shuffle(filePaths);

    for (String filePath : filePaths) {

        if (selectedFiles.size() >= count) {

            return selectedFiles;

        } else {

            Path path = Paths.get(filePath);
                  try {
                long lineCount = Files.lines(path).count();
```

```
17
18                  if (lineCount >= minimalLength && !filePath.contains("android")) {
19                            selectedFiles.add(filePath);
20                  }
21
22                    } catch (IOException e) {
23
24                  e.printStackTrace();
25              }
26          }
27      }
28
29      return selectedFiles;
30  }
```

Diese Methode wird dann aufgerufen:

```
1  ArrayList<String> filesToParse = Utils.selectFilesLongerThan(matchingFiles, 14, 20);
```

Im Unterabschnitt 3.6.1 wurde das Konzept einer Matching-Tabelle zum ersten Mal vorgestellt und erklärt. Deren Logik und Implementierung befindet sich in der Klasse LibraryMatcher. Für jedes File dieser Liste wird eine Instanz der JavaFile-Klasse erstellt, welche Instanz dazu dient, dass Statistiken berechnet werden. Dieser Schritt findet in der LibraryMatcher-Klasse statt:

```
1  public class LibraryMatcher {
2
3      private HashMap<String, HashSet<MatchingFile>> matchingTable;
4      private ArrayList<String> filesPath;
5      private SQLManager sqlManager;
6      private MatchingSettings matchingSettings;
7
8
9      public LibraryMatcher(ArrayList<String> filesPath, SQLManager sqlManager) {
10
11         this.matchingTable = new HashMap<String, HashSet<MatchingFile>>();
12         this.filesPath = filesPath;
13         this.sqlManager = sqlManager;
14     }
15
16     public void setMatchingSettings(MatchingSettings matchingSettings) {
17
18         this.matchingSettings = matchingSettings;
19     }
20
21     public HashMap<String, HashSet<MatchingFile>> getMatchingTable() {
22
```

```
23          for (String filePath : this.filesPath) {
24              JavaFile javaFile = new JavaFile(filePath);
25              javaFile.parse();
26              HashSet<MatchingFile> matchingFiles =
                  ↪  sqlManager.getMatchingLibraries(javaFile, matchingSettings);
27              this.matchingTable.put(filePath, matchingFiles);
28          }
29
30          return this.matchingTable;
31      }
32  }
```

In der Methode getMatchingTable() wird für jeden absoluten Pfad eine Instanz der klasse JavaFile erstellt. Mithilfe dieser lassen sich alle benötigten Statistiken berechnen. Diese werden dann dem SQL-Manager über die getMatchingLibraries()-Methode übergeben. Dieser Methode wird auch eine Instanz der Klasse MatchingSettings übermittelt, über welche die erlaubten Abweichungen festgestellt werden. Diese Klasse ist im Prinzip ein Wrapper um die Klasse SQLManager. Alle Ergebnisse aus dieser Klasse werden dann in eine andere Datenstruktur verpackt und in der getMatchingTable()-Methode zurückgeliefert.
In diesem Codeabschnitt lässt sich erkennen, dass die sogenannte Matching-Tabelle in Java folgender Datenstruktur entspricht:

```
1  HashMap<String, HashSet<MatchingFile>> matchingTable;
```

In Java ist HashMap eine schnelle und effiziente Implementierung einer Hash-Tabelle. In unserem Fall bedeutet das jetzt, dass jedem Key mindestens ein HashSet zugewiesen wird. Dieses HashSet enthält Instanzen von der MatchingFile-Klasse. Links ist also der Pfad zu dem aus der Applikation extrahierten File, und rechts stehen eins oder mehrere MatchingFile-Instanzen (mögliche aus einer Bibliothek kommende Files).
Dank der Implementierung der HashMap ist es relativ günstig, sowohl mit einem bestimmten Key auf eines oder mehrere MatchingFile zu kommen, als auch über alle MatchingFile-Instanzen zu iterieren (mit der dafür vorgesehenen entrySet()-Methode). Außerdem HashMap sollte seinem Konkurrent HashTable in Single-Thread Programmen bevorzugt werden [12].

4.5.3 Optimierungen

Wir müssen nicht aus den Augen verlieren, dass wir durch diesen ersten Schritt schon mehrere False-Positives bekommen. Vor allem, wenn sich viele Bibliotheken in jeweils zwei oder drei verschiedenen Versionen in der Datenbank befinden. Alle für ähnlich erklärte Files werden 1:1 verglichen. Diese

Vergleich-Operation ist CPU- und IO-intensiv, es müssen also Optimierungen durchgeführt werden, damit der Ablauf trotzdem relativ schnell bleibt.

Version-Optimierung

Alle Optimierungen befinden sich in der Klasse LibraryMatcher, die im Unterabschnitt 4.5.2 beschrieben wurde. Die erste umgesetzte Optimierung wurde im Abschnitt 3.6.2 ausführlich beschrieben. Deren Implementierung wird in diesem Absatz detailliert beschrieben.

```
1  private HashMap<String, HashSet<MatchingFile>> removeDoubleVersion(HashMap<String,
    ↪  HashSet<MatchingFile>> matchingTable) {
2
3      HashMap<String, HashSet<MatchingFile>> simplifiedMatchingTable = new
        ↪  HashMap<>();
4
5      for (String file : this.filesPath) {
6
7          ArrayList<String> doublesID = new ArrayList<>();
8          HashSet<MatchingFile> matchesForFile = new HashSet<>();
9
10         HashSet<MatchingFile> matchingFiles = matchingTable.get(file);
11
12         for (MatchingFile matchingFile : matchingFiles) {
13
14             String id = matchingFile.getLibrary().getPackageName() + "." +
                ↪  matchingFile.getFileName();
15
16             if (! doublesID.contains(id)) {
17                 matchesForFile.add(matchingFile);
18                 doublesID.add(id);
19             }
20         }
21
22         simplifiedMatchingTable.put(file, matchesForFile);
23
24         System.out.println("Shrinking for file " + file + " from " +
            ↪  matchingFiles.size() + " to " + matchesForFile.size());
25     }
26
27     return simplifiedMatchingTable;
28 }
```

Die Optimierung wird folgendermaßen implementiert: Für jedes betrachtete File wird ein Kenner aus dem Filenamen und dem Files Packagenamen berechnet (im Code *id* genannt). Nehmen wir an, das File ABC.java aus dem Package at.fhstp.mylib in Version 1.0 wird analysiert, dann sieht der Kenner wie folgt aus: at.fhstp.mylib.ABC.java. Dann kommt das File ABC.java aus dem Package at.fhstp.mylib in

Version 1.1. Für dieses zweite File ist der Kenner: at.fhstp.mylib.ABC.java. Dieser befindet sich schon in der doublesID-List, das File wird also ignoriert.
Sollte dieses File in einer anderen Version noch einmal vorkommen, würde genau dasselbe passieren. Der Grund ist jener, dass Files aus ähnlichen Versionen kaum voneinander abweichen. Darum werden sie eben vom SQLManager gefunden. Da wir wissen, dass eine Vergleich-Operation viel Zeit kostet, macht es schon Sinn, diese Optimierungen zu haben.

4.6 Code-Cloning

4.6.1 Einbindung in das Projekt

In der Klasse ApkParser kommt die LibraryMatcher-Klasse zum Einsatz, um die Matching-Tabelle zu berechnen:

```
1 LibraryMatcher libraryMatcher = new LibraryMatcher(filesToParse, sqlManager);
2 libraryMatcher.setMatchingSettings(new MatchingSettings());
3 HashMap<String, HashSet<MatchingFile>> matchingTable =
   ↪  libraryMatcher.getMatchingTable();
```

Die von der LibraryMatcher generierten Daten dienen als Basis zur *Code-Cloning-Detection*.
Für das Code-Cloning kommt das jccd-Projekt zum Einsatz. Jccd ist eine Open-Source API, welche verschiedene Klassen zum Code-Cloning-Detection zur Verfügung stellt. Die API ist erweiterbar und verfügt über mehrere sogenannte Pipelines (Erkennungsmethoden), eines davon ist die auf dem AST-basierende Methode, welche in Unterabschnitt 3.6.3 erwähnt wurde. Obwohl diese Bibliothek erweiterbar, open-source und in ihrer Genre einzigartig ist, wird sie leider nicht mehr aktiv weiterentwickelt. Die letzten Änderungen stammen laut Git-Repository aus dem Jahr 2011. Mit diesem Programm lassen sich aus dem APK-File analysierte Codeabschnitte mit den passenden Known-Files verglichen, je nach Zwischenergebnisse in der Matching-Tabelle.
In diesem Projekt ist es leider nicht vorgesehen, dass man direkt auf die internen Werte zugreift. Im originalen Code können Ergebnisse nur angezeigt werden, aber es werden keine Ergebnisse in Form von Zahlen zur Verfügung gestellt. Weil es aber das ist, was hier benötigt wird, wurde die Bibliothek gepatched (auf der CD als jccd.patch zu finden).
Die Implementierung dieses Verfahrens befindet sich in der CodeCloningDetector-Klasse. Hier wird eine Instanz der Klasse ASTDetector erzeugt, welcher dann zwei Files übergeben werden. In dieser weiteren Klasse befinden sich die Vergleich-Algorithmen:

```java
import org.eposoft.jccd.data.JCCDFile;
import org.eposoft.jccd.detectors.APipeline;
import org.eposoft.jccd.detectors.ASTDetector;

public class CodeCloningDetector {

    private String firstFilePath;
    private String secondFilePath;
    private CodeCloningDetectorSettings settings;

    public CodeCloningDetector(String firstFilePath, String secondFilePath,
     ↪  CodeCloningDetectorSettings settings) {

        this.firstFilePath = firstFilePath;
        this.secondFilePath = secondFilePath;
        this.settings = settings;
    }

    public CodeCloningDetector() {

    }

    public void setFirstFilePath(String firstFilePath) {

        this.firstFilePath = firstFilePath;
    }

    public void setSecondFilePath(String secondFilePath) {

        this.secondFilePath = secondFilePath;
    }

    public int getMatchingFactor() {

        APipeline detector = new ASTDetector();

        JCCDFile[] files = {
                new JCCDFile(this.firstFilePath),
                new JCCDFile(this.secondFilePath)
        };

        detector.setSourceFiles(files);

        return APipeline.getSimilarityIndex(detector.process());
    }
}
```

Mit der setSettings-Methode werden die anzuwendenden Operatoren konfiguriert. Eingentlich ist die CodeCloningDetectorSettings-Klasse nichts mehr als eine List an Operatoren. Wie bei der MatchingSettings-

Klasse sind bereits konfigurierte Klassen zu finden, welche LooseCodeCloningDetectionSettings, NormalCodeCloningDetectionSettings und ConservativeCodeCloningDetectionSettings heißen. Dank dieser Klassen können Auswirkungen der angewendeten Operatoren gemessen werden. Um die bestmögliche Detektion zu gewährleisten, werden verschiedene Operatoren verwendet. Eine ausführliche Liste aller Operatoren kann auf der Homepage des Projekts gefunden werden. Nicht alle Operatoren sind in jedem Fall sinnvoll. In diesen Konfigurationsklassen werden folgende Operatoren bereitgestellt:

- GeneralizeMethodArgumentTypes: Typ von Methoden-Argumenten werden weggenommen. Dann wird diese Funktion computeFactorial(int x) durch computeFactorial(x) ersetzt.
- GeneralizeMethodDeclarationNames: Methoden-Namen werden im Vergleichsverfahren nicht mehr in Betracht gezogen. Dieser zählt im Android Usecase zu einem der wichtigsten Operatoren, weil davon ausgegangen werden muss, dass sehr viele Methoden (wenn nicht alle) vom Obfuskator umbenannt werden.
- GeneralizeMethodReturnTypes: Der Rückgabewert-Typ wird gelöscht, int getMemberVariable() wird zu getMemberVariable().
- GeneralizeVariableNames: Alle Variablennamen werden gelöscht, int factorial(int x) wird int factorial(int).
- RemoveGenericTypes: HashMap<String, MeineKlasse> wird HashMap. Jegliche Typ-Informationen werden nicht mehr betrachtet.
- RemoveGetMethodDeclerations: Alle *getters*, also get-Methoden, werden ignoriert.
- RemoveIsMethodDeclerations: Ähnlich wie bei RemoveGetMethodDeclerations, alle is-Methoden werden ignoriert.
- RemoveSimpleMethods: Alle einfache Methoden werden ignoriert (zum Beispiel Methoden, die nur aus einem *return*-Statement bestehen).

Den CodeCloningDetector-Instanzen werden dann zwei zu vergleichende Files gegeben, daraus wird eine Zahl berechnet. Diese Zahl, auch Matchingfaktor benannt, wird von folgender Funktion berechnet:

```
1 public static int getSimilarityIndex(final SimilarityGroupManager groupContainer) {
2     // output similarity groups
3     SimilarityGroup[] simGroups = groupContainer.getSimilarityGroups();
4     int similarityIndex = 0;
5
```

```
6       if (null == simGroups) {
7           simGroups = new SimilarityGroup[0];
8       }
9
10      if ((null != simGroups) && (0 < simGroups.length)) {
11          for (int i = 0; i < simGroups.length; i++) {
12              final ASourceUnit[] nodes = simGroups[i].getNodes();
13              for (int j = 0; j < nodes.length; j++) {
14                  final SourceUnitPosition minPos = getFirstNodePosition((ANode)
                     ↪  nodes[j]);
15                  final SourceUnitPosition maxPos = getLastNodePosition((ANode)
                     ↪  nodes[j]);
16
17                  ANode fileNode = (ANode) nodes[j];
18                  while (fileNode.getType() != NodeTypes.FILE.getType()) {
19                      fileNode = fileNode.getParent();
20                  }
21
22                  // Summ up total of matching lines
23                  int matchingLinesCount = maxPos.getLine() - minPos.getLine();
24                  similarityIndex += matchingLinesCount;
25              }
26          }
27      }
28
29      return similarityIndex;
30  }
```

Um diesen Abschnitt verstehen zu können, muss man mit einigen Klassen der Jccd-Bibliothek vertraut sein:

- SimilarityGroupManager: In dieser Klasse werden Informationen über ähnliche Codeabschnitte gespeichert.
- ANode: Eine ANode-Instanz ist ein Element des AST. Jede ANode-Instanz kann entweder ein Blatt (*Leaf*) sein oder selber Kinder (*Child*) haben.
- SourceUnitPosition: Dies ist eine einfache Klasse, die nur aus zwei *int* besteht und den Anfang beziehungsweise das Ende eines gemeinsamen Codeabschnittes abgrenzt.

Die CloningDetector-Klasse wird im folgenden Abschnitt verwendet. An dieser Stelle ist bekannt, den Statistiken nach welches aus der Applikation extrahierte File mit welches in der Datenbank sein könnte. Dies muss nun mittels *Code-Cloning-Detection* geprüft werden.

```
1   CodeCloningDetector codeCloningDetector = new CodeCloningDetector();
2   CommonCloningDetectorSettings commonCloningDetectorSettings = new
     ↪  CommonCloningDetectorSettings();
```

```
3  codeCloningDetector.setSettings(commonCloningDetectorSettings);
4
5  HashMap<String, HashMap<MatchingFile, Integer>> matchingScoresTable = new
     ↪  HashMap<>();
6
7  for (String filePath : filesToParse) {
8
9      HashSet<MatchingFile> tmpMatchingFiles = matchingTable.get(filePath);
10
11     // If matching files were found
12     if (tmpMatchingFiles != null)  {
13
14         codeCloningDetector.setFirstFilePath(filePath);
15         HashMap<MatchingFile, Integer> matchingScores = new HashMap<>();
16
17         for (MatchingFile matchingFile : tmpMatchingFiles) {
18
19             String sqlFilePath =
                 ↪  sqlManager.getFilePathForMatchingFile(matchingFile);
20             codeCloningDetector.setSecondFilePath(sqlFilePath);
21             matchingScores.put(matchingFile,
                 ↪  codeCloningDetector.getMatchingFactor());
22         }
23
24         matchingScoresTable.put(filePath, matchingScores);
25     }
26 }
```

4.6.2 Auswertung der Ergebnisse

Am Ende des *Code-Cloning-Detections* wird folgende Tabelle gefüllt:

Tabelle 4.2: Ergebnisse von Code-Cloning-Detection

File aus der App	Package	File aus der Bibliothek
ABC.java	org.package.mylib	Auto.java
DQOK.java	org.package2.mylib2	LaserBeam.java
DCNBS.java	org.package3.mylib3	Banana.java

Diese wird folgendermaßen berechnet:

- Zuerst wird die Matching-Tabelle als Input genommen. Jedes Match wird von jccd geprüft
- Für jedes aus der Applikation analysierte File werden alle Ähnlichkeitsfaktoren verglichen und nur der höchste Wert samt das jeweilige File werden behalten.

- Schließlich werden all diese Files in eine Tabelle vereint. Diese Tabelle stellt das Ende des Erkennungsverfahrens dar.

Dies wird folgendermaßen implementiert:

```
1  HashMap<String, CodeCloningResult> resultsMap = new HashMap<>();
2
3  for (String filePath : filesToParse) {
4
5      HashMap<MatchingFile, Integer> results = matchingScoresTable.get(filePath);
6      int bestScore = 0;
7      String bestId = "NONE";
8      String bestPackage = "NONE";
9      String bestVersion = "NONE";
10     String bestFileName = "NONE";
11
12     for (Entry<MatchingFile, Integer> entry : results.entrySet()) {
13
14         int fileScore = entry.getValue().intValue();
15
16         if (fileScore > bestScore) {
17             bestScore = fileScore;
18             bestPackage = entry.getKey().getLibrary().getPackageName();
19             bestVersion = entry.getKey().getLibrary().getVersion();
20             bestFileName = entry.getKey().getFileName();
21         }
22     }
23
24     CodeCloningResult tmpResult = new CodeCloningResult(bestPackage, bestVersion,
       ↪  bestFileName, bestScore);
25     resultsMap.put(filePath, tmpResult);
26 }
```

An dieser Stelle ist jetzt klar, mit welchem File in der SQL-Datenbank jedes aus der Applikation extrahierte und analysierte File am Ähnlichsten ist. Wenn für ein aus der Applikation extrahierte File das beste immer noch NONE ist, heißt das, dass kein ähnliches File gefunden wurde. Diese Tabelle, die sich immer nur noch im Arbeitsspeicher befindet, muss nun abgespeichert werden.

4.7 Erstellung des Reports

Der Report kann entweder als reiner Text oder als PDF-Dokument erhalten werden. Das PDF-Dokument wird nur generiert, wenn eine LaTeX-Umgebung vorhanden ist.
Ein Java-Interface namens AbstractReportGenerator definiert zwei Methoden, die von den anderen Generatoren dann implementiert werden.

```
package at.ac.fhstp.apkray.apkparser;

import at.ac.fhstp.apkray.core.Library;

public interface AbstractReportGenerator {

        public void addLibrary(Library library);

        public void generateTo(String where);
}
```

Im Fall eines Text-Reports wollen wir nur ein einziges File auf der Festplatte schreiben. Handelt es sich jedoch um ein PDF-Dokument, das mit LaTeXerzeugt wird, wird eher innerhalb eines Unterverzeichnisses gearbeitet.

Zuerst muss herausgefunden werden, ob eine LaTeX-Installation überhaupt auf der Maschine vorhanden ist. Dafür wird einfach im PATH nach dem Programm pdflatex gesucht. Wenn sich diese Datei im PATH befindet, kann davon ausgegangen werden, dass eine vollständige LaTeX-Installation auf diesem Computer abgeschlossen wurde.

```
boolean latexInstalled = true;

try {

    Runtime.getRuntime().exec("pdflatex --version");

} catch (IOException e) {

    latexInstalled = false;
}
```

In der resultsMap-HashMap befinden siche alle Informationen, die zur Abbildung der Ergebnisse gebraucht werden. Für jedes analysierte File passiert Folgendes:

- Wurde das File erkannt, werden der Packagename, der Dateiname, die Versionsnummer angezeigt. All diese Informationen befinden sich in der CodeCloningResult-Instanz.
- Wurde das File jedoch nicht erkannt, wird dies explizit geschrieben.

Sollte eine TXT-Report generiert werden, ergibt sich foldender Abschnitt:

```
File resultFile = new File(workingDirectory + File.separator + apkFile.getAppName()
 ↪  + ".txt");
logFile.createNewFile();
```

```
3  FileOutputStream outputStream = new FileOutputStream(resultFile);
4  OutputStreamWriter outputStreamWriter = new OutputStreamWriter(outputStream);
5  Writer resultWriter = new BufferedWriter(outputStreamWriter);
6
7  resultWriter.write(Utils.getAsciiHeader());
8
9  resultWriter.write(lineBreak);
10 resultWriter.write("App analyzed: " + args[0] + lineBreak);
11 resultWriter.write("Number of extracted files: " + matchingFiles.size() +
    ↪  lineBreak);
12 resultWriter.write("Number of analyzed files: " + filesToParse.size() + lineBreak);
13
14 for (Entry<String, CodeCloningResult> entry : resultsMap.entrySet()) {
15
16     if (!entry.getValue().getPackageName().equals("NONE")) {
17
18         resultWriter.write(entry.getKey() + " is " +
19                            entry.getValue().getPackageName() + "." +
20                            entry.getValue().getVersion() + "." +
21                            entry.getValue().getFileName() + " with " +
22                            entry.getValue().getMatchingFactor() + lineBreak);
23
24     } else {
25         resultWriter.write(entry.getKey() + " not recognized" + lineBreak);
26     }
27 }
28
29 resultWriter.close();
```

5 Tests und Ergebnisse

Allgemein zu den Tests sei es erwähnt, dass wegen der rechtlichen Lage, nur zu diesem Zweck entwickelte beziehungsweise open-source Applikationen getestet werden können.
Alle Tests wurden auf einem DELL Latitude E7450 (Intel(R) Core(TM) i5-5300U CPU @ 2.30GHz, 16Go Arbeitsspeicher und Samsung SSD 850 500Go) ausgeführt, mit folgenden Softwareversionen:

- MariaDB: mysql Ver 15.1 Distrib 10.1.26-MariaDB, for Linux (x86_64) using readline 5.1
- Java: openjdk version "1.8.0_144"
- Linux: 4.12.11-300.fc26.x86_64

5.1 Erste Testphase

In der ersten Testphase wurde eine zum Testing-Zweck entwickelte Applikation behandelt, welche nur von einer externen Bibliothek abhängt. Die Applikation baut auf keiner Bibliothek auf, die sich in der Datenbank befindet. Diese Applikation ist auf der CD als test1.apk zu finden.

Tabelle 5.1: In erster Testphase verwendete Bibliotheken

Package	Version	Dateienanzahl
com.google.gson	2.4	364
com.google.gson	2.6	374
com.google.gson	2.8.0	386
com.github.com.PhilJay.MPAndroidChart	2.5.9	219
com.github.com.PhilJay.MPAndroidChart	3.0.0	234

Performance

Aus dieser kleinen Applikation wurden 14 Files extrahiert. Da es sich um eine kleine Applikation wurden alle Files analysiert. Die gesamte zur Erkennung gebrauchte Zeit beträgt 10 Sekunden.

Genauigkeit der Erkennung

Das generierte Report sagt Folgendes:

```
App analyzed: /tmp/test1.apk
Number of extracted files: 14
Number of analyzed files: 8
/home/arnold/ApkRay/test1/diff/strazzere/anti/debugger/FindDebugger.java not
 ↪ recognized
/home/arnold/ApkRay/test1/diff/strazzere/anti/taint/FindTaint.java is not recognized
/home/arnold/ApkRay/test1/diff/strazzere/anti/common/Utilities.java not recognized
/home/arnold/ApkRay/test1/org/livefor/mobilehacking/diary/R.java not recognized
/home/arnold/ApkRay/test1/org/livefor/mobilehacking/diary/Pastebin.java is not
 ↪ recognized
/home/arnold/ApkRay/test1/diff/strazzere/anti/emulator/FindEmulator.java not
 ↪ recognized
/home/arnold/ApkRay/test1/diff/strazzere/anti/EmuDetect.java not recognized
/home/arnold/ApkRay/test1/org/livefor/mobilehacking/diary/Diary.java not recognized
```

Es wurden da keine Dateien erkannt und es is gut so, weil sich keine der eingebundenen Bibliotheken in der Datenbank befindet.

5.2 Zweite Testphase

In der zweiten Testphase wurde eine zu diesem Zweck entwickelte Applikation getestet, welche nur von einer externen Bibliothek abhängt (com.google.gson.2.6.1). Diese Applikation wurde nicht obfuscated und ist auf der CD als test2.apk zu finden. Für die zweite Testphase wurde die Datenbank ergänzt, sie enthält nun folgende Packages:

Tabelle 5.2: In zweiter Testphase verwendete Bibliotheken

Package	Version	Dateienanzahl
org.apache.common.io	2.5.1	454
com.google.gson	2.4	364
com.google.gson	2.6	374
com.google.gson	2.8.0	386
org.jmrtd	0.5.0	117
org.jmrtd	0.5.9	89
com.github.com.PhilJay.MPAndroidChart	2.5.9	219
com.github.com.PhilJay.MPAndroidChart	3.0.0	234

Erste Konfiguration

ApkParser ist nun so konfiguriert, dass 20 Files analysiert werden. Die gesamte zur Erkennung gebrauchte Zeit beträgt 14 Sekunden.

Generiertes Report:

```
App analyzed: /tmp/test2.apk
Number of extracted files: 699
Number of analyzed files: 20
/home/arnold/ApkRay/app-debug/com/google/gson/internal/LazilyParsedNumber.java is
 ↪  com.google.gson.2.8.0.LazilyParsedNumber.java with 62
/home/arnold/ApkRay/app-debug/com/google/gson/internal/bind/ReflectiveTypeAdapterFactory.java
 ↪  is com.google.gson.2.8.0.ReflectiveTypeAdapterFactory.java with 97
/home/arnold/ApkRay/app-debug/com/google/gson/internal/bind/SqlDateTypeAdapter.java
 ↪  is com.google.gson.2.4.TimeTypeAdapter.java with 4
/home/arnold/ApkRay/app-debug/com/google/gson/internal/bind/JsonTreeWriter.java not
 ↪  recognized
/home/arnold/ApkRay/app-debug/com/google/gson/stream/JsonReader.java not recognized
/home/arnold/ApkRay/app-debug/com/google/gson/GsonBuilder.java is
 ↪  com.google.gson.2.6.GsonBuilder.java with 115
/home/arnold/ApkRay/app-debug/com/google/gson/internal/ConstructorConstructor.java
 ↪  is com.google.gson.2.8.0.ConstructorConstructor.java with 47
/home/arnold/ApkRay/app-debug/com/google/gson/internal/bind/util/ISO8601Utils.java
 ↪  not recognized
/home/arnold/ApkRay/app-debug/com/google/gson/internal/bind/DateTypeAdapter.java is
 ↪  com.github.PhilJay.2.2.5.LargeValueFormatter.java with 8
/home/arnold/ApkRay/app-debug/com/google/gson/stream/JsonWriter.java not recognized
/home/arnold/ApkRay/app-debug/com/google/gson/internal/bind/TypeAdapterRuntimeTypeWrapper.java
 ↪  is com.google.gson.2.4.TypeAdapterRuntimeTypeWrapper.java with 24
/home/arnold/ApkRay/app-debug/com/google/gson/Gson.java not recognized
/home/arnold/ApkRay/app-debug/com/google/gson/internal/bind/TreeTypeAdapter.java not
 ↪  recognized
/home/arnold/ApkRay/app-debug/com/google/gson/internal/bind/JsonTreeReader.java is
 ↪  com.google.gson.2.8.0.JsonTreeReader.java with 207
/home/arnold/ApkRay/app-debug/com/google/gson/stream/MalformedJsonException.java is
 ↪  com.google.gson.2.4.MalformedJsonException.java with 18
/home/arnold/ApkRay/app-debug/com/google/gson/JsonElement.java is
 ↪  com.google.gson.2.8.0.JsonElement.java with 180
/home/arnold/ApkRay/app-debug/com/google/gson/internal/C$Gson$Preconditions.java is
 ↪  com.google.gson.2.8.0.$Gson$Preconditions.java with 12
/home/arnold/ApkRay/app-debug/com/google/gson/JsonStreamParser.java is
 ↪  com.google.gson.2.8.0.JsonStreamParser.java with 28
/home/arnold/ApkRay/app-debug/com/google/gson/JsonObject.java is
 ↪  com.google.gson.2.8.0.JsonObject.java with 24
/home/arnold/ApkRay/app-debug/com/google/gson/reflect/TypeToken.java is
 ↪  com.google.gson.2.8.0.TypeToken.java with 143
```

In diesem Fall wurden 12 Files erfolgreich erkannt, obwohl die Versionsnummer manchmal vom Ori-

ginal abweichen. Dies ist aber aufgrund des Ähnlichkeitsgrad der aufeinander folgenden Versionen mit dem entwickelten Ansatz kann vermeidbar. Es werden aber manche Files nicht erkannt, obwohl sie aus der Bibliothek com.google.gson stammen, in einer Version die sich in der Datenbank befindet. Eigene Annotationen zum Beispiel bereiten jccd Probleme. Es ist auch so, dass kleine Files oft nicht erkannt werden, weil sie eben aus nur ein paar Zeilen bestehen, die durch den Kompilierungsprozess nicht mehr dem Originalcode entsprechen.

5.3 Dritte Testphase

In dieser letzten Testphase wurde den Code der zweiten Applikation kompiliert und obfuscated. Diese ist auf der CD als test3.apk zu finden:

Tabelle 5.3: In letzter Testphase verwendete Bibliotheken

Package	Version	Dateienanzahl
org.apache.common.io	2.5.1	454
com.google.gson	2.4	364
com.google.gson	2.6	374
com.google.gson	2.8.0	386
org.jmrtd	0.5.0	117
org.jmrtd	0.5.9	89
com.github.com.PhilJay.MPAndroidChart	2.5.9	219
com.github.com.PhilJay.MPAndroidChart	3.0.0	234

Performance

Die gesamte zur Erkennung gebrauchte Zeit beträgt 28,5 Sekunden.

Genauigkeit der Erkennung

Das generiert Report besteht aus Folgendem:

```
1 App analyzed: /tmp/test3.apk
2 Number of extracted files: 354
3 Number of analyzed files: 20
4 /home/arnold/ApkRay/maybe_obfuscated/a/a/a/b/e.java is
    ↪ com.google.gson.2.4.LazilyParsedNumber.java with 52
```

```
5  /home/arnold/ApkRay/maybe_obfuscated/a/a/a/f.java not recognized
6  /home/arnold/ApkRay/maybe_obfuscated/a/a/a/b/a/e.java is
    ↪  com.google.gson.2.4.JsonTreeWriter.java with 112
7  /home/arnold/ApkRay/maybe_obfuscated/a/a/a/b/b.java is
    ↪  com.google.gson.2.6.$Gson$Types.java with 161
8  /home/arnold/ApkRay/maybe_obfuscated/a/a/a/b/c.java not recognized
9  /home/arnold/ApkRay/maybe_obfuscated/a/a/a/i.java not recognized
10 /home/arnold/ApkRay/maybe_obfuscated/a/a/a/b/h.java not recognized
11 /home/arnold/ApkRay/maybe_obfuscated/a/a/a/t.java is
    ↪  com.google.gson.2.8.0.JsonStreamParser.java with 4
12 /home/arnold/ApkRay/maybe_obfuscated/a/a/a/d.java not recognized
13 /home/arnold/ApkRay/maybe_obfuscated/a/a/a/b/a/l.java is
    ↪  com.github.PhilJay.2.2.5.ScatterData.java with 12
14 /home/arnold/ApkRay/maybe_obfuscated/a/a/a/b/a/f.java not recognized
15 /home/arnold/ApkRay/maybe_obfuscated/a/a/a/d/a.java not recognized
16 /home/arnold/ApkRay/maybe_obfuscated/a/a/a/k.java is
    ↪  com.google.gson.2.4.TypeAdapter.java with 22
17 /home/arnold/ApkRay/maybe_obfuscated/a/a/a/b/a/i.java is
    ↪  com.github.PhilJay.2.2.5.ScatterData.java with 12
18 /home/arnold/ApkRay/maybe_obfuscated/a/a/a/b/a/m.java not recognized
19 /home/arnold/ApkRay/maybe_obfuscated/fhstp/ac/at/gson_2_8_test_no_obfuscation/MainActivity.java
    ↪  not recognized
20 /home/arnold/ApkRay/maybe_obfuscated/a/a/a/p.java not recognized
21 /home/arnold/ApkRay/maybe_obfuscated/a/a/a/c/a.java is org.jmrtd.0.5.9.DG15File.java
    ↪  with 18
22 /home/arnold/ApkRay/maybe_obfuscated/a/a/a/b/a/h.java not recognized
23 /home/arnold/ApkRay/maybe_obfuscated/a/a/a/b/a/c.java not recognized
```

Diese Testphase zeit entauschende Ergebnisse, es wurden ein Viertel (5 aus 20 Files) aller analysierten Files richtig erkannt, rund ein Fünftel (3 aus 20 Files) sind False-Positives und eine große Hälfte wird einfach nicht erkannt (11 aus 20 Files).
Im Laufe der Testphase wurden zusätzliche Optimierungen eingeführt, welche in vorherigen Kapiteln erläutern wurden:

- Berechnung der Tiefe der Ordnerstruktur
- der Ausschluss aller aus der Android-Support Bibliotheken kommenden Dateien
- die Anpassung der angewendeten Operatoren

Diese Tests zeigen unter anderem, dass kleine Dateien und Files, in denen moderne Java-Instruktionen verwendet werden, schlecht beziehungsweise gar nicht erkannt werden.

6 Verfügbarkeit und Anwendbarkeit

Das Programm erfordert eine JVM und eine Verbindung zu einer SQL-Datenbank. Die SQL-Datenbank ist einer der Kernkomponente des Konzepts. Es muss im Vorhinein keine besondere Konfiguration erfolgen. Falls die benötigte Tabelle zur Laufzeit nicht vorhanden ist, wird sie von LibParser erstellt. An dieser Stelle sei nochmal erwähnt, dass LibParser und ApkParser ausschließlich mit MariaDB getestet wurden. Was die JVM betrifft, wurde immer mit OpenJDK gearbeitet, in Version 1.8. Jedoch werden keine OpenJDK spezifische Funktionen angewendet und der Code ist portabel konzipiert, was zum Beispiel die Pfade-Behandlung betrifft.

6.1 Einrichtung der SQL-Datenbank

Doch vor der Ausführung des Programms müssen alle Informationen zur Herstellung einer Verbindung mit dem SQL-Server im File at.ac.fhstp.apkray.libparser.Settings.java bekanntgegeben werden:

```
1 public class Settings {
2
3     public static final String DB_USERNAME = "benutzername";
4     public static final String DB_PASSWORD = "passwort";
5     public static final String DB_IP_ADDRESS = "localhost";
6     public static final String DB_NAME = "apkray";
7 }
```

6.2 Konfiguration von LibParser

Damit Pfad-bezogene Probleme vermieden werden, müssen sich in diesen Feldern keine Anführungszeichen befinden. Folgend steht ein Beispiel einer gut eingerichteten Bibliothek:

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <libraries>
3    <library>
4       <package>org.apache.common.io</package>
5       <version>2.5.1</version>
```

```
6         <source>/home/username/Libraries/commons-io-2.5-src.zip</source>
7     </library>
8 </libraries>
```

Das Version-Tag darf Nummern und Zeichen beinhalten, wie zum Beispiel 1.3-rc1 oder 0.1-beta2. Wenn dieses File alle Bibliotheken beschreibt, kann folgend das Programm wie folgt aufgerufen werden:

```
1 java -jar libparser.jar /path/to/db.xml
```

So sieht dann der Importprozess aus:

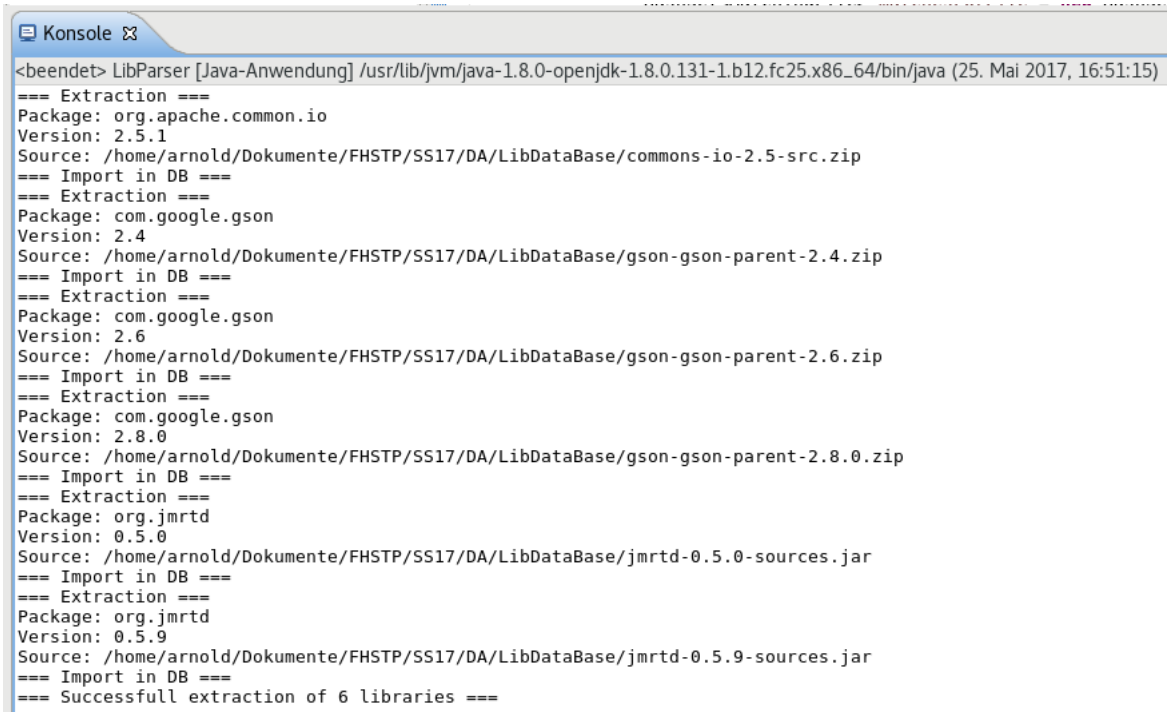


Abbildung 6.1: Importprozess einiger Bibliotheken

6.3 Konfiguration von ApkParser

Wenn die SQL-Datenbank erfolgreich eingerichtet und gefüllt wurde, kann mit dem tatsächlichen Analyseverfahren begonnen werden.

Das Verzeichnis, wo der extrahierte Code am Ende des Extrahierens liegen soll, ist konfigurierbar. Dies kann dem Programm bei der Ausführung mitgeteilt werden, indem ein zweiter Wert einfach übergeben wird:

```
1 java -jar apkparser.jar myapp.apk /my/tmp/dir
```

Dies muss jedoch ein beschreibbares Verzeichnis sein, weil in dieses sowohl das LogFile als auch das Report hingelegt werden. Dieser Wert ist jedoch optional. Das standard Verzeichnis für den User wolfgang und die applikation myapp.apk lautet unter GNU/Linux:

```
1 /home/wolfgang/ApkRay/myapp/
```

7 Schlußfolgerung

Diese Diplomarbeit hat sich die Frage gestellt, wie bekannte Bibliotheken in fertigen Applikationen erkannt werden können. Zu diesem Zweck wurde an erste Stelle recherchiert, wie Android-Applikationen und Bibliotheken aufgebaut werden. Es wurde dann entschieden, nur mit in Java geschriebenen Applikationen zu arbeitet.

Es wurde dann ein Konzept entworfen, welches dann in Java implementiert und getestet wurde. Das entworfene Konzept ist eine Kombination aus zwei Ansätzen. Es wurde zuerst ein statistischer Prozess entwickelt, welcher einige Files für die zweite Stufe des Erkennungsverfahrens auswählt. Nach dieser ersten Etappe werden dann Files eins zu eins verglichen, hier findet *Code-Cloning-Detection* statt.

Die Erkennungsmethode wurde dann getestet, mit verschiedenen Applikationen und unterschiedlichen Konfigurationen. Die Ergebnisse zeigen, dass die entwickelte Methode grundsätzlich funktioniert. Jedoch wurde eine gute Erkennung nicht erzielt. Es sind zwei verschiedene Probleme aufgetaucht. Das erste Problem ist jenes, das für kurze Files berechnete Statistiken nicht aussagekräftig genug sind. Das zweite Problem betrifft das *Code-Cloning-Detection*, kleine Java-Dateien und vor allem Klassen mit einer komplizierten Syntax bereiten Probleme. Android-Applikationen werden in moderner Java geschrieben, jedoch bereiten Java-Neuerungen jccd viele Probleme, weil dieses Programm seit 2011 nicht mehr entwickelt wird. In der Test-Phase wurde klar, dass zwei sehr ähnliche Dateien dadurch jedoch als verschieden erkannt wurden. Außerdem hat der in dieser Arbeite angewendete Dekompiler, Jadx, manchmal Schwierigkeiten mit bestimmten Files, an dieser Stelle besteht noch Verbesserungsbedarf.

In dieser Arbeit wurde ein Lösungsansatz konzipiert, dessen Implementierung unbefriedigende Ergebnisse liefert. Es bestehen jedoch Verbesserungsmöglichkeiten, indem Komponente, auf denen die Implementierung aufbaut, durch leistungsfähigere Programme ausgetauscht werden.

Abbildungsverzeichnis

1.1 Android-Fragmentierung 1

2.1 Generierung des classes.dex-Files 9
2.2 Generieung des APK-Files 9
2.3 Erzeugung eines neuen Key 10
2.4 Signieren des APK-Files 10
2.5 Kompilierte Applikation ohne Obfuskation in jadx-gui geöffnet 17
2.6 Kompilierte Applikation mit Obfuskation in jadx-gui geöffnet 17

3.1 Abbildung des gesamten Erkennungsverfahrens 25
3.2 Schema der SQL-Datenbank 26

4.1 UML-Diagramm des Core-Packages 38
4.2 UML-Diagramm des LibParser-Packages 39
4.3 UML-Diagramm des ApkParser-Packages 39

6.1 Importprozess einiger Bibliotheken 71

Tabellenverzeichnis

2.1 10 populärste Android-Bibliotheken 7

3.1 Statistiken aus einem Java-File 23
3.2 Analyse des AIODJ.java-Files 24
3.3 Abbildung einer fiktiven Matching-Tabelle 28
3.4 Typisches Double-Match Beispiel 29
3.5 Berechnete Merkmale für das File ABC.java 34
3.6 Ergebnisse von Code-Cloning-Detection 34

4.1 Auflistung aller gesuchten Elemente 44
4.2 Ergebnisse von Code-Cloning-Detection 61

5.1 In erster Testphase verwendete Bibliotheken 65
5.2 In zweiter Testphase verwendete Bibliotheken 66
5.3 In letzter Testphase verwendete Bibliotheken 68

Literaturverzeichnis

[1] M. A. Bari and D. S. Ahamad, “Code Cloning: The Analysis, Detection and Removal ,” http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.206.3588&rep=rep1&type=pdf, 2011.

[2] Roy, Chanchal K., James R. Cordy, and Rainer Koschke, “Comparison and evaluation of code clone detection techniques and tools: A qualitative approach.” Science of Computer Programming 74.7 (2009): 470-495.

[3] D. Griffiths, *Head First Android Development: A Brain-Friendly Guide*. Sebastopol, California: O’Reilly Media, 2015.

[4] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, “A study of android application security,” https://www.usenix.org/legacy/event/sec11/tech/slides/enck.pdf, 2011.

[5] P. O. Fora, “Beginners guide to reverse engineering android apps,” https://www.rsaconference.com/writable/presentations/file_upload/stu-w02b-beginners-guide-to-reverse-engineering-android-apps.pdf, 2014.

[6] A. Desnos and G. Gueguen, “Android: From reversing to decompilation,” http://media.blackhat.com/bh-ad-11/Desnos/bh-ad-11-DesnosGueguen-Andriod-Reversing_to_Decompilation_WP.pdf, 2011.

[7] Google, “Android security white paper,” https://static.googleusercontent.com/media/enterprise.google.com/en//android/static/files/android-for-work-security-white-paper.pdf, 2016.

[8] L. Ndwaru, “Introduction to android art; the next generation of android runtime.” https://www.researchgate.net/publication/313242785_Introduction_to_Android_ART_The_next_generation_of_Android_Runtime, 2014.

[9] J. Wu and M. Yang, “Lachouti: kernel vulnerability responding framework for the fragmented android devices,” https://www.researchgate.net/publication/318871337_LaChouTi_kernel_vulnerability_responding_framework_for_the_fragmented_Android_devices, 2017.

[10] D.-H. You and B.-N. Noh, "Android platform based linux kernel rootkit," http://ieeexplore.ieee.org/document/6112330/, 2011.

[11] "The Java Language Environment A White Paper," http://www.inf.ufsc.br/ rosvelter.costa/java/docs/java-whitepaper.pdf, 1995.

[12] N. Karumanchi, *Data Structures and Algorithms Made Easy in Java: Data Structure and Algorithmic Puzzles*. Careermonk Publications, 2011.

[13] C. Ullenboom, *Java ist auch eine Insel*. Rheinwerk Computing, 2017.

[14] Cullen Linn Saumya Debray, "Obfuscation of executable code to improve resistance to disassembly," https://www2.cs.arizona.edu/solar/papers/CCS2003.pdf, 2004.

[15] Y. G. Ziang Ma, Haoyu Wang and X. Chen, "LibRadar: Fast and Accurate Detection of Third-party Libraries in Android Apps," http://sei.pku.edu.cn/ maziang14/papers/Ma-ICSE-16.pdf, 2015.

[16] A. Kulkarni and R. Metta, "A new code obfuscation scheme for software protection," http://ieeexplore.ieee.org/document/6830939/, 2014.

[17] C. Collberg and J. Nagra, *Surreptitious Software, obfuscation, watermarking and tamperproofing for software protection*. Addison-Wesley, 2010.

[18] P. Schulz, "Code protection in android," http://net.cs.uni-bonn.de/fileadmin/user_upload/plohmann/2012-Schulz-Code_Protection_in_Android.pdf, 2012.

[19] M. Chilowicz, E. Duris, and G. Roussel, "Syntax tree fingerprinting for source code similarity detection," http://ieeexplore.ieee.org/document/5090050, 2009.

[20] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," https://dl.acm.org/citation.cfm?id=2635900, 2014.

[21] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," http://ieeexplore.ieee.org/document/4288192/, 2007.

[22] V. Juričić, "Detecting source code similarity using low-level languages," http://ieeexplore.ieee.org/document/5974090/, 2011.

[23] K. Raheja and R. K. Tekchandani, "An efficient code clone detection model on java byte code using hybrid approach," http://ieeexplore.ieee.org/document/6832302/, 2013.

[24] D. Yu, J. Wang, Q. Wu, J. Yang, J. Wang, W. Yang, and W. Yan, "Detecting java code clones with multi-granularities based on bytecode," http://ieeexplore.ieee.org/document/8029624/, 2017.

[25] T. Sager, A. Bernstein, M. Pinzger, and C. Kiefer, "Detecting similar java classes using tree algorithms," https://dl.acm.org/citation.cfm?id=1138000, 2006.

[26] T. Mende and R. K. F. Beckwermert, "An evaluation of code similarity identification for the grow-and-prune model," https://dl.acm.org/citation.cfm?id=1138000, 2009.

[27] M. Linares-VásquezCollege, A. Holtzhauer, C. Bernal-Cárdenas, and D. Poshyvanyk, "Revisiting android reuse studies in the context of code obfuscation and library usages," https://dl.acm.org/citation.cfm?id=1138000, 2014.