Philipps Universität Marburg

FACHBEREICH MATHEMATIK UND INFORMATIK
Arbeitsgruppe Verteilte Systeme

Bachelorarbeit

Parallele Analyse von Android-Anwendungen

Nils Tobias Schmidt
14. August 2014

Gutachter
Prof. Dr. Bernd Freisleben

Philipps-Universität Marburg
Fachbereich Mathematik und Informatik
Hans-Meerwein-Straße
35032 Marburg

Inhaltsverzeichnis

1 Einleitung

Die Benutzung von mobilen Geräten wie Handys und Tablets nimmt immer weiter zu. Im Vergleich zu Computern speichern mobile Geräte wesentlich mehr sensible Daten. Dazu gehören die persönlichen Kontakte, die im Adressbuch zusammen mit der Telefonnummer und ggf. weiteren Metainformationen abgelegt werden. Es können Telefonate geführt und Nachrichten durch Senden von SMS, E-Mail oder der Benutzung von Programmen wie *WhatsApp* ausgetauscht werden. Außerdem sind diverse Sensoren verfügbar, mit denen neben Audio- auch Videodaten aufgezeichnet und der Standort bestimmt werden kann.

Durch die Speicherung und Verfügbarkeit von mehr sensiblen Daten, ergeben sich auch neue Herausforderungen. Die Daten müssen geschützt und Applikationen an der Weitergabe der Daten gehindert werden. Boshafte Anwendungen (*Malware*) können z.B. finanziellen Schaden anrichten, weshalb Werkzeuge zur Aufspürung notwendig sind.

Laut Gartner, dominiert *Android* mit einem Anteil von 78,4% den Markt mobiler Betriebssysteme. Seit der Einführung von *Android* im Jahr 2008, wurden 900 Millionen *Android*-Geräte aktiviert. Täglich kommen 1,5 Millionen Neue hinzu[1].

Gleichzeitig mit der steigenden Popularität mobiler Geräte, stellt *Sophos* jedoch ein exponentielles Wachstum an *Android-Malware* fest.

Die Funktionalität von mobilen Geräten lässt sich durch Applikationen, die in *App Stores* verfügbar sind, leicht erweitern. Im Gegensatz zu *iOS*, können für *Android* Apps aus beliebigen Quellen installiert werden. Bekannte und zum Teil kostenpflichtige Applikationen werden mit Schadcode versehen und kostenlos in alternativen *App Stores* veröffentlicht.

Die Gefahren, die von *Malware* ausgehen, sind vielfältig und reichen von Diebstahl sensibler Daten, Mitschneiden von Kommunikation über Audio- und Videoquellen bis hin zu finanziellen Schäden. Nicht zuletzt können Opfer selbst Teil krimineller Aktivitäten werden, indem sie Teil eines *Botnetzes* werden.

Bekannte Gefahren sind z.B. das Senden von SMS an Premium-Nummern oder der Sperrung des Gerätes, die nur gegen Bezahlung wieder aufgehoben wird. *Android-*

[1] Die Android-Daten beziehen sich auf das zweite Quartal im Jahr 2013: `http://www.androidcentral.com/larry-page-15-million-android-devices-activated-every-day`. Eingesehen am 30.07.2014

Defender ist ein Beispiel solcher *ransomware*, welches sich als Antiviren-Software ausgibt. Zusätzlich zur Sperrung des Gerätes ist auch eine Verschlüsselung der Daten üblich.

1.1 Problemstellung

Zur Erkennung von *Malware*, sodass diese erst gar keinen Schaden anrichten kann, werden Werkzeuge benötigt, um Schadcode abzuwenden. Die Herausforderung ist umso größer, da *Malware* immer intelligenter wird und dem Auffinden durch Antiviren-Software entgegensteuert[2] [Sva14].
Es werden zwar Berechtigungen für den Zugriff auf sensible Daten benötigt, die der Nutzer bei der Installation erteilen muss, jedoch kann sich der Benutzer nicht sicher sein, ob diese weitergegeben werden (Datenleck).
Es existieren eine Menge von Programmen, die versuchen boshaftes Verhalten aufzudecken, jedoch skalieren diese nicht. Da allerdings mittlerweile mehr als eine Millionen Apps alleine in dem *App Store* von Google (*Play Store*) existieren [Mas], ist dies zwingend erforderlich, wenn man eine große Anzahl an Applikationen analysieren möchte.
Das, im Rahmen dieser Bachelor-Arbeit, entwickelte Programm AndroLyzeLab (ALL) stellt ein Framework zur Verfügung, um skalierbare Analysetools für *Android*-Applikationen zu implementieren. Analysen können durch ein flexibles und mächtiges Skript-Framework basierend auf der Funktionalität des bekannten Reverse-Engineering-Tool *androguard* implementiert werden. Dafür wird ein einfaches und strukturiertes Loggen der Ergebnisse zur Verfügung gestellt, sodass diese automatisiert in einer Datenbank zur späteren Analyse abgelegt werden können.
Die Arbeit ist wie folgt gegliedert: In Kapitel zwei wird zunächst das zum Verständnis dieser Arbeit benötigte Hintergrundwissen vermittelt. Anschließend wird in Kapitel drei auf verwandte Arbeiten in dem Bereich eingegangen. Danach wird das Design von ALL sowie Auszüge aus dessen Implementierung vorgestellt (respektive Kapitel vier und fünf). Zum Schluss wird die Arbeit evaluiert und eine Zusammenfassung sowie eine Aussicht auf zukünftige Arbeiten durch Verbesserungen und neue Features präsentiert.

[2] Dazu gehört z.B. Obfuscation und Mutation von Code ohne Änderung der Semantik (mittels polymorpher Techniken)

2 Grundlagen

Das, im Rahmen dieser Bachelorarbeit, entwickelte Programm ALL, hilft dem Analysten beim Untersuchen und Analysieren von Android-Anwendungen. Es bietet dafür ein Scripting-Framework, mithilfe dessen beliebige Eigenschaften untersucht werden können.
Im Folgenden wird die Android-Platform im Detail erläutert, um den Leser mit allen Informationen zu versorgen, die er für das Verständnis dieser Arbeit benötigt. Dafür wird auf grundlegende Techniken der App-Analyse eingegangen und das Hintergrundwissen für paralleles und verteiltes Rechnen vermittelt. Zusätzlich wird auf die benutzten Frameworks eingegangen, die zur Implementierung von ALL genutzt wurden.

2.1 Android

Android ist ein noch sehr junges Betriebssystem für mobile Geräte wie Handys, Tablets und mittlerweile auch Netbooks.
Die Platform wurde von Android Inc. gegründet, 2007 von Google übernommen und als das Android Open Source Project veröffentlicht. Entwickelt und verteilt wird es durch die Open Handset Alliance (OHA), die aus einer Gruppe von 78 verschiedenen Firmen besteht.
Der Code kann kostenlos über ein zentrales Repository bezogen und unter den bestehenden Lizenzbedingungen modifiziert werden. Dabei handelt es sich zum größten Teil um die Apache-, sowie die BSD-Lizenz.
Zum Zeitpunkt dieser Thesis ist Android 4.4 mit dem Namen *KitKat* die aktuelle Version des mobilen Betriebssystems [andb, andi, Bra10].

2.1.1 Grundlagen

Android ist ein Multiuser-System basierend auf dem Linux Kernel. Apps werden in der Programmiersprache Java geschrieben, durch die *Android SDK Tools* kompiliert und zusammen mit anderen Daten und Ressourcen in einem APK bereitgestellt. Auf den Aufbau eines APK wird in Kapitel 2.1 eingegangen.
Android implementiert das *least-privilege*-Prinzip. Das bedeutet, dass einer Applikation nur soviele Rechte zur Verfügung gestellt werden, wie sie auch wirklich benötigt.

Erreicht wird dies, indem jeder App ein eindeutiger *user identifier* sowie *group identifier* zugewiesen wird und alle Dateien der Applikation an diesen Benutzer bzw. diese Gruppe gebunden werden. Somit kann keine andere App auf applikationsfremde Dateien zugreifen.
Weitere Isolation wird durch Ausführung einer Android-Anwendung in jeweils einem eigenen *Linux-Prozess* sowie einer eigenen virtuellen Maschine erreicht. Der sogenannten Dalvik Virtual Machine (DVM), die in Kapitel 2.1.5 behandelt wird.
Für den Fall, dass Applikationen jedoch gemeinsamen Zugriff auf Dateien benötigen, können sie sich denselben *user identifier* teilen. Außerdem ist es möglich, zwei Anwendungen in dem gleichen *Prozess* sowie der gleichen DVM auszuführen. Dafür müssen beide jedoch mit dem gleichen Zertifikat signiert worden sein.
Eine Applikation wird also isoliert in einer *Sandbox* ausgeführt [andf].

2.1.2 Components

Die Grundbausteine einer App sind Komponenten. Unterschieden wird hier zwischen vier Typen: *Activities*, *Broadcast Receivers*, *Services* sowie *Content Providers*. Während *Activities* die Aufgabe der grafischen Oberfläche zugeschrieben wird, sind *Broadcast Receivers* und *Services* für Hintergrundaktivitäten zuständig. Letztere für einen längeren Zeitraum. Bedingt durch die Isolation der Applikationen, fehlen die Berechtigungen andere Apps zu starten. Somit ist ein Umweg über das Android-System notwendig, was mithilfe von *Intents* (asynchronen Nachrichtenobjekten) erreicht wird. Sie regeln also die Kommunikation innerhalb der Komponenten, indem sie diese starten und optional ein Ergebnis zurückliefern können [andf].

2.1.2.1 Activity

Eine *Activity* ist die grafische Hauptkomponente. Sie umfasst ein einzelnes Fenster des User Interface (UI), bündelt visuelle Elemente und ermöglicht Benutzerinteraktionen. Eine App kann aus mehreren *Activities* bestehen. Kommunizieren können diese durch *Intents* [Bra10], auf die später in Kapitel 2.1.3.2 näher eingegangen wird.

2.1.2.2 Content Provider

Im Gegensatz zu *Activities* haben *Content Provider* keine grafische Oberfläche. Sie dienen dem Zweck der Datenbereitstellung. Genutzt werden kann z.B. das Dateisystem oder eine *SQLite-Datenbank* [andf].
Somit können Daten auch zwischen Applikationen ausgetauscht werden. Dabei greift eine App jedoch nicht direkt auf einen *Content Provider* zu, sondern über einen *Content Resolver* [Bra10]. Mittels eines Uniform Resource Identifier (URI), wird der *Content Provider* sowie die weiteren Filterkriterien festgelegt. Damit eine andere Applikation auf den *Content Provider* zugreifen kann, muss sie jedoch die erforderlichen Berechtigungen besitzen [andg].

2.1.2.3 Broadcast Receiver

Für den Nachrichtenaustausch über Applikationsgrenzen hinweg, gibt es *Broadcast Receiver*. Sie reagieren auf spezielle *Events* und führen eine vordefinierte Aktion aus. Dafür kann auf Systemevents zurückgegriffen oder eigene Events definiert werden. Sie dienen jedoch nur als Vermittlungspunkt und sollten nicht für zeitintensive oder

länger andauernde Aufgaben benutzt werden. Für diesen Zweck kann ein *Service* gestartet werden [andf].

2.1.2.4 Service

Im Gegensatz zu einem *Broadcast Receiver* läuft ein *Service* zwar auch im Hintergrund, dient jedoch dem Ausführen von länger laufenden Aufgaben wie z.B. dem Abspielen von Musik, ohne dass die Applikation noch im Vordergrund sein muss. Applikationen können *Services* sowohl selbst starten, als auch mit ihnen interagieren [andf].

2.1.3 Application Package File (APK)

Das gezippte APK umfasst alle Ressourcen, die das Android-System zum Ausführen der Applikation benötigt und ist in Abbildung 2.1 dargestellt. Darunter fallen der ausführbare Code, essentielle Informationen über die Applikation sowie andere Ressourcen wie Grafiken und Layout-Dateien [andf].
Die ersten beiden Elemente befinden sich respektive in *classes.dex* und in *AndroidManifest.xml* [Nol12]. Letzteres ist im Ordner *res* abgespeichert, der die Ressourcen des APK, Layout-Files in Extensible Markup Language (XML) sowie Bilder (verschiedene *drawable-* Ordner) und *assets* (Ablage für sonstige Dateien) umfasst [andc].
Signatur- und Integritätsverbundene Dateien liegen in *META-INF*. Hier ist u.A. das Zertifikat zu finden mit dem die Dateien signiert wurden (*cert.rsa*) sowie eine Liste mit Hashes aller Dateien (*MANIFEST.MF*) [Nol12].

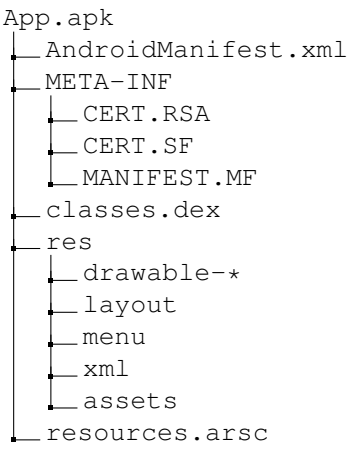

Abbildung 2.1: Ausschnitt APK Struktur

2.1.3.1 Manifest

Die Manifest-Datei stellt dem System wichtige Informationen bereit und muss im Root-Verzeichnis mit dem Dateinamen *AndroidManifest.xml* liegen [ande]. Gespeichert ist sie als binäres XML [Nol12].
Eine XML-Datei definiert genau das Layout und damit aus welchen Elementen sie bestehen darf. Es handelt sich wie bei *HTML* um eine Auszeichnungssprache, jedoch mit Flexibilität in Bezug auf das Aussehen des Dokumentes hinsichtlich der strukturellen Elemente [TvS08].
Android bietet für die Strukturierung 23 vordefinierte Elemente an. Ein Ausschnitt ist in Abbildung 2.2 zu sehen.

```xml
<?xml version="1.0" encoding="utf-8"?>

<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="string"
          android:sharedUserId="string"
          android:sharedUserLabel="string resource"
          android:versionCode="integer"
          android:versionName="string"
          android:installLocation=["auto" | "internalOnly" | "
              preferExternal"] >

    <uses-permission />
    <permission />
    <uses-sdk />
    <uses-feature />

    <application>

        <activity>
            <intent-filter>
                <action />
                <category />
                <data />
            </intent-filter>
            <meta-data />
        </activity>

        <service>...</service>

        <receiver>...</receiver>

        <provider>...</provider>

        <uses-library />

    </application>

</manifest>
```

Abbildung 2.2: Modifizierte Struktur des AndroidManifest.xml [ande]

Beschrieben wird die App durch einen eindeutigen *identifier* (den *fully-qualified Java package name*) sowie einen String, der die Versionsnummer nach außen definiert. Beide sind in dem <*manifest*>-Element zu finden, dessen Struktur in Abbildung 2.2 angedeutet ist. Ersteres unter dem Tag *package* und letzteres unter dem Tag *android:versionName*.

Von großer Bedeutung sind im Kontext der Sicherheit die Berechtigungen der App. Hierbei muss zwischen Berechtigungen unterschieden werden, die die Applikation für den Zugriff auf API-Funktionen benötigt und zwischen Berechtigungen, die andere Anwendungen benötigen, um auf die internen Komponenten zuzugreifen.

So werden z.B. mit dem <*uses-permission*>-Element die für die Ausführung der App benötigten Rechte definiert [ande], welche bei der Installation vom Benutzer angefordert werden [Gun12].

Eine Anwendung kann seine Komponenten mit dem <*permission*>-Element vor unauthorisierter Benutzung schützen. Dabei können entweder die Android-Berechtigungen benutzt oder eigene definiert werden. Somit muss für die Mitbenutzung der Komponente erst das jeweilige Recht angefordert werden.

Außerdem definiert ist der minimale API-Level, die genutzten Bibliotheken gegen die gelinkt werden muss, sowie eine Reihe von *Intent-Filtern*, die die Fähigkeiten der Komponenten nach außen repräsentieren und Anforderungen der App wie z.B. das Vorhandensein einer Kamera [ande].

2.1.3.2 Intents

```
1 <activity android:name="MainActivity">
2     <intent-filter>
3         <action android:name="android.intent.action.MAIN" />
4         <category android:name="android.intent.category.LAUNCHER" />
5     </intent-filter>
6 </activity>
```

Abbildung 2.3: Intent-Filter [andh]

Applikationen kommunizieren indem Sie *Intents* senden. Diese Nachrichtenobjekte können benutzt werden, um beispielsweise eine *Activity* wie die von Android gebotene Kamera-App zu starten und über deren Funktionalität eine Bildaufnahme zurückzuliefern. *Intents* sind asynchron, damit der aufrufende *Prozess* nicht auf das Ergebnis warten muss und somit nicht blockiert wird.
Unterschieden werden zwei Arten von *Intents*: *explizite* und *implizite Intents*. Ersteres wird für das interne Aufrufen von *Komponenten* über den vollständigen Java-Paketnamen benutzt. Die Komponente muss also genau benannt werden. Allerdings gibt es noch die Möglichkeit über *Intent-Filter* in dem Manifest die Fähigkeiten der einzelnen Komponenten zu beschreiben. Dies wird durch eine Beschreibung aus Aktionen und Kategorien erreicht. Ein Beispiel für *implizite Intents* kann in Abbildung 2.3 nachvollzogen werden. Durch das Abgleichen der Filter im *Manifest* kann das Android-System alle *Komponenten* heraussuchen, die auf den *Intent* passen. Gibt es mehrere, so wird dem Benutzer über einen Dialog eine Auswahlmöglichkeit geboten [andh].

2.1.4 System Architektur

Das Android-System besteht aus vier Schichten: der Kernel-, Laufzeit-, Framework- und Applikationsschicht. Auf der untersten Ebene befindet sich der Kernel. Er interagiert als Abstraktionsschicht zwischen Soft- und Hardware. Dabei ist er für grundlegende Aufgaben wie Prozess- und Speichermanagement sowie Netzwerk, Geräteverwaltung und Dateisystemfunktionalität zuständig [Gun12] .
Insbesondere ist er für Energie- und Speichermanagement angepasst [Bra10].
Darüber befindet sich die Laufzeitschicht. Sie enthält Bibliotheken, die aus Performanzgründen in nativem Code (C/C++) implementiert sind, sowie die Laufzeitumgebung, die aus der DVM und einigen Hauptbibliotheken besteht. Auf der Ebene befindet sich z.B. das *MediaFramework*, das aus der Anwendungsschicht herausgezogen wurde, um eine bessere Leistung zu erzielen sowie die Datenbank *SQLite* und Grafikbibliotheken für 2D- und 3D-Grafik.
Das Herzstück für App-Entwickler ist das *Application Framework*. Es bildet das Application Programming Interface (API) und ist in Java geschrieben. Als Abstraktionsschicht bietet es Zugriff auf die darunterliegenden Bibliotheken.
Auf der letzten Ebene befinden sich die System- sowie Drittanbieter-Apps, folglich die dem Endbenutzer bereitgestellten Applikationen [Gun12].

2.1.5 Dalvik Virtual Machine (DVM)

Obwohl Android-Applikationen in Java geschrieben sind, werden sie nicht in der Java Virtual Machine (JVM) sondern der DVM ausgeführt. Diese wurde auf der einen Seite aus Lizenzgründen, hauptsächlich aber aus dem Grund, dass mobile Geräte durch limitierte Ressourcen spezielle Anforderungen stellen, eingeführt.

Sie haben weniger Rechenleistung, einen kleineren Hauptspeicher und keinen Swap-Speicherplatz. Außerdem laufen sie auf Batterie und verfügen somit nur über eine beschränkte Laufzeit. Die DVM wurde speziell auf diese Anforderungen zugeschnitten [Gun12].
Wie in Abbildung 2.4 zu sehen ist, wird mit dem normalen *Java-Compiler* *Java-Bytecode* erzeugt, der anschließend weiter in *Dalvik-Bytecode* übersetzt wird. Außerdem werden einige Optimierungen vorgenommen, wie z.B. das Entfernen von redundanten Daten, indem diese mit Zeigern simuliert werden. Als Ergebnis erhält man einen wesentlich kompakteren und kleineren *Bytecode*, welcher in der Datei *classes.dex* im APK gespeichert ist. Die DVM ist also der *Interpreter* für den *Dalvik-Bytecode* [Bra10].

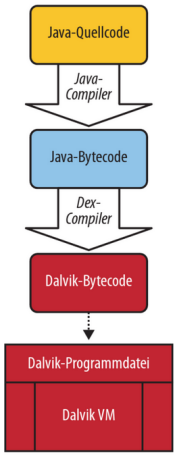

Abbildung 2.4: Erzeugen von Dalvik-Bytecode [Gar11]

2.2 Analyse

Bei der Analyse von Programmen, egal ob es sich dabei um mobile oder Desktop-Applikationen handelt, unterscheidet man zwischen zwei Analyseverfahren. Zum einen die *statische Analyse*: Hier wird das *binary* disassembliert oder dekompiliert und basierend auf dem statischen Quellcode nach bestimmten Mustern gesucht.
Zum anderen gibt es die *dynamische Analyse*: Hier wird die Applikation in einer isolierten und kontrollierten Umgebung ausgeführt, wodurch das Verhalten zur Laufzeit untersucht werden kann. Der Vorteil gegenüber der *statischen Analyse* liegt in der Immunität gegenüber *code obfuscation* [DPLR]. Dabei handelt es sich um eine Technik, bei der der Quellcode unleserlich gemacht wird, indem Klassen-, Variablen- sowie Methodennamen umbenannt werden, sodass deren Namen keine inhaltliche Bedeutung mehr suggerieren. Die Analyse wird somit deutlich erschwert [Gun12].

2.3 Paralleles und verteiltes Rechnen

In diesem Abschnitt soll dem Leser ein Überblick über die generelle Funktionsweise von *verteilten Systemen* sowie parallelem Rechnen vermittelt werden.

2.3.1 Paralleles Rechnen

Zum Verständnis von parallelem Rechnen muss zunächst der Unterschied zwischen wichtigen Begriffen wie *Threads* und *Prozessen* verdeutlicht, sowie erklärt werden, wie diese untereinander kommunizieren, um gemeinsam größere Aufgaben schneller zu erledigen.

2.3.1.1 Multiprogrammierung und Prozesse

Parallelität bzw. Quasiparallelität ist ein wichtiges Konzept moderner Betriebssysteme. Während ersteres echte Parallelität durch mehrere Prozessoren oder Prozessorkerne bietet, hat der Benutzer bei letzterem nur das Gefühl von Nebenläufigkeit (*Multiprogrammierung*). Für das weitere Verständnis von Parallelität, muss zunächst das Konzept eines *Prozesses* eingeführt werden, mithilfe dessen Nebenläufigkeit model-

liert bzw. implementiert wird. Es handelt sich dabei um eine abstrakte Beschreibung eines Programmes in Ausführung. Ein *Prozess* umfasst alle Informationen, die das Betriebssystem benötigt, um ein Programm laufen zu lassen. Dazu gehört der eigentliche Programmcode, die Daten des Programms und andere Ressourcen wie z.B. eine Liste von offenen Dateien.
Damit Programme parallel laufen können, werden diese entweder mehreren Prozessoren bzw. Prozessorkernen zugeteilt oder für den Fall eines Einprozessorsystems schnell nacheinander ausgeführt.
Diese Quasiparallelität wird erreicht, indem den einzelnen *Prozessen* Zeitscheiben zugeteilt werden, mithilfe derer der *Scheduler* die *Prozesse* für eine gewisse Zeit laufen lässt. Jedoch können sie auch während sie ausgeführt werden durch *Interrupts* unterbrochen werden, damit z.B. Ereignisse der Tastatur oder Festplatte abgearbeitet werden können.
Ein *Prozess* kann sich in verschiedenen Zuständen befinden. So kann er z.B. *running*, *ready* oder *blocked* sein. *running* ist er, wenn er aktuell einem Prozessor zur Ausführung zugeteilt ist. Allerdings kann er bedingt durch seine Aufgabe auch blockiert sein, weil er z.B. auf I/O warten muss. Ein *Prozess*, der nicht am arbeiten ist, aber dafür bereit wäre, befindet sich im Zustand *ready* [Tan09].
Beim Hin- und Herschalten zwischen den *Prozessen*, dem sogenannten *context switch*, werden sie vom *Scheduler* in Warteschlangen je nach Ausführungszustand eingereiht. Er legt also die Ausführungsreihenfolge der *Prozesse* fest. Je nach Betriebssystem weichen die verwendeten Zustände sowie die Scheduling-Strategien voneinander ab. Das Überführen des *Prozesses* in einen anderen Zustand wird vom *Dispatcher* ausgeführt [SSP06].
Damit ein *Prozess* an der gleichen Stelle fortfahren kann, an der er unterbrochen wurde, müssen einige Kontextinformationen gespeichert werden. Dazu gehören z.B. die Registerwerte sowie Stackvariablen. Außerdem müssen wichtige Zeiger wie der aktuelle Befehlszeiger (*program counter*), das Programmstatuswort sowie der Zeiger des Kellerregisters (*stack pointer*) gespeichert werden.
Das Wechseln des Prozesszustands wird über *Interrupts* ausgelöst. Dafür sind an festen Speicheradressen *Interruptvektoren* abgelegt, die auf die Adresse der Unterbrechungsroutine zeigen. So sind alle Ein-/Ausgabeklassen mit einem *Interruptvektor* verknüpft, die durch einen *Interrupt* die Unterbrechungsroutine aufrufen, um bestimmte Ereignisse zu signalisieren, z.B. das Tippen auf der Tastatur.
Beim Auslösen eines *Interrupts* sichert die Hardware zunächst einige Register auf dem Stack und übergibt dann die Kontrolle an den *Interrupthandler*.
Der *Scheduler* sichert alle zur Ausführung benötigten Informationen zur Prozess-, Speicher sowie Dateiverwaltung in der *Prozesstabelle* ab. Wird ein anderer *Prozess* für die Ausführung ausgewählt, so müssen diese Information geladen werden. Das Umschalten zwischen *Prozessen* ist also eine kostspielige Systemaufgabe [Tan09].

2.3.1.2 Threads und Multithreading

Neben *Prozessen* gibt es auch noch *leichtgewichtige Prozesse*. Sogenannte *Threads*.
Diese ermöglichen mehrere Ausführungsfäden in einem *Prozess*. Ein *Thread* läuft also in einem *Prozess*, benötigt aber nicht alle Ressourcen bzw. Informationen, die ein *Prozess* bereits bündelt. Zu den Elementen, die jeder *Thread* benötigt, zählen u.A. ein eigener Stack, Register, Befehlszähler etc. Ein *Thread* kann sich in den gleichen

Zuständen wie ein *Prozess* befinden.

Elemente pro Prozess	Elemente pro Thread
Adressraum	Befehlszähler
Globale Variablen	Register
Geöffnete Dateien	Stack
Kindprozesse	Zustand
Signale und Signalroutinen	
Verwaltungsinformationen	

Abbildung 2.5: Benötigte Elemente für Prozesse und Threads [Tan09]

Jedoch teilen sich *Threads* den gleichen Adressraum, was einerseits zu Problemen führen kann, wenn Daten simultan gelesen und geschrieben werden, andererseits aber auch den Vorteil mit sich bringt, dass Daten zwischen mehreren Ausführungseinheiten geteilt werden können. Bei *Prozessen* benötigt man dafür *shared memory* sowie Inter Process Communication (IPC) zum Austausch von Nachrichten. Hierauf wird genauer in Kapitel 2.3.1.3 eingegangen.
Abbildung 2.5 verdeutlicht die Unterschiede zwischen *Prozessen* und *Threads*. Genau wie bei *Prozessen* können *Threads* nebenläufig ausgeführt werden. Das funktioniert analog zur *Multiprogrammierung*, indem zwischen den *Threads* hin- und hergewechselt wird. Mehrere *Threads* in einem *Prozess* zu ermöglichen fällt unter den Begriff *Multithreading*. Außerdem gibt es noch Hardware-Support für das Umschalten von *Threads*, wodurch dieses innerhalb von Nanosekunden erreicht werden kann (z.B. *Hyperthreading* bei Intel-Prozessoren) [Tan09].

2.3.1.3 Kommunikation

Damit mehrere *Prozesse* zusammen an der Lösung von größeren Aufgaben arbeiten können, müssen sie untereinander kommunizieren. Durch das Fehlen eines gemeinsamen Adressraums, können Daten nicht einfach ausgetauscht werden, wie dies bei *Threads* der Fall ist. Aus diesem Grund kommen Techniken wie *shared memory* sowie IPC zum Einsatz.
Ein Beispiel für IPC zwischen *Prozessen* ist die Unix-Shell. So können Kommandos beliebig miteinander in Verbindung gebracht werden.
Abbildung 2.6 zeigt beispielsweise die Interaktion zwischen dem Kommando *echo* und *awk*. *Echo* schreibt Daten auf die Standardausgabe, während *awk* eine Mustergetriebene Such- und Verarbeitungssoftware für Text ist[3]. *Echo* schreibt also den String

```
1 echo bibliography/literature.bib | awk -F "." '{print $1}'
2 bibliography/literature
```

Abbildung 2.6: Unix-Shell IPC

"bibliography/literature.bib" auf die *Standardausgabe*, während *awk* diesen verarbeitet, indem es diesen von der *Standardausgabe* liest und den Teil vor dem Punkt abspaltet. Die Kommunikation wird hierbei durch die beiden Kanäle Standardeingabe sowie Standardausgabe ermöglicht. Dabei handelt es sich um einen unidirektionalen Kanal (*Pipe*). Die Kommunikation erfolgt also nur in eine Richtung.
Das Beispiel zeigt die einfache Kommunikation zwischen zwei *Prozessen*. Außerdem ist es möglich, dass sich *Prozesse* einen gemeinsamen Speicherbereich teilen, auf den diese sowohl lesend als auch schreibend zugreifen können. *shared memory* kann z.B.

[3] Siehe *"man echo"* sowie *"man awk"* für eine Beschreibung der Kommandos

durch Speicher im Random-Access Memory (RAM) oder auf der Festplatte implementiert werden.
Mit der Nebenläufigkeit von *Threads* und *Prozessen* ergeben sich jedoch auch einige Schwierigkeiten. Andrew S. Tanenbaum nennt hierfür das Beispiel eines Flugreservierungssystem, bei dem sich *Prozesse* nicht gegenseitig behindern, wenn sie beide den letzten Sitz reservieren wollen [Tan09].
Ein anderes Beispiel ist das *Erzeuger-Verbraucher-Problem* (*producer-consumer problem*). Es beschreibt einen Erzeuger und einen Verbraucher, die über einen Puffer fester Größe miteinander verbunden sind. Der Erzeuger fügt Daten bzw. Informationen in den Puffer ein, die der Verbraucher anschließend liest. Das können z.B. Daten sein, auf denen der Verbraucher Simulationen oder Berechnungen ausführt. Unabhängig von dem Inhalt des Puffers ergeben sich jedoch einige Schwierigkeiten. Ist der Puffer bereits gefüllt, kann der Erzeuger keine Daten mehr in den Puffer schreiben. Umgekehrt, kann der Verbraucher keine Daten aus dem Puffer lesen, wenn dieser leer ist. Beide müssen also in den genannten Fällen warten, bis sich der Zustand bzw. die Anzahl der Elemente im Puffer verändert hat. Beim Schreiben in den Puffer, muss der Verbraucher zunächst ein Element herausgenommen haben. Beim Lesen hingegen, muss der Erzeuger erst wieder etwas in den Puffer gelegt haben.
Da der Puffer nur eine beschränkte Größe hat, benötigt man also einen Zähler. Zu Problemen kommt es nun, wenn Erzeuger und Verbraucher gleichzeitig die Puffergröße verändern: Der Erzeuger liest die aktuelle Größe aus, wird jedoch vom *Scheduler* schlafen gelegt. Dem Verbraucher wird eine Zeitscheibe zugeteilt und liest sowie inkrementiert den Zähler. Wenn der Erzeuger nun an der Reihe ist, hat er einen falschen Wert für den Zähler, da der Verbraucher ein Element konsumiert hat, der Erzeuger jedoch noch den alten Wert des Zählers im Speicher hat.
Je nach Programmcode kann es leicht zu *Deadlocks* bzw. fehlerhaftem Verhalten kommen. *Deadlock* bezeichnet den Zustand des wechselseitigen Wartens auf eine Aktion, die die beteiligten Akteure selbst nicht mehr anstoßen können.
Probleme dieser Art, die je nach zeitlicher Abfolge von Ausführungsfäden ein unerwartetes Ergebnis produzieren, nennt man *race conditions*. Sie sind nur schwer reproduzierbar, was das Auffinden solcher Fehler erschwert.
Das Problem ist hierbei der simultane Zugriff auf gemeinsamen Speicher. Dieser wird auch als *critical section* bezeichnet. Die im Beispiel ausgeführte Operation war nicht atomar. Der Lese- und Schreibzugriff auf den Zähler wurde durch den *Scheduler* unterbrochen.
Eine einfache, aber ineffiziente Lösung liegt darin, einen wechselseitigen Ausschluss (*mutual exclusion*) zu garantieren. *Prozesse* dürfen also nicht gleichzeitig in ihren kritischen Abschnitten lesen und schreiben.
Zur Lösung dieser Art von Problemen, gibt es einige Primitiven: *Mutex*, *Lock*, *Semaphore*, *Monitor*, *Queue* [Tan09] etc.. Auf diese wird jedoch nicht weiter eingegangen.

2.3.1.4 Python und der Global Interpreter Lock

Die Verwendung von *Threads* kann Geschwindigkeitsvorteile im Gegensatz zu *Prozessen* hervorbringen, da der Kontextwechsel von *Threads* wesentlich schneller ist und mit *Multithreading* sogar Hardware-Support für diesen Zweck existiert [Tan09]. In Python, der Sprache in der ALL implementiert ist, bringt die Benutzung von *Threads* allerdings keinen Geschwindigkeitsvorteil.

Brahler zeigt ein Beispiel auf, bei der eine einfache Countdown-Funktion mit zwei anstatt nur einem *Thread*, die Geschwindigkeit sogar um den Faktor 1,97 verlangsamt. Die Hinzunahme von noch mehr *Threads* zur Berechnung verringert die Performanz sogar noch geringfügig[4] [Bea10].
Der Grund hierfür liegt im Global Interpreter Lock (GIL) des Python-*Interpreters*[5]. Dieser gewährleistet exklusiven Zugriff auf Python Objekte [FBV09]. Er versichert, dass nur ein *Thread* im *Interpreter* gleichzeitig läuft, was einige Implementierungsdetails des in C geschriebenen *Interpreters* [Bea10] vereinfacht.
Python unterstützt zwar *Threads*, jedoch bringen sie für die Parallelisierung von CPU-intensiven Aufgaben keinen Geschwidkeitsvorteil [pyta] [6]. Der GIL[7] stellt also eine Sperre für den Python-*Interpreter* dar, der nur durch die Benutzung von externen C-Modulen manuell freigegeben werden kann [FBV09]. Andernfalls erfolgt die Freigabe bei I/O Aktivität oder für rechenintensive Aufgaben nach 100 Ticks[8] [Bea10]. *Threads* sind jedoch immer noch für I/O-gebundene Aktivitäten nützlich [pyta]. Durch diese Einschränkung ist es also nicht möglich mehr als Quasiparallelität auf mehreren Prozessoren/Kernen zu erreichen [Bea10]. Aus diesem Grund müssen mehrere Instanzen des Python-*Interpreters* ausgeführt werden. Erreicht wird dies durch das Starten mehrerer *Prozesse* [FBV09].
Folglich ist Parallelität mit Python durch *Prozesse* zwar möglich, jedoch würden *Threads* durch den einfacheren *context switch* einen zusätzlichen Leistungsschub ermöglichen.

2.3.2 Verteilte Systeme

In diesem Abschnitt wird dem Leser ein grundlegendes Verständnis für verteilte Systeme vermittelt, die nötig sind, um das verteilte Rechnen von ALL zu verstehen.
Nach Andrew S. Tanenbaum ist ein verteiltes System eine ”[...] Ansammlung unabhängiger Computer, die den Benutzern wie ein einzelnes kohärentes System erscheinen” [TvS08].
Damit wird das Ziel verfolgt, den Zugriff auf entfernte bzw. verteilte Ressourcen zu vereinfachen bzw. überhaupt erst zu ermöglichen, sodass diese vom Benutzer oder einer Anwendung in Anspruch genommen werden können. Ein solches System sollte offen und skalierbar sein.
Jeder Akteur in diesem System kann dabei unterschiedliche Hardware und Betriebssystem(e) aufweisen. Die Aufgabe des verteilten Systems ist es, die Unterschiede zwischen den Akteuren so weit wie möglich zu abstrahieren. Selbst bei einem Defekt einzelner Komponenten oder Akteure sollte es Ausfallsicherheit bieten.
Dafür ist also eine gewisse Zusammenarbeit wie auch Kommunikation erforderlich. Für die Unterstützung der heterogenen Akteure bzw. Komponenten wird meist eine Softwareschicht zwischen Betriebssystem, Kommunikationsgeräte und der eigentlichen Anwendung bzw. Benutzer, implementiert. Die Anordnung durch eine solche Softwareschicht wird als *Middleware* bezeichnet und ist in Abbildung 2.7 zu sehen. Jeder Anwendung wird also die gleiche Schnittstelle angeboten [TvS08].

[4] Das Testsystem war hierbei ein MacPro mit Quad-Core Prozessor [Bea10]
[5] Der standard Python-*Interpreter* ist *CPython* [FBV09]
[6] Zumindest in *CPython*, jedoch aber nicht in *IronPython* oder *Jython*
[7] Implementiert ist der GIL durch ein binäres Semaphor (mithilfe von Mutex und Konditionsvariable der *pthreads* in C) [Bea10]
[8] Ein Tick entspricht in etwa einer Instruktion des Python-*Interpreters* [Bea10]

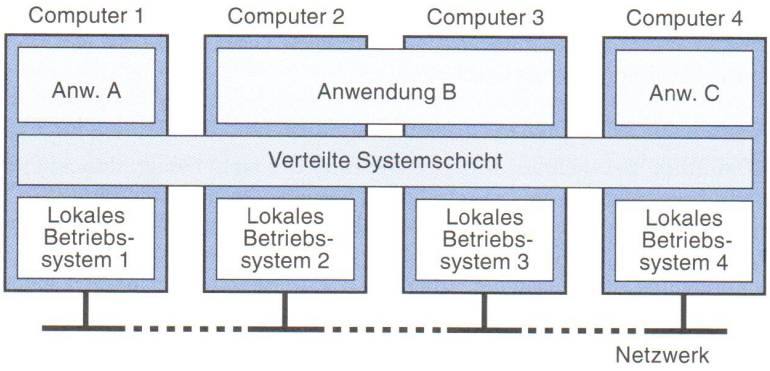

Abbildung 2.7: Middleware [TvS08]

2.3.2.1 Kommunikation

Aufgrund des Fehlens von *shared memory* erweist sich die Kommunikation in verteilten Systemen schwierig. Jedoch ist die Kommunikation zwischen *Prozessen* grundlegend für ein solches System.
Die verbreitesten Kommunikationsmodelle verteilter Systeme sind: Remote Procedure Call (RPC), Message-Oriented Middleware (MOM) sowie *streamorientierte Kommunikation*.
Ersteres ist hauptsächlich für die Kommunikation in Client-Server Architekturen durch den Aufruf entfernter Prozeduren gebräuchlich. Somit wird die Komplexität des Nachrichtenaustauschs beim Aufrufen der entfernten Prozedur (auf dem Server), die vom Client initiiert wurde, weitgehend verborgen.
Die *streamorientierte Kommunikation* wird eingesetzt, wenn Medien wie Audio und Video vorliegen und ein kontinuierlicher Datenfluss benötigt wird.
Für Situationen, in denen die Client-Server Architektur nicht vorliegt und auch keine Medien verwendet werden, bietet sich Message-Oriented Middleware zur Kommunikation an, auf die im Folgenden näher eingegangen wird, da ALL diese zum verteilten Rechnen benutzt [TvS08].

2.3.2.2 Nachrichtenbasierte persistente Kommunikation

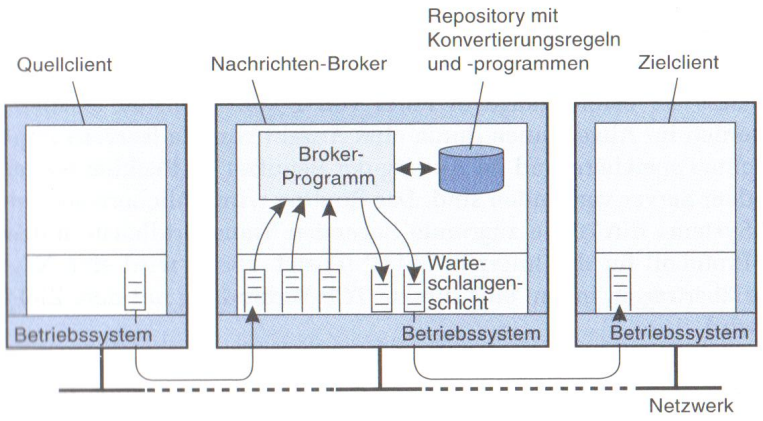

Abbildung 2.8: Message-Oriented Middleware am Beispiel eines Warteschlangensystems [TvS08]

Nachrichtenbasierte persistente Kommunikation wird mit Warteschlangensystem (*Message Queuing Systems*), die ebenfalls zu Message-Oriented Middleware zählen, implementiert.
Dabei handelt es sich um eine Form der asynchronen Kommunikation, bei der Nachrichten ausgetauscht werden. Die Kommunikation zwischen Sender und Empfänger

erfolgt nicht direkt, sondern über eine Warteschlange.
Persistent ist die Kommunikation dadurch, dass, wenn der Sender eine Nachricht transferiert, diese im Warteschlangensystem gespeichert wird. Es wird keine Garantie dafür übernommen, dass sie auch an den Empfänger übermittelt wurde, sondern nur, dass sie in die Warteschlange eingereiht wurde. Der Empfänger kann die Nachricht anschließend aus der Warteschlange entnehmen. Wie genau auf das Warteschlangensystem zugegriffen wird, d.h. wie Nachrichten gesendet und empfangen werden, wird näher in Kapitel 2.3.2.2 erläutert .
Die Kommunikation ist folglich zeitlich lose gekoppelt und daher asynchron.
Sie bietet durch eine zusätzliche Abstraktionsschicht den Vorteil von Fehlertoleranz, da Nachrichten (persistent) gespeichert werden.
Abbildung 2.8 zeigt die Architektur eines solchen Warteschlangensystems. Zusätzlich zu der eigentlichen Warteschlange ist noch ein *Nachrichten-Broker* abgebildet. Diesem wird die Aufgabe zuteil, für ein einheitliches Nachrichtenformat zu sorgen. Er nimmt also eine Umwandlung der Nachrichten vor, sodass die jeweilige Anwendung sie versteht. Er ist kein eigentlicher Bestandteil des Warteschlangensystems, sondern kann als Gateway auf Anwendungsebene gesehen werden [TvS08].
Ein Szenario für verteiltes Rechnen könnte also nun wie folgt aussehen: Der Sender sendet eine Menge von Jobs an das Warteschlangensystem (über den *Nachrichten-Broker*). Registrierte Arbeiter holen sich Aufgaben, führen ihre Berechnungen aus, und fügen ihre Ergebnisse in eine Ergebniswarteschlange ein, aus der der Sender diese entnehmen kann.
Die Operationen, mit denen Sender und Empfänger arbeiten können, sind folgende:

- Put: Eine Nachricht in die Warteschlange einreihen (nicht blockierend).
- Get: Synchrones Empfangen der am längsten wartenden Nachrichten aus der Warteschlange (Aufruf blockiert bis Nachricht erhalten).
- Poll: Überprüfen auf neue Nachrichten sowie Herausholen falls eine Neue existiert (asynchron, d.h. blockiert nicht).
- Notify: *Callback-Handler* einrichten, der über neue Nachrichten in der Warteschlange benachrichtigt, indem die *Callback-Funktion* aufgerufen wird.

2.3.2.2.1 AMQP ist ein Open-Source Standard und Protokoll, das den Nachrichtenaustausch für Warteschlangensysteme spezifiziert. Ursprünglich aus dem Finanzsektor entstanden, definiert es die Interaktionen zwischen Sender (*publisher*) und Empfänger (*consumer*), eine Reihe von Kommandos sowie das Nachrichtenformat.
ALL benutzt *RabbitMQ* als Warteschlangensystem. Eine in Erlang geschriebene Implementierung des Advanced Message Queueing Protocol (AMQP), die außerdem Clustering bietet und somit beliebige Formen von Messaging-Topologien errichtet werden können, je nach Anwendungsbereich und Skalierbarkeitsanforderungen [Dos14].

3 Verwandte Arbeiten

Im folgenden Kapitel werden mit ALL verwandte Arbeiten vorgestellt. Zunächst wird das einzige skalierbare Analysewerkzeug vorgestellt, das während der Recherche gefunden wurde. Anschließend werden sonstige Programme, die mithilfe statischer- und/oder dynamischer Analyse von Android-Anwendungen, sicherheits- und datenschutzkritische Aspekte untersuchen, präsentiert.
Abschließend wird ein Vergleich zweier Frameworks zum verteilten Rechnen vorgestellt.

3.1 PlayDrone

An Applikationen zur Untersuchung von Android-Applikationen mangelt es nicht. Allerdings fehlen skalierbare Tools zur Massenanalyse.
PlayDrone ist ein verteilter und skalierbarer Crawler des *Google Play Store* sowie ein Analyse-Framework für APKs. Es ermöglicht das Durchforsten des *App Store* und Herunterladen von APKs. Dabei werden die APKs in lokalen *git repositories* auf den Workern zusammen mit ihren Metadaten versioniert gespeichert.
Es bietet durch seine *Plugin-Middleware* das Erweitern um selbstgeschriebene Skripte für benutzerdefinierte Analysen. Zum Zeitpunkt dieser Thesis ist es das einzige bekannte skalierbare und verteilte System zur Analyse von Android-Applikationen [VGN14].

3.1.1 Architektur

Die Architektur von *PlayDrone* ist in Abbildung 3.1 dargestellt. Entwickelt wurde es in der Programmiersprache *Ruby* und besteht aus dem asynchronen processing Framework *Sidekiq*, das als Job *Scheduler* dient, dem Key-Value-Store *Redis*, der als asynchrone *Job Queue* genutzt wird und *Elasticsearch*, einer verteilten Such- und Analyseengine.
Die Metadaten sowie die App selbst werden jeweils in einem *git repository* auf den verschiedenen *Knoten* gespeichert. Somit wird eine Versionsverwaltung eingeführt. Anfragen an den *Play Store* gehen indirekt über einen *Amazon EC2 Proxyserver*. Die

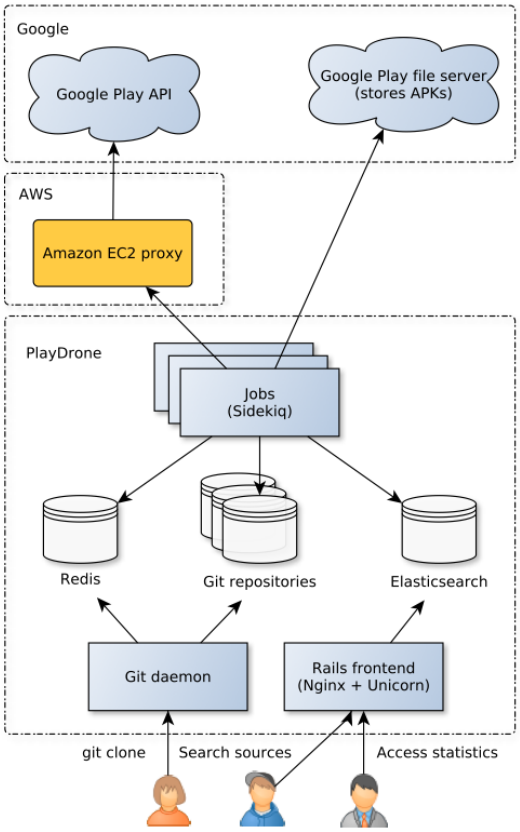

Abbildung 3.1: PlayDrone Architektur [VGN14]

Benutzeroberfläche wird durch das Webframework *Ruby on Rails* und den Webserver *nginx* bereitgestellt [VGN14].

3.1.2 Crawlvorgang

Das *Crawlen* des *Play Store* ist nicht so leicht, da keine vollständige und öffentliche Liste von Android-Anwendungen existiert. Die API des *Play Store* limitiert die Auflistung von Apps auf 500, um ein komplettes *Crawlen* zu verhindern. Dieser Restriktion entgeht *PlayDrone*, indem es mithilfe eines Wörterbuches, das Einträge aus verschiedenen Sprachen beinhaltet, Anfragen startet.
Es werden zwei unterschiedliche *Job Queues* definiert. Eine für das Entdecken von neuen APKs, die andere zum Herunterladen und Analysieren.
Suchanfragen werden mit Termen aus dem Wörterbuch gestartet, wobei jede Suchanfrage nur ein Wort enthält und als ein Job definiert ist. Die Anfrage wird über den *Proxyserver* an die API des *Play Store* gesendet. Falls ein neues APK entdeckt wurde, wird ein neuer Job zum Herunterladen und Analysieren initiiert, indem er über *Sidekiq* in *Redis* gespeichert und von dort aus an einen Knoten weitergeleitet wird. Dieser lädt die Metadaten sowie das APK herunter und speichert es in einem eigenen *git repository*. Jede Version wird mit einem Tag versehen, der entweder die Version oder der Zeitpunkt des Downloads ist.
Der Knoten dekompiliert außerdem den Quellcode und speichert ihn mit im *repository* ab, sodass z.B. Versionsvergleiche einfach mit *git* einsehbar sind. Außerdem werden die Metainformationen sowie der dekompilierte Quellcode in *Elasticsearch* abgelegt. Das System implementiert Fehlertoleranz, indem bei Netzwerkfehlern etc. die Jobs erneut in die Warteschlange eingereiht werden.
Weiterhin werden beim Herunterladen der Metadaten eine Liste von verwandten Ap-

plikationen zur weiteren Entdeckung von Android-Apps benutzt.
Insgesamt wurde mit *PlayDrone* eine Kollektion von über 1.100.000 APKs erstellt. Davon sind 880.000 kostenlose Apps, die alle dekompiliert wurden. Die Daten beziehen sich auf den 30. November 2013. Da der *Play Store* im Juli 2013 die Grenze von 1.000.000 Applikationen erreicht hat, wird eine Gesamtabdeckung von 90% eingeschätzt. Jedoch wurden nur Applikationen heruntergeladen, die auf die Anforderungen eines Galaxy Nexus passen, da nicht kompatible Apps erst garnicht über den *Play Store* propagiert werden. Das Entdecken neuer Applikationen sowie Herunterladen wird auf täglicher Basis durchgeführt. Morgens werden alle Metadaten über die *Details API* des Store aktualisiert und den restlichen Tag wird der *Play Store* nach unbekannten Anwendungen durchforstet und neue Anwendungen bzw. Versionen heruntergeladen [VGN14].

3.1.3 Analyse

Das System ist durch den Einsatz von Plugins erweiterbar. So ist z.B. das Dekompilieren als Plugin mithilfe der Programme *apktool* und *JD-Core* implementiert.
Als weitere Checks sind u.a. eine Ähnlichkeitsanalyse von Anwendungen zur Feststellung von Klonen sowie die Benutzung von Bibliotheken integriert. Außerdem ist mithilfe von *PlayDrone* festgestellt worden, dass *access tokens* für die Benutzung des *OAuth* Authentifizierungsprotkolls sowie für Amazon Web Services (AWS) in vielen Apps statisch im Code implementiert sind. Somit sind sie nicht sicher, da sie durch das Dekompilieren leicht einsehbar sind. Auch dieser Check ist als Plugin implementiert und verfügbar [VGN14]. Im Folgenden wird noch näher auf die einzelnen Plugins eingegangen.

3.1.3.1 Ähnlichkeitsanalyse von Applikationen

Das Feststellen von Ähnlichkeit hilft dabei Anwendungen zu finden, die entweder duplikaten Inhalt bereitstellen oder eine Applikation klonen. Interessant ist dies sicherheitstechnisch, da sich Applikationsklone als Überträger von Malware herausgestellt haben. Bestehende Applikationen werden mit Schadcode versehen und neu verpackt in einem *App Store* veröffentlicht.
Der Ansatz basiert nicht auf statischer Code-Analyse, da dieses Verfahren einerseits bei *obfuscated code* fehlschlägt und andererseits nicht skaliert, wenn man nahezu den gesamten *Play Store* analysieren möchte, da das Verfahren sehr rechenintensiv ist.
PlayDrone implementiert die Ähnlichkeitsanalyse basierend auf *Ressourcen* und *Assets*[9]. Dazu gehören Bilder, Musikdateien sowie Layout-Files, die die Oberfläche der App definieren.
Zum Feststellen von Ähnlichkeit wird ein *feature set* erstellt, das aus Ressourcen-Namen sowie Signaturen (*MD5*-Hash) der Assetdateien besteht. Als Maß wird ein *score* verwendet, der Werte aus dem Intervall $[0, 1]$ annehmen kann. Dabei bedeutet ein Wert von 1 ein exakter Klon und 0 keinerlei Ähnlichkeit.
Mithilfe dieses *scores* werden ähnliche Apps in Cluster eingeteilt und durch den Entwicklernamen sowie dem APK beigefügten Zertifikat wird die Applikation mit *clone* or *rebranded* markiert [VGN14].
Letzteres bedeutet also eine Neuzusammenstellung der App, da der Entwickler sich nicht verändert hat.

[9] Siehe Abbildung 2.1. Ressourcen liegen unter *res/* und assets unter *res/assets*

3.1.3.2 Access tokens

Ein weiteres Analyse-Plugin sucht nach *access tokens* im Quellcode. Dies geschieht unter Verwendung der Suchengine, die den Quellcode via Volltextsuche scannt.
Access tokens sind eine Möglichkeit, Drittanbieter-Apps für bestimmte Funktionen zu authentifizieren, sowie die Berechtigungen zu bekommen, um auf benutzerspezifische Daten zuzugreifen.
Ein Beispiel hierfür sind Amazon Web Services, Facebook oder Twitter. Mithilfe von *OAuth* kann eine Applikation z.B. auf die Pinnwand in Facebook posten. Das funktioniert, indem jeder Applikation ein Paar bestehend aus ID und geheimen Schlüssel zugewiesen wird. Mit diesem Paar kann sich eine Applikation gegenüber einem *OAuth provider* registrieren und somit Anfragen bzw. Aktionen im Namen der Applikation ausführen. Außerdem kann ein erweiterter *user access token* eingeholt werden, sodass auch Aktionen im Namen des Benutzers ausgeführt werden können.
Ursprünglich für Service-zu-Service-Kommunikation gedacht und korrekt implementiert, stellen solche *access tokens* keine Gefahr dar. Dadurch, dass sich dieses Konzept auch in mobilen Anwendungen etabliert hat, kommt es zu Sicherheitsrisiken, da diese *access tokens* oft im Quellcode abgelegt werden. *PlayDrone* dekompiliert jedoch jede Applikation, weshalb diese leicht gefunden werden können. Dafür werden reguläre Ausdrücke benutzt, die auf den jeweiligen Service zugeschnitten sind.
Wer in den Besitz dieser *credentials* kommt, kann z.B. neue Ressourcen innerhalb der Amazon Web Services allokieren und somit Kosten erzeugen. Außerdem können applikationsspezifische Daten ausgelesen und Phishing-Attacken benutzt werden, um an den *user token* zu gelangen. Dieser ist nicht im Quellcode abgespeichert, sondern wird meist über einen Login-Dialog erst angefordert [VGN14].

3.1.4 Abgrenzung

ALL enthält keinen *Crawler* wie *PlayDrone*. Es bietet zwar die Möglichkeit APKs herunterzuladen, jedoch ist das Auflisten von Android-Applikationenen auf 500 beschränkt [VGN14].
Beide benutzen eine *Job Queue* zum Verteilen der Arbeit auf die Worker. ALL benutzt *RabbitMQ* wohingegen *PlayDrone* den *Key-Value-Store Redis* benutzt. Außerdem unterscheiden sie sich grundlegend in der Speicherung der Daten. Während *PlayDrone* die APKs in lokalen *git repositories* ablegt, sind diese bei ALL entweder auf dem Client gespeichert oder liegen im verteilten Dateisystem von *MongoDB*. Die Speicherung auf den Clients bietet auf der einen Seite eine bessere Datenlokalität, auf der anderen Seite wird *static scheduling* implementiert, da nur Apps, die lokal vorliegen, analysiert werden können.
PlayDrone speichert den Quelltext der Applikationen sowie Metadaten in der Suchmaschine *Elasticsearch*, wodurch eine effiziente Volltextsuche möglich ist. ALL hingegen speichert die Daten im *JSON*-Format optional lokal auf der Festplatte sowie in der *MongoDB*, wodurch auch komplexe Abfragemuster zur Auswertung der Resultate möglich sind.

3.2 ScanDal

ScanDal ist ein statisches Analysewerkzeug für Android-Applikation. Es untersucht, ob sensible und private Daten nicht nur gelesen, sondern auch weitergegeben werden. Kim et al. weisen darauf hin, dass der Zugriff auf sensible Informationen zwar über die Berechtigungen deklariert werden müssen, jedoch keine Kontrolle über den anschließenden Datenfluss existiert.
Um den Datenfluss analysieren zu können, liest *ScanDal* den *Dalvik-Bytecode* aus der *classes.dex* aus und konvertiert diesen durch Erstellen eines Control Flow Graph (CFG) in die Zwischensprache *Dalvik Core*. Diese abstrahiert die *Dalvik*-Instruktionen und erleichtert die Analyse.
Damit der Informationsfluss nachvollzogen werden kann, prüft *ScanDal* auf den Einsatz von API-Funktionen, die sensible Daten wie Standortinformationen, Telefondaten wie die eigentliche Nummer und die Geräte-ID (*IMEI*) und zusätzlich Audio- und Videodaten liefern.
Funktionen aus dem Android-Framework, mit denen Daten weitergegeben werden können, werden von Kim et al. als *Sinks* bezeichnet. Dazu zählen API-Funktionen die Daten über das Netzwerk oder SMS transferieren bzw. im Dateisystem speichern können.
Existiert ein Datenfluss zwischen einer Quelle, die sensible Daten offenlegt und durch einen *Sink* fließt, wird er durch *ScanDal* erkannt.
Mögliche Quellen und *Sinks* sind fest in das Analysewerkzeug integriert. Probleme treten allerdings auf, wenn *Reflection* oder das Java Native Interface (JNI) benutzt wird, um nativen Code auszuführen und kann im ersten Fall nur partiell, im zweiten gar nicht erkannt werden.
ScanDal arbeitet direkt auf dem *Bytecode* der DVM, denn das Übersetzen in JVM-Bytecode ist fehleranfällig [KYY+12].

3.3 Julia

Julia ist ein kommerzielles Produkt für die statische Analyse von Java-Programmen. Basierend auf dem *Bytecode* der JVM werden semantisch Programme auf formale Korrektheit untersucht, um Programmierfehler aufzudecken.
Payet et al. haben *Julia* so erweitert, dass die Besonderheiten des Android-Systems bei der Analyse berücksichtigt werden. Dazu gehört, dass mehrere mögliche Startpunkte in Android-Applikationen existieren, da die Bibliotheken der Android-Umgebung eventbasiert sind.
Darüber hinaus ist nicht der komplette Quelltext im *Dalvik-Bytecode* präsent, da grafische Elemente aus *XML*-Layoutdateien dynamisch durch die Verwendung von *Reflection* in *Java-Bytecode* umgesetzt werden.
Julia erstellt intern einen *Kontrollflussgraphen*. Bedingt durch das Konzept der Objektorientierten Programmierung werden die Implementierungen der aufgerufenen Methoden dynamisch zur Laufzeit in der jeweiligen Klasse nachgeschlagen, sodass der exakte *Kontrollflussgraph* nicht erstellt werden kann. Das Problem kann jedoch gelöst werden, indem eine Approximation der möglichen Methoden vorgenommen wird, indem das Einführen von neuen Typen durch das Kommando *new* nachvollzogen wird. *Julia* verwendet dabei eine verbesserte Version der *0-CFA class analysis*.

Basierend auf der abstrakten Darstellung der Android-Anwendung implementiert die Erweiterung von *Julia*, die Payet et al. vorgenommen haben, die folgenden semantischen Tests, die auf Bugs hindeuten können, aber nicht müssen:

- Equality: In Java kann die Gleichheit von Variablen mit == und equals überprüft werden. Dabei können je nach Vergleich Fehler entstehen, da == auf der Adresse des Zeigers und equals auf der Implementierung der equals() Funktion basiert. Die Benutzung beider Vergleichsoperatoren auf dem gleichen Objekt oder auf Arrays deuten auf Programmierfehler hin.
- Class casts: Durch das Android-System müssen insbesondere bei der Inflation der Layout-Dateien *casts* vorgenommen werden. Ein Cast auf die falsche Klasse führt jedoch erst zur Laufzeit zu einem Fehler.
- Static update: Die Definition von statischen Feldern innerhalb des Konstruktors oder Instanzmethoden deuten auf Bugs hin.
- Dead code: Code, der niemals aufgerufen wird.
- Inkonsistenzen zwischen *equals* und *hashCode* in der Implementierung können zu undefiniertem Verhalten in einer *Collection* führen. Überprüft wird, ob beide Methoden definiert wurden.
- Überschreiben von Methoden: Das korrekte Überschreiben von Methoden kann durch eine falsche Methoden-Signatur fehlschlagen.
- Nullness: Besitzen Objekte den Wert *null*, kommt es beim Zugriff zu einem Laufzeitfehler.
- Terminierung: Die Erweiterung von *Julia* überprüft, ob Methoden terminieren.

Die Analyse basiert allerdings auf dem *Java-Bytecode*. Entweder muss die Applikation als .jar-Datei exportiert werden (z.B. bei der Entwicklung via *Eclipse*) oder mit einem Werkzeug wie *dex2jar* oder *undx* konvertiert werden [PS12].

3.4 SAAF

SAAF ist die Abkürzung für *Static Android Analysis Framework*. Dabei handelt es sich um ein vollautomatisiertes Werkzeug für die statische *Datenflussanalyse*, um die Argumente, die beim Aufruf einer Methode übergeben wurden, zurückzuverfolgen. Neben der *Datenflussanalyse* kann *SAAF* außerdem *Kontrollflussgraphen* erstellen und Quellcode auffinden, der in Verbindung mit Werbung steht.
Während für eine bessere Automatisierung das Kommandozeileninterface genutzt werden kann, bietet die grafische Oberfläche eine einfachere Bedienung.
Analysiert werden Applikationen auf Basis des *smali-code*. Dieser disassemblierte Code wird durch den Einsatz von *android-apktool* erzeugt. Durch das Arbeiten direkt auf dem *Bytecode*, ohne den Code zu dekompilieren, wird eine höhere Präzision erzielt, da ein *Decompiler* in vielen Fällen den Quellcode nicht wiederherstellen kann.
SAAF ermöglicht die Suche nach Konstanten, die an eine Methode übergeben werden. Durch Auffinden dieser, kann z.B. das Senden einer SMS an eine fest einkodierte Telefonnummer, lange *sleep*-Intervalle oder Aufrufe des *sudo*-Kommandos erkannt werden. Alle gefundenen Konstanten werden in einer *MySQL*-Datenbank abgelegt,

sodass der Analyst durch Heuristiken entscheiden kann, ob die App näher untersucht werden soll [HUHS13].
Möglich ist diese Suche nach Konstanten durch eine *Datenflussanalyse*. Es wird untersucht wie Daten das Verhalten des Programmes beeinflussen [Kri04]. Die Methode, die *SAAF* benutzt, wird *static backtracking* bzw. *program slicing* genannt. Damit die Analyse erfolgen kann, muss ein *slicing criterion* angegeben werden, das die Methodensignatur, den vollständigen Klassennamen sowie den Parameterindex enthält, der auf konstante Werte überwacht werden soll.
program slicing versucht die Beeinflussung eines Statements auf ein Anderes aufzudecken. Genauer verwendet *SAAF* jedoch ein *backward slice*. Bestimmt durch das *slicing criterion* werden alle *opcodes*, die aufgerufen werden, gescannt. Anschließend werden die Register ausfindig gemacht, die die Werte des definierten Methodenparameters enthalten, damit rückwärts alle Interaktionen zurückverfolgt werden können, bis auf eine Konstante oder Objektreferenz gestoßen wird. Im ersten Fall war die Suche erfolgreich und die Konstante wird in der Datenbank gespeichert. Im zweiten Fall wurde lediglich das Abbruchkriterium erreicht [HUHS13].

3.5 Androguard

Die wichtigste verwandte Arbeit ist *androguard*. Der Kern von ALL, der die Analysefunktionalität für Android-Applikationen bereitstellt, ist durch die Benutzung von *androguard* implementiert.
Als Open-Source-Software beinhaltet das Framework Werkzeuge zur Manipulation und Analyse von *Dalvik-Bytecode*, APKs und Androids als binäres *XML* kodiertes Manifest.
Durch den alternativen Python-*Interpreter IPython*, bietet es eine interaktive *Shell* zum statischen Analysieren und Modifizieren der Android-Applikationen. Zum Beispiel können der *Dalvik-Bytecode* oder der dekompilierte Quelltext eingesehen werden. Außerdem ermöglicht es das Arbeiten mit *basic blocks*, *Dalvik*-Instruktionen und Android-Permissions. Programme können außerdem als Graph visualisiert werden, indem die Methodenaufrufe dargestellt werden. Durch Signaturen können Apps schnell auf bekannte *Malware* überprüft werden, indem die Signatur mit einer an *androguard* gekoppelten Datenbank abgeglichen wird [Des].
Des Weiteren implementiert *androguard* einen Algorithmus zur Erkennung von Ähnlichkeiten zwischen zwei Applikationen. Ähneln sich Applikationen, kann dies ein Indiz für *Malware* sein, indem Applikationen modifiziert und als neues APK kompiliert werden. Neben der Berechnung des Ähnlichkeitsmaßes können Unterschiede zweier Apps komfortabel identifiziert werden, sodass eingefügter Schadcode schnell erkannt werden kann.
Sehr interessant ist das Framework außerdem, da es einen *Decompiler* zur Verfügung stellt, der direkt aus dem *Dalvik-Bytecode* Java-Quelltext erzeugt. Somit entfällt eine Indirektionsstufe, indem nicht zunächst der *Dalvik-Bytecode* in JVM-*Bytecode* konvertiert wird[10], um anschließend einen auf JVM-basierenden *Decompiler*[11] zu benutzen.
Um die beschriebene Funktionalität zu implementieren, führt *androguard* sowohl *Daten-* als auch eine *Kontrollflussanalyse* durch. Davor muss jedoch zunächst aus der *classes.dex* das *Disassembly* erstellt werden. Die *Datenflussanalyse* verbessert die so

[10] Dies ist z.B. mit *dex2jar* oder *ded* möglich
[11] Siehe z.B. *jd-gui* oder *dave*

erzeugte Repräsentation des *Bytecodes*, indem Konstanten propagiert und gemeinsame Unterausdrücke ersetzt werden.
Mithilfe der *Kontrollflussanalyse* wird der Kontrollfluss der Methoden zum Erkennen von Bedingungen und Schleifen genutzt. Abschließend wird für weitere Optimierungen und endgültige Repräsentation ein Abstract Syntax Tree (AST) erstellt.
androguard bildet Klassen und Methoden mit Sichtbarkeit, Typ und Namen auf Python-Objekte ab. Klassen haben zusätzlich noch eine Liste von Methoden. Außerdem kennen die Methoden ihre Parameter und haben ihren *Bytecode* vorliegen [DG11].
Alle Klassen sind in einem Objekt (*DalvikVMFormat*) gespeichert, das den gesamten *Dalvik-Code* repräsentiert. Dadurch, dass jeder Methode der *Bytecode* vorliegt, kann auch selektiv dekompiliert werden.
Wer den integrierten *Decompiler DAD* nicht benutzen möchte, kann bekannte Werkzeuge wie z.B. *jd-gui* benutzen [Des].
Auf die Analyse-Objekte, die die Funktionalität für statische Analyse ermöglichen, wird im Design-Kapitel in 4.5.1 eingegangen.
Androguard ist sehr flexibel einsetzbar, dadurch, dass die komplette Funktionalität über eine *Python-API* propagiert wird. Komfortabel kann mit der Klasse *AndroAuto* zwar ein eigener *Thread*-basierter Analysierer geschrieben werden, wie jedoch in Kapitel 2.3.1.4 beschrieben, nimmt bei mehreren *Threads* auf einem Multiprozessorsystem die Leistung sogar noch ab, wenn diese für CPU-intensive Aufgaben in *CPython* benutzt werden.

3.6 Andrubis

Ein sehr interessantes System stellt *Andrubis* dar, da es statische und dynamische Analyse vereint, um *Malware* zu erkennen. Dabei wird das *Malware-Analysetool Anubis* um Android-Funktionalität erweitert, sodass sowohl die DVM als auch das System überwacht werden. Erreicht wird dies durch Emulierung der Applikationen mithilfe von *Qemu*, wodurch *system calls* und die Benutzung nativer Bibliotheken überwacht werden kann. Zusätzlich wird die *DVM* kontrolliert. Beispielsweise findet ein *taint tracking* statt, bei dem der Informationsfluss sensibler Daten im Auge behalten wird.
Die statische Analyse extrahiert mithilfe von *androguard* [Des] Informationen aus dem Manifest sowie aus dem *Dalvik-Code* die Liste aller Klassen und Methoden. Die gesammelten Informationen werden für eine effizientere dynamische Analyse verwendet.
Anschließend werden die Ergebnisse der dynamischen Analyse ausgewertet und z.B. der mitgeschnittene Netzwerkverkehr analysiert.
Die Analysefunktionalität des Werkzeugs ist online verfügbar. APKs können über ein Webformular hochgeladen und analysiert werden[12] [WNL+14].

3.7 Verschiedene

Es gibt noch eine Vielzahl anderer statischer und dynamischer Analysewerkzeuge für Android, weshalb diese im Folgenden nur kurz angeschnitten werden.
Neben *Andrubis* benutzen *MalloDroid*, *CryptoLint* und *Androwarn* ebenfalls *androguard*. Ersteres versucht mithilfe statischer Analyse die fehlerhafte Benutzung des *SS-*

[12] Siehe `https://anubis.iseclab.org`

L/TLS-Protokolls aufzudecken, sodass eine Man-in-the-middle (MITM)-Attacke möglich ist [FHM+12].

CryptoLint prüft auf typische Fehler bei der Benutzung der Java Krypto API. Durch *program slicing* wird der Fluss zwischen kryptografischen Schlüsseln, Initialisierungsvektoren und anderen Kryptooperationen nachvollzogen. Dafür arbeitet das Werkzeug direkt auf dem *smali*-Code. Nicht berücksichtigt werden jedoch eigens entwickelte kryptografische Lösungen oder Kryptografie, die nicht über die Java API genutzt wird. Somit ist nativer Code ebenfalls ein Problem, da er nicht analysiert werden kann [EBFK13].

Androwarn benutzt *Datenflussanalyse*, um boshaftes Verhalten von Apps zu erkennen. Dazu gehört z.B. das Auslesen von Geoinformationen oder dem Ausführen von nativem Code. Die Ergebnisse werden als Report, je nach einstellbarer Ausführlichkeit, generiert [D.].

Für eine interaktive Analyse eignet sich *Dexter*. Online verfügbar, können komfortabel und einfach APKs eingesehen werden. Der Analyst kann sowohl das *Disassembly* als auch den dekompilierten Quellcode (*JAD*) einsehen. Visualisiert, werden Abhängigkeiten zwischen den Paketen leicht ersichtlich. In *UML*-Diagrammen werden Klassen mit ihren Methoden und einer Vererbungsstruktur aufgezeigt. Im gesamten Code kann nach Strings, Methoden oder Berechtigungen gesucht werden.

Damit eine interaktive Analyse stattfinden kann, bietet es ein Tagging-System. Außerdem können Analysen mit anderen Benutzern geteilt werden [dex].

Schmidt et al. verfolgen einen Ansatz, der auf maschinellem Lernen beruht, um *Malware* ausfindig zu machen. Dafür extrahieren sie die Methodenaufrufe und Vergleichen sie mit denen von bekannter *Malware*, indem Machine Learning Algorithmen zur Klassifikation verwendet werden [SBS+09].

3.8 Frameworks für verteiltes Rechnen

Nachdem diverse Analysewerkzeuge vorgestellt wurden, folgt ein Vergleich von Frameworks zum verteilten Rechnen.

Lunacek et al. versuchen die Frage zu beantworten wie mehrere Jobs am effizientesten mit Python auf einem Cluster ausgeführt werden können. Dabei werden drei Frameworks untersucht: *mpi4py*, *IPython Parallel* und *Celery*.

mpi4py stellt eine Implementierung des Message Passing Interface (MPI) dar, der Standard für High Performance Computing (HPC).

Im Gegensatz zu HPC, bei dem Operationen über einen Zeitraum von mehreren Monaten ausgeführt wird und die Metrik für Performanz Operationen pro Monat sind, beschäftigt sich Many-Task Computing (MTC) mit Aufgaben, die eine große Menge an Ressourcen benötigen, jedoch über einen kürzeren Zeitraum im Vergleich zu HPC.

IPython Parallel und *Celery* ermöglichen zusätzlich noch Fehlertoleranz, z.B. wenn ein Knoten bzw. Prozessorkern ausfällt oder ein Netzwerkfehler auftritt.

Außerdem kann der Pool an Ressourcen dynamisch verkleinert und vergrößert werden. So können je nach Bedarf zusätzliche Ressourcen wie Knoten, hinzugefügt oder entfernt werden bzw. je nach Auslastung die Anzahl der Prozesse zur Ausführung des Jobs angepasst werden.

Zusätzlich zu dieser Elastizität können die Jobs persistent gespeichert werden, sodass diese auch nach einem Neustart des Systems immer noch präsent sind [LBH13].

Im weiteren Verlauf werden *IPython Parallel* und *Celery* näher vorgestellt und abschließend auf die Performanz der beiden Frameworks eingegangen und erläutert, warum *Celery* dem Vorrang für die verteilte Ausführung der Analyse-Jobs gegeben wurde. Zunächst muss jedoch noch die Unterscheidung zwischen *static* sowie *dynamic scheduling* verdeutlicht werden.

3.8.1 Static versus dynamic scheduling

Je nachdem wie die Aufgaben auf die Arbeiter bzw. die zur Verfügung stehenden Ressourcen aufgeteilt werden, spricht man von *static* oder *dynamic scheduling*.
Bei ersterem erfolgt eine faire Aufteilung der Arbeit. Folglich führt jeder Arbeiter die gleiche Arbeitsmenge aus. Diese Aufteilung eignet sich vor allem dann, wenn die Laufzeit der Jobs gleichverteilt ist.
Variiert diese jedoch, kann man eine bessere Lastverteilung erreichen, indem dynamisch, erst nach Beendigung der jeweilige Aufgabe, eine neue Aufgabe verteilt wird. Die Aufgaben werden also je nach Auslastung verteilt, sodass eine bessere Ausnutzung der Kapazitäten der Arbeiter erfolgen kann.
Prinzipiell von Vorteil durch die dynamische Arbeitsaufteilung, kann die Lastverteilung, die meist durch einen *Broker* erreicht wird, jedoch auch zum Flaschenhals werden und eine zusätzliche Latenz implizieren.
IPython Parallel bietet sowohl *static* als auch *dynamic scheduling* an, wohingegen *Celery* nur letzteres benutzt [LBH13].

3.8.2 IPython Parallel

IPython ist als eine Alternative zur Python-Shell gestartet. Zusätzlich zur klassischen Shell, bietet *IPython* ein interaktives webbasiertes Notebook an. Außerdem stellt es ein System zum verteilten Rechnen dar.
Die Architektur von *IPython Parallel* ist in Abbildung 3.2 dargestellt [ipy]. Die eigentlichen Arbeiter sind die *Engines*. Aufgaben bekommen sie über einen *Scheduler* zugewiesen.
Der Benutzer agiert mit einem *Client*, der die Verteilung der Aufgaben vornimmt.
Das System wird dabei durch den *Hub* überwacht. Dieser führt Buch über die offenen Verbindungen der *Engines*, *Scheduler*, *Clients* sowie Jobs und deren Ergebnisse.

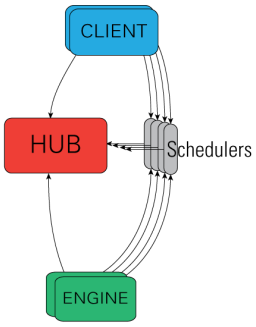

Abbildung 3.2: IPython Parallel Architektur

Zum Austauschen von Nachrichten benutzt *IPython Parallel ZeroMQ*, ein Framework für Message-Oriented Middleware, das jedoch noch Flexibilität darin bietet wie genau der Nachrichtenaustausch implementiert wird [LBH13]. *ZeroMQ* bietet standardmäßig keine Sicherheit, sodass sicherer Netzwerkverkehr nur über *SSH* erreicht werden kann [ipy].

3.8.3 Celery

Celery ist sehr bekannt für das verteilte Abarbeiten von Aufgaben im Webkontext [LBH13]. So kann es z.B. für das sehr bekannte Python Webframework *Django* oder *Flask* benutzt werden.
Es stellt eine asynchrone *job queue* dar, die in Python geschrieben ist und einen *Nachrichten-Broker* zum Austausch von Nachrichten benutzt. Zum Bearbeiten der Aufgaben stellt es *Worker* zur Verfügung, die die Jobs aus dem *Warteschlangensystem* bekommen.
Celery ist dabei sehr flexibel, da jede Komponente erweiterbar oder veränderbar ist. So können z.B. eine Vielzahl von *Nachrichten-Brokern* genutzt werden wie z.B.: *RabbitMQ*, *Redis*, *MongoDB*, *CouchDB* etc. Es müssen also keine *Warteschlangensysteme* benutzt werden, die den AMQP-Standard implementieren, sondern es können auch Datenbanken verwendet werden. Diese bieten allerdings nicht den vollen Funktionsumfang von *Celery* [celb].
Durch den Einsatz von *Kombu*, der Python Bibliothek, die *Celery* für den Nachrichtenaustausch benutzt, können neben den vielen *Nachrichten-Brokern* außerdem beliebige *Serialisierer* sowie *Kompressions-Verfahren* registriert werden. Standardmäßig sind z.B. *Pickle*, JavaScript Object Notation (JSON) *MsgPack* etc. als Nachrichtenformat verfügbar und standard Kompressionsverfahren wie *zip* und *bzip2*.
Auch sicherheitstechnisch ist *Celery* bestens ausgestattet. So bietet es etwa das Signieren von Nachrichten sowie *SSL/TLS*-Kommunikation, zumindest wenn der *Nachrichten-Broker* dieses unterstützt[13].
Zusätzlich bietet es geeignete Monitoring-Werkzeuge wie *Celery Flower* oder Werkzeuge von *RabbitMQ* an. Somit können Warteschlangen aufgelistet, die Benutzung des Speichers analysiert sowie direkt von den *Workern* erweiterte Statistiken und geplante Jobs abgerufen werden[14] .
Es können sowohl einfache als auch komplexe Arbeitsabläufe mithilfe von *Celery* bereitgestellten Primitiven erstellt werden. Dazu gehören die Ausführung einer Gruppe von Jobs sowie die Synchronisierung von Aufgaben, die als Input das Ergebnis eines anderen Jobs benötigen[15].
Genau wie unter Unix mithilfe von *cron* ist es möglich periodische Arbeitsabläufe zu erstellen, die zu einer bestimmten Zeit oder wiederholt ausgeführt werden [celi].
Für Aufgaben deren Laufzeit unbekannt ist, können zeitliche Limits gesetzt werden, die wenn sie eintreten, gesondert behandelt werden können [celg]. Je nach Arbeitslast des Clusters (bestehend aus *Celery Workern*) kann der Ressourcenpool dynamisch der Last angepasst oder ein eigener *Autoscaler* implementiert werden [celi].
Celery ist also ein sehr flexibel einsetzbares Framework, das den individuellen Anforderungen angepasst werden kann. Der Funktionsumfang, die gebotene Flexibilität sowie der Performanzunterschied zu *IPython Parallel*, der im Folgenden aufgezeigt wird, haben zur Entscheidung für *Celery* und gegen *IPython Parallel* sowie *mpi4py* geführt.

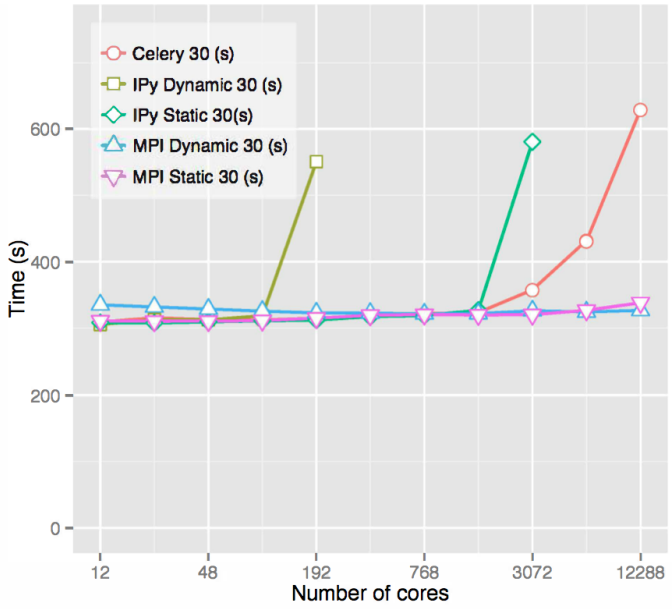

Abbildung 3.3: Skalierbarkeit - MPI vs. IPython Parallel vs. Celery [LBH13]

3.8.4 Performanzvergleich

Abbildung 3.3 zeigt *mpi4py*, *IPython Parallel* und *Celery* im Performanzvergleich. Dabei wurden für *mpi4py* und *IPython Parallel* sowohl ein *static* als auch ein *dynamic scheduling* getestet.

Um einer Beeinflussung durch Speicherzugriff wie Arbeitsspeicher oder Festplatte zu umgehen, wurde als Aufgabe die Ausführung der *sleep*-Funktion gewählt. Es bestehen also keine Abhängigkeiten oder Kommunikation zwischen den Jobs.

Im Test wurde mit einem Knoten angefangen und nach jedem weiteren Durchlauf die Anzahl der Knoten verdoppelt. Mit je 12 Prozessorkernen, ergeben sich mit einer maximalen Anzahl von 1.024 Knoten 12.228 Prozessorkerne. Proportional zur Anzahl der Kerne wurden mehr Jobs der Ausführung hinzugefügt.

Mit bis zu 96 Kernen zeigt sich noch kein Unterschied in der Performanz. Danach verringert sich die Effizienz von *IPython Parallel* mit *dynamic scheduling* jedoch drastisch. Bis zu 1.536 Kernen sind sowohl *IPython Parallel* (*static scheduling*) als auch *Celery* noch sehr effizient. Erst danach verschlechtert sich die Leistung. Bei 3.072 weist *Celery* noch eine Effizienz von 85% (bezogen auf die Ausführung der Jobs) auf, *IPython Parallal* (*static scheduling*) jedoch nur noch über 50%.

Bei der finalen Anzahl an Prozessorkernen (12.228) weist jedoch auch *Celery* nur noch eine Effizienz von knapp unter 50% auf.

Die Vermutung der Autoren ist, dass der *Hub* bei *IPython Parallel* zum Flaschenhals wird [LBH13].

Um die Effizienz von *Celery* noch weiter zu steigern, kann ein Cluster von *RabbitMQ*-Servern verwendet werden[16], da die zentrale Architektur des Brokers vermutlich zum Verhängnis in dem Vergleich wurde.

Insgesamt schneidet *mpi4py* zwar von der Performanz am besten ab, bietet jedoch

[13] Siehe `http://docs.celeryproject.org/en/latest/configuration.html?highlight=ssl#std:setting-BROKER_USE_SSL`. Zuletzt eingesehen am 01.07.2014

[14] Siehe `http://docs.celeryproject.org/en/latest/userguide/monitoring.html#guide-monitoring`. Zuletzt eingesehen am 07.07.2014

[15] Siehe `http://docs.celeryproject.org/en/latest/userguide/canvas.html#guide-canvas` Zuletzt eingesehen am 07.07.2014

[16] `https://www.rabbitmq.com/clustering.html`

weder Fehlertoleranz noch Elastizität an [LBH13]. Außerdem beschreiben Lunacek et al. wie *IPython Parallel* und *Celery* einen Job erfolgreich bearbeiten, bei dem eine Fehlerrate von 10% hinzugefügt wurde, sowie zufällig drei Kerne aus dem Ressourcenpool während der Ausführung entfernt wurden.
Da *Celery* eine bessere Performanz aufweist, wurde sich gegen *IPython Parallel* entschieden.

4 Design

In diesem Kapitel wird das Design von ALL aufgezeigt. Dafür wird zunächst die genaue Problemstellung erläutert, um die Komponenten von ALL anschließend näher beschreiben zu können.

4.1 Problemstellung

Das Problem, das ALL zu lösen versucht, besteht darin, ein einfaches und effizientes Framework zur Massenanalyse von Android-Anwendungen bereitzustellen. Dazu gehört die Verwaltung von APK-Kollektionen, als auch einem flexiblen Skript-Framework, das verwendet werden kann, um beliebige Analysen durchzuführen.
Des Weiteren sollen die Ergebnisse der Analyse in einem strukturierten Format gespeichert werden.
Damit die Ergebnisse einfach eingesehen und verwertet werden können, sollten sie in einer Datenbank gespeichert werden.
Um dem Benutzer einen möglichst großen Komfort zu bieten gehört auch eine grafische Oberfläche zu den Anforderungen.
Das Hauptproblem ist allerdings die Skalierbarkeit solcher Analyse-Tools. Es sollte möglich sein auch große Mengen von APKs zu analysieren, indem die Analyse einerseits effizient ist und auf der anderen Seite auf mehrere Arbeiter verteilt werden kann, sodass eine horizontale Skalierung erreicht wird.
Für kleine Mengen von Apps sollte es jedoch nicht notwendig sein ein verteiltes System installieren zu müssen.

4.2 Workflow

Abbildung 4.1 zeigt den typischen Workflow bei der Benutzung von ALL auf.
APKs werden in eine *SQLite*-Datenbank importiert, die die Metainformationen der Applikationen speichert. Basierend auf dem bekannten Reverse-Engineering-Tool *androguard* bietet das Skripting-Framework von ALL das einfache Schreiben eigener Skripte. Alternativ können von ALL mitgelieferte Skripte[17] benutzt werden.

[17] Eine Auflistung ist im Appendix unter 7.2 einsehbar

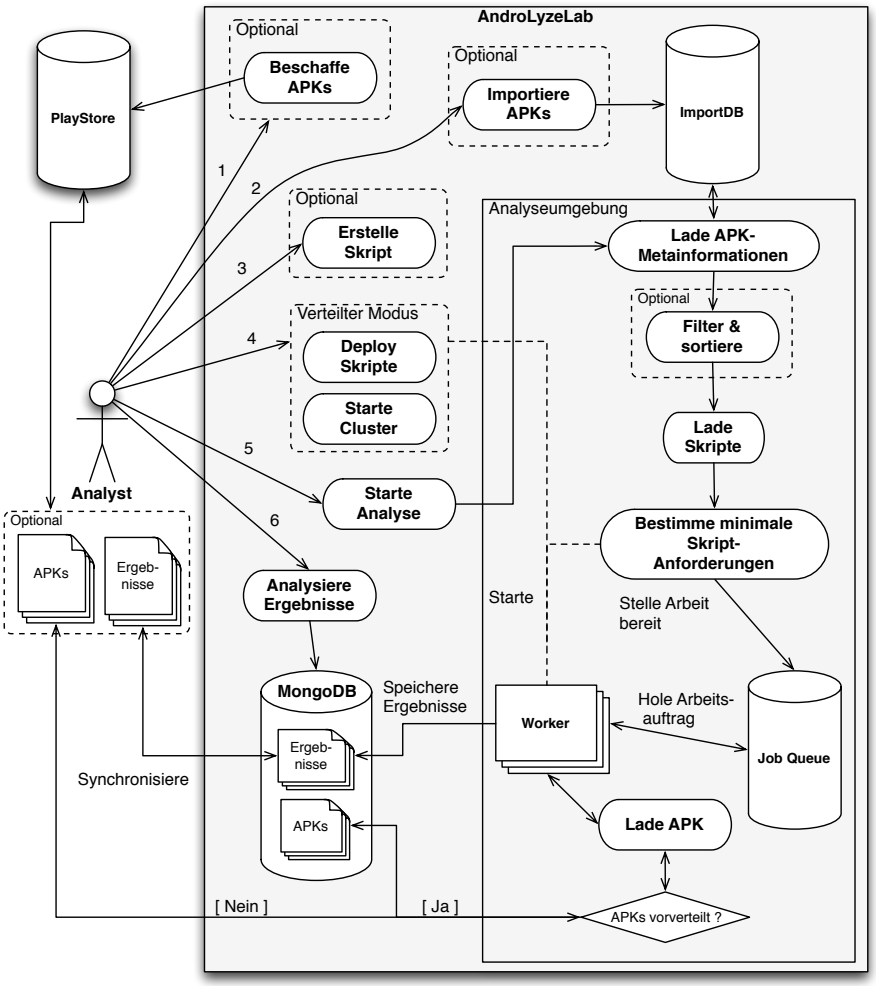

Abbildung 4.1: Workflow AndroLyzeLab

Die Analyse ist parallelisiert, sodass diese effizient entweder lokal auf dem eigenen Computer oder einem Cluster von Arbeitern ausgeführt werden kann.
Die Analyseergebnisse werden sowohl lokal als auch in einer Datenbank gespeichert.
Der typische Arbeitsfluss sieht nun wie folgt aus: Zunächst können Android-Applikationen aus dem *Google Play Store* heruntergeladen werden. Anschließend können diese in die *SQLite*-Datenbank importiert werden, sodass eine Filterung der Apps vor der Analyse vorgenommen werden kann. Entweder werden in ALL integrierte Skripte verwendet oder eigene geschrieben. Im verteilten Modus müssen diese auf den Workern deployed werden.
Anschließend kann die Analyse erfolgen und die Resultate über die Ergebnis-Datenbank abgefragt und analysiert werden.
Damit diese leichter einsehbar sind, können sie zusätzlich lokal als *JSON*-Datei gespeichert werden.

4.3 Gesamtübersicht

Die Paket-Struktur von ALL ist in Abbildung 4.2 dargestellt. Jedes Paket stellt dabei unterschiedliche und eigenständige Funktionalität bereit. Für das Importieren der APKs in die Import-Datenbank ist das Paket *import* zuständig. *model* stellt die Modellierung der Skripte und APKs zur Verfügung.

Zur Verwaltung der Android-Applikationen sowie Speicherung der Analyseergebnisse wird das Paket *storage* genutzt.
Die eigentliche Analyse ist in *analyze* implementiert. In *celery* sind alle Komponenten untergebracht, die für das Framework *Celery* benötigt werden. Dazu gehört dessen Konfiguration sowie Hilfsfunktionalität. Die verteilte Analyse ist jedoch im *analyze*-Paket implementiert.
Die Verwaltung der *Celery-Worker* sowie Deployment des ALL-Code und der Skripte sind im *fabric*-Paket untergebracht.
Komponenten der grafischen Oberfläche sind in *ui* abgelegt.
Außerdem gibt es noch Pakete für die Einstellungen, Logging und sonstige Hilfsfunktionen (*settings*, *log*, *util*).

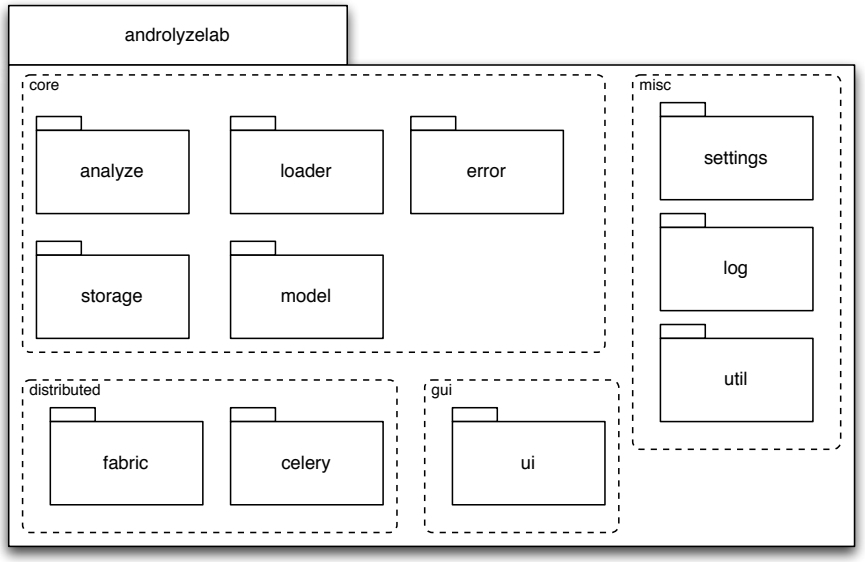

Abbildung 4.2: Paket-Diagramm AndroLyzeLab

4.4 Apk

Zum Modellieren eines APKs für die Analyse benötigt ALL zunächst einen eindeutigen *identifier* für die Applikation. Der Paketname ist zwar eindeutig [andd], jedoch können mehrere Versionen der Applikation präsent sein. Aus diesem Grund wird eine *Hash-Funktion*[18] verwendet. Weiterhin ist die Versionsnummer zur Beschreibung der Applikation wichtig. Beide Elemente sowie die Berechnung des *Hashes* werden beim Importieren aus der Datei *AndroidManifest.xml* gelesen.
Außerdem wird der Pfad sowie das Datum zum Zeitpunkt des Importiervorgangs abgespeichert. Zur Unterscheidung unterschiedlicher Sets in der Datenbank kann optional ein Tag gesetzt werden.
Die eben aufgeführten Attribute stellt die Basisklasse *Apk* bereit, die in Abbildung 4.3 zu sehen ist. Die Hash-Funktionalität wird durch die Klasse *Hashable* zur Verfügung gestellt, die durch die Definition des Dateipfades des APK den Hash einmalig und nur auf Anforderung berechnet.
ALL stellt zwei Modellierungen für eine Android-Applikation bereit. Zum einen die Klasse *FastApk*, die nur die bereits erwähnten Metainformationen zur Verfügung stellt.

[18] Verwendet wird eine SHA-256 Hashfunktion

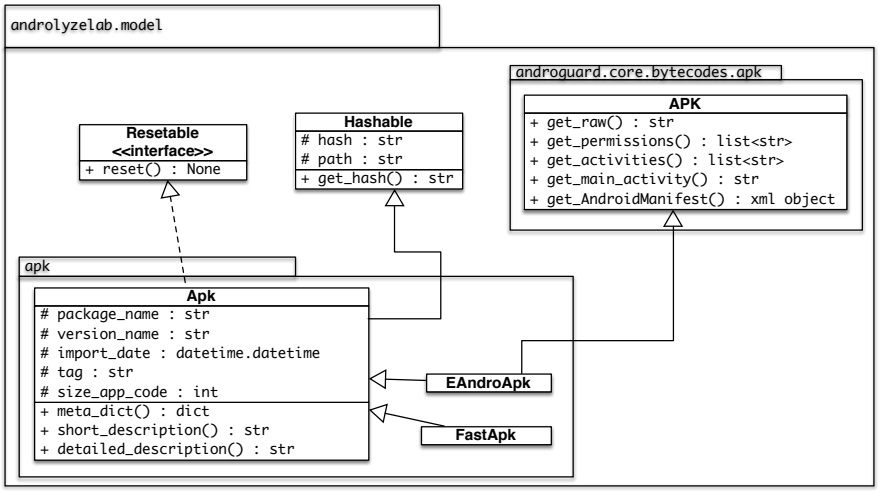

Abbildung 4.3: Apk Modell

Zum anderen die Klasse *EAndroApk*, die zusätzlich noch den Inhalt der APK-Datei bereitstellt sowie alle Informationen, die im Manifest definiert sind. Dafür wird die *APK*-Klasse von *androguard* benutzt. Damit jedoch das gleiche Interface zu den Metainformationen angeboten und das Hashing ermöglicht wird, erbt sie zusätzlich noch von *Apk*[19,20].

4.5 Skript

Das Herzstück von ALL ist das Skript-Framework. Eine noch so effiziente Analyse ist zwecklos ohne ein mächtiges und flexibles Framework zum Definieren von Tests bzw. Analysen, damit Android-Applikationen auf beliebige Eigenschaften untersucht werden können.
ALL schränkt den Analysten dabei nicht auf statische oder dynamische Analyse ein. Im Folgenden wird das Design des Skript-Frameworks näher beschrieben und auf das Logging-Framework eingegangen, das zum strukturierten Loggen der Analyseresultate genutzt werden kann.

4.5.1 Architektur

Benutzerdefinierte Skripte können geschrieben werden, indem sie von einer der in Abbildung 4.4 dargestellten Klassen abgeleitet werden. Die Oberklasse aller Skripte ist dabei *AndroScript*.
Es stellt die Rahmenfunktionalität zur Analyse eines APKs zur Verfügung. Angepasst wird es durch Überschreiben der von *AndroScript* bereitgestellten Methoden.
Die eigentliche Analyse wird dabei in der Methode *_analyze* ausgeführt. Dem Benutzer wird dafür das APK durch die im letzten Kapitel vorgestellte Klasse *EAndroAPK* präsentiert. Es können also sowohl die Metainformationen aus dem Manifest als auch die Rohdaten der Applikation eingesehen werden.
Für die Analyse wird *androguard* benutzt. Das bedeutet, dass die gesamte Analysefunktionalität durch dieses Framework vorgegeben ist.
Um eine effiziente Analyse zu ermöglichen, werden die Analyse-Objekte, die *andro-*

[19] In Python ist Mehrfachvererbung möglich
[20] Man beachte den Unterschied zwischen der von *androguard* bereitgestellten Klasse *APK* und der von ALL genutzten Klasse *Apk*

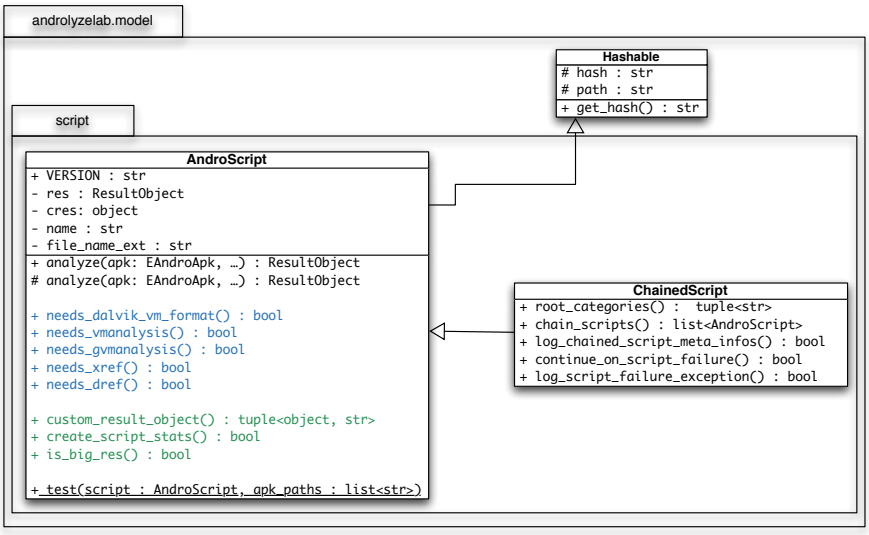

Abbildung 4.4: Skript Modell

guard bereitstellt, nur nach Bedarf initialisiert. Denn das Erstellen dieser Objekte hat einen deutlichen Einfluss auf die Laufzeit, wie ein Experiment in Abbildung 6.5 zeigt. Zum Signalisieren welche Analyse-Objekte[21] benötigt werden, können die im Folgenden gezeigten Methoden der Klasse *AndroScript* überschrieben werden, die im Klassendiagramm in Abbildung 4.4 zu sehen sind:

- needs_dalvik_vm_format() : Das Objekt repräsentiert die Datei *classes.dex*, die den *Dalvik-Bytecode* enthält, indem ein *Disassembly* erstellt wird. Somit kann auf Klassen, Methoden und Felder zugegriffen werden. Durch semantische Informationen im *Dalvik-Bytecode* sind ebenfalls Sichtbarkeit, Typ und Name der Objekte einsehbar.
- needs_vmanalysis() : *VMAnalysis* analysiert den disassemblierten *Dalvik*-Code. Durch die Analyse kann z.B. abgefragt werden an welcher Codestelle welche Berechtigungen benutzt werden[22] und ob dynamisch *Dalvik*-Code nachgeladen wird oder nativer Code sowie *Reflection* benutzt wird. Außerdem kann eingesehen werden, an welcher Stelle ein String, ein Paket oder eine Methode benutzt wird.
- needs_gvmanalysis() : Das Objekt erstellt den *Kontrollflussgraphen*, der für die Querverweise benötigt wird. Außerdem kann ein Graph der Methodenaufrufe erstellt und exportiert werden[23].
- needs_xref() : Mit dieser Option werden die Querverweise zwischen Methoden erstellt.
- needs_dref() : *DREF* erstellt die Querverweise zwischen Feldern.

4.5.2 Logging

Neben der eigentlichen Analyse ist es wichtig, die Ergebnisse strukturiert zusammen mit den Metainformationen des APKs sowie des Skriptes speichern zu können. Als Er-

[21] Die Beschreibung der Analyse-Objekte wurde dem Tutorial, das unter `https://code.google.com/p/androguard/wiki/RE` zu finden ist und dem Quellcode entnommen

[22] Das in ALL integrierte Skript *CodePermissions* nutzt die Klasse, um die Berechtigungen im Code ausfindig zu machen und die Codestücke ebenfalls zu dekompilieren

[23] Siehe Skript *GVMAnalysisExample.py*

gebnisformat wurde sich für *JSON* entschieden, da es eine übersichtliche Darstellung von Daten ermöglicht und im Gegensatz zu *XML* wesentlich kompakter ist.
Die Logging-Funktionalität wird dem Analysten durch die Klasse *ResultObject* bereitgestellt. Diese ist standardmäßig in jedem *AndroScript* enthalten.
Zunächst stellt sich jedoch die Frage, welche Arten von Informationen überhaupt erfasst werden können bzw. sollen.
Zum einen sollte man in der Lage sein einfache Tests zu schreiben. Beispielsweise auf eine Eigenschaft prüfen und entweder wahr oder falsch abspeichern.
Zum anderen sollte man auch beliebige Informationen, die bei der Analyse gesammelt werden, komfortabel loggen können. Das könnten z.B. Informationen aus dem *Manifest* sein. Sammelt man beispielsweise dynamisch in einer Schleife Informationen, sollte es einfach sein, Daten einer Aufzählung hinzuzufügen.
Genau diese Funktionalität bietet die Klasse *ResultObject*.
Neben dem eigentlichen Loggen von Informationen ist es außerdem wichtig, dass die Ergebnisse vergleichbar sind, sodass die Unterschiede zwischen zwei Ergebnissen des gleichen Skripts, aber unterschiedlicher Version einer Android-Applikation, leicht einsehbar sind.
Aus den eben genannten Gründen, muss, bevor das eigentliche Loggen erfolgen kann, zunächst die Struktur des Ergebnisses definiert werden. Dies kann zur Initialisierungszeit des Skriptes erfolgen, aber auch dynamisch zur Laufzeit der Analyse. Wird ein Ergebnis geloggt, das nicht vorher registriert wird, signalisiert der *KeyNotRegisteredError* das Fehlverhalten. Das Registrieren ist notwendig, da Tests, die nicht anschlagen, also keinen Wert Loggen, nicht in dem Ergebnis visualisiert werden und somit Ergebnisse schlecht verglichen werden können.
Die Methoden zum Loggen bzw. Registrieren sind dabei:

- register_keys(keys, *categories) und log(key, value, *categories): Zum Loggen von einzelnen Feldern[24] wie z.B.: str, int, list, bool, dict etc.. Registriert wird standardmäßig None[25].

- register_bool_keys(keys, *categories) und log_true(key, *categories): Ein Test kann entweder positiv oder negativ ausfallen. Daher kann dieser mit einem *boolean* modelliert werden. Registriert wird *False* als Standardwert, der mit *log_true(...)* auf *True* gesetzt werden kann.

- register_enum_keys(keys, *categories) und log_append_to_enum(key, value, *categories): Hinzufügen von Elementen in eine Aufzählung. Standard-Wert ist eine leere Liste.

Für eine übersichtliche und strukturierte Speicherung der Ergebnisse, ist es möglich, verschiedene Kategorien zu definieren, um Ergebnisse zu gruppieren. Es kann eine beliebig tiefe Verschachtelung bzw. Strukturierung vorgenommen werden. Zu sehen ist diese Strukturierung in Abbildung 4.5. Hier sind die Ergebnisse der Analyse unter der Kategorie "*apkinfo*" abgelegt, die eine weitere Unterteilung in die Unterkategorie "*components*" vornimmt.
Zusätzlich zur dynamischen Struktur des Ergebnisses, d.h. den geloggten Ergebnissen

[24] Die Werte die geloggt werden können sind durch den Decoder der *JSON-Bibliothek* beschränkt. Siehe `https://docs.python.org/2/library/json.html#encoders-and-decoders`
[25] None ist ähnlich zu *null*, das u.A. aus Java bzw. JavaScript bekannt ist und wird, wenn es in *JSON* umgewandelt wird, auch in *null* übersetzt

```
{
    "apk meta": {
        "package name": "com.facebook.katana",
        "version name": "11.0.0.11.23",
        "sha256": "
           eda8d69acadb0d892e5f0cf4ac251c148039518444dfb
            52519646f21367444b3",
        "import date": "2014-07-06T18:47:53.299000",
        "path": "/mnt/stuff/btsync/apks_manual_downloads
           /28.06.2014/apps_topselling_free/APP_WIDGETS/
           com.facebook.katana.apk",
        "tag": "top4_free"
    },
    "script meta": {
        "name": "ChainedApkInfos",
        "sha256": "38
           e61c0619dd2569b0989e4dab80c2703f1cc982b52a62b
            4e4aba4dbe11b0a52",
        "analysis date": "2014-07-10T16:16:33.757000",
        "version": "0.1"
    },
    "apkinfo": {
        "components": {
            "activities": {
                "all": [
                    "com.facebook.katana.
                       FacebookLoginActivity"
                ]
            }
        },
        "permissions": [
            "android.permission.READ_CONTACTS"
        ]
    }
}
```

Abbildung 4.5: Beispiel Logging Analyseergebnis

des Analysten, werden die Metainformationen des APKs sowie des Skriptes abgespeichert. Dafür ist das *ResultObject* mit einem *Apk* verknüpft.
Abbildung 4.5 zeigt diese statische Struktur der Metainformationen unter der Kategorie ”*apk meta*” sowie ”*script meta*”. Diese Metainformationen sind in allen Ergebnissen vorhanden, wenn das *ResultObject* zum Loggen benutzt wird. Andernfalls werden sie zumindest in der Datenbank gespeichert.

4.5.2.1 Modulare Skript-Programmierung

In der Softwaretechnik sind *Modularisierung* sowie *Kohäsion* wichtige Konzepte. Dabei geht es darum eine komplexe Struktur in kleine, unabhängige und überschaubare Module zu zerlegen. Jedes Modul beschreibt ein eigenständiges Problem, das so weit wie möglich modularisiert wird. Somit wird ein starker innerer Zusammenhang (*Kohäsion*) sowie eine lose Koppelung erreicht.
ALL unterstützt diesen Ansatz, indem es erlaubt, Skripte beliebig zu modularisieren. Benötigt man dennoch die Resultate kombiniert aus mehreren anderen Skripten, so können diese zu einer größeren Ausführungseinheit gebündelt werden. Diese Funktionalität ist in der Klasse *ChainedScript* implementiert, die ebenfalls in Abbildung

4.4 dargestellt ist.
Eine komfortable Zusammenfassung von Skripten benötigt außerdem eine Reihe von Eigenschaften. Dazu gehört das Gruppieren der Skriptergebnisse unter einer benutzerdefinierten Kategorie und die Möglichkeit, Metainformationen, bezogen auf die Skripte, wie z.B. eine Liste der zusammengefassten Skripte, zu loggen.
Nicht fehlen darf außerdem das Loggen erfolgreicher und fehlgeschlagener Skripte. Ferner ist es möglich zu definieren, ob ein *ChainedScript* nach Fehlschlagen eines der gebündelten Skripte, die verbleibenden Skripte noch ausführen soll. Zusätzlich kann die Exception sowie ihr Traceback geloggt werden.
Abbildung 4.6 zeigt die Metainformationen eines *ChainedScript*, bei dem ein Skript fehlschlägt.

```
"ChainedScript": {
    "scripts": [
        "ClassListing",
        "ClassDetails",
        "Disassembly"
    ],
    "successful": [
        "ClassListing",
        "ClassDetails"
    ],
    "failures": [
        {
            "Disassembly": [
                "Traceback (most recent call last):\n",
                ...
                "IndexError: pop from empty list\n"
            ]
        }
    ]
}
```

Abbildung 4.6: ChainedScript Metainformationen

4.5.2.2 Skript-Testing

Damit Skripte nicht erst zur Laufzeit zu unerwarteten Fehlern oder Ergebnissen führen, darf eine Testumgebung nicht fehlen. Dafür bietet jedes Skript eine Test-Methode[26], der nur eine Liste von Pfaden zu APKs übergeben werden muss. Somit können vor allem nicht registrierte Kategorien schnell entdeckt werden.

4.5.3 Skript-Optionen

Neben den Skript-Anforderungen gibt es noch eine Reihe von Optionen, die aktiviert werden können. Um Flexibilität im Bezug auf das Logging der Ergebnisse zu erreichen, erlaubt ALL dem Benutzer, falls er das eingebaute Logging nicht benutzen möchte, ein Loggingformat selbst zu definieren.
Signalisieren kann der Benutzer dies durch die Methode *custom_result_object()*[27], mithilfe derer ein benutzerdefiniertes Objekt zum Loggen eingerichtet werden kann. Außerdem kann eine andere Dateiendung als ".json" definiert werden, die für die Speicherung des Analyseergebnisses genutzt wird.

[26] Dabei handelt es sich um die statische Methode *AndroScript.test*
[27] Dafür muss die Methode überschrieben werden, sodass sie True zurückliefert

Somit kann z.B. der Quellcode nach dem Dekompilieren als Textdatei[28,29] gespeichert werden.
Außerdem können durch *create_script_stats()* weitere Skript-Statistiken angezeigt werden. Ein Beispiel dafür ist in Abbildung 4.7 zu sehen. Zusätzlich wird die Analysezeit des Skriptes (Laufzeit der *_analyze*-Methode) sowie die Zeit, die benötigt wird, um das APK zu öffnen und die notwendigen Analyse-Objekte bereitzustellen, abgespeichert. Des Weiteren wird die Gesamtzeit für die jeweilige Analyse des Skriptes berechnet.
Für Skripte, die große Mengen an Daten erzeugen, muss dies mit *is_big_res()* signalisiert werden, damit das Dokument in der Ergebnisdatenbank gespeichert werden kann. Dies hängt mit Limitierungen der Dateigröße der Ergebnisdatenbank [monc] zusammen und wird inklusive des Speicherns der Analyseergebnisse nachfolgend näher behandelt.

```
"script meta": {
  ...
    "time script": 0.38891005516052246,
    "time androguard open": 1.426624059677124,
    "time total": 1.8155341148376465
}
```

Abbildung 4.7: Skript Statistiken

4.6 Storage

Der folgende Abschnitt beschäftigt sich mit der Speicherung der Analysedaten und der Verwaltung der APKs.
Für diese Zwecke nutzt ALL zwei verschiedene Datenbanken. Das Importieren und Verwalten der Android-Applikationen wird mit einer einfachen *SQLite*-Datenbank implementiert. Daten werden in sogenannten *Flat-Files* abgespeichert, wodurch die Datenbanken zwischen Analysten austauschbar sind.
SQL eignet sich jedoch nicht für die Speicherung der Analyseergebnisse, da sie nur zum Teil eine statische Struktur aufweisen. Damit der dynamische Charakter erhalten bleiben kann, wird auf die *NoSQL-Datenbank MongoDB* zurückgegriffen. Dabei handelt es sich um eine schemafreie Datenbank, die in C++ implementiert ist.
Sie zählt zu der Klasse der Dokumenten-Datenbanken, die im Gegensatz zu einfachen *Key-Value-Stores* beliebig strukturierte Daten abspeichern können [TB11].
Zusätzlich zur Speicherung der Ergebnisse in *MongoDB*, können sie optional auf der Festplatte als *JSON*-Dateien abgelegt werden. Dafür wird eine Verzeichnistruktur angelegt, die nach Paketnamen Versionsnummer und Hash organisiert ist. Die Verzeichnishierarchie ist in Abbildung 4.8 zu sehen. Außerdem ist zu sehen, dass neben den Resultaten, die unter *res* gespeichert werden, die APKs beim Importieren strukturiert kopiert und einsortiert werden können. Diese werden unter dem Ordner *apk* abgelegt. Das Verzeichnis, in dem die dargestellte Ordnerstruktur abgebildet wird, kann durch den Benutzer vorgegeben werden.

[28] ALL bietet genau diese Funktionalität mit dem Skript *DecompileText.py*
[29] Ein nach Klassen und Methoden strukturiertes *Dekompilierung* (*JSON*) ist außerdem mit *Decompile.py* verfügbar

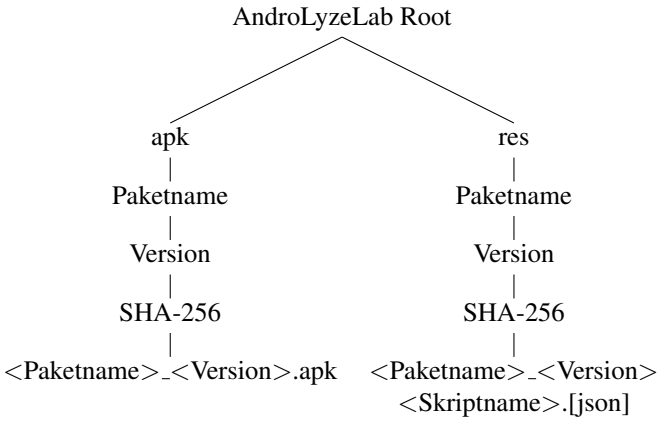

Abbildung 4.8: Verzeichnisstruktur APK und Analyseresultate

4.6.1 APK-Import

Die Datenbank für die APK-Verwaltung speichert alle wichtigen Attribute eines APKs. Dazu gehören der Paketname, die Versionsnummer sowie das Datum des Imports. Außerdem wird der Pfad zur Datei sowie ein Tag, der zur Unterscheidung verschiedener Sets in der Datenbank genutzt werden kann, gesichert. Zusätzlich wird noch die Größe der *classes.dex* Datei abgespeichert, die als Metrik für die Größe des Quellcodes genutzt wird. Zur eindeutigen Identifizierung des Eintrags wird der Hash der APK-Datei als Primärschlüssel[30] benutzt, da zu einem Paketnamen mehrere Versionen existieren können.
Abbildung 4.9 zeigt, dass alle Informationen in einer einzigen Tabelle mit dem Namen *apk_import* abgelegt sind.

Hash	Paketname	Version	Pfad	Importdatum	Tag	Größe *classes.dex*

Abbildung 4.9: Tabelle apk_import zur Modellierung der Import-Daten eines APK

Um an die Informationen, wie den Paketnamen sowie die Versionsnummer zu gelangen, müssen diese beim Importieren aus dem *Manifest* ausgelesen werden.
androguard bietet Funktionalität zum Öffnen des APKs und Auslesen der Metainformationen, jedoch wird dabei die gesamte *Zip*-Datei entpackt und in den Speicher gelesen[31]. ALL optimiert den Sachverhalt dadurch, dass lediglich die *Manifest-Datei* aus dem *Zip*-Archiv ausgelesen wird[32]. Das Manifest ist binär kodiert [Nol12], *androguard* bietet jedoch Funktionen zum Konvertieren an.
Die Datenbank, die für den Importiervorgang benutzt wird, kann durch den Benutzer vorgegeben werden, sodass man auch verschiedene Datenbanken zur Trennung von APK-Kollektionen anlegen kann.
Neben dem Importieren der APK-Informationen in die Datenbank, können die APKs zusammen mit ihren Metainformationen außerdem noch in die *MongoDB* kopiert werden[33], worauf in Kapitel 4.6.3 näher eingegangen wird.

4.6.2 MongoDB

Das Speichern von Daten in *MongoDB* erfolgt über *Dokumente*. Eine *Collection* fasst mehrere *Dokumente* in einer Datenbank zusammen.

[30] Ein Hash ist zwar nicht eindeutig, jedoch ist die Wahrscheinlichkeit für Kollisionen durch die Länge der SHA256-Hashfunktion sehr gering
[31] Siehe *androguard.core.bytecodes.apk.APK* sowie die dazugehörige Dokumentation unter `http://doc.androguard.re/html/apk.html#androguard.core.bytecodes.apk.APK`
[32] Das Experiment in Abbildung 6.4 zeigt den Performanz-Unterschied auf
[33] Die *MongoDB* wird dabei als verteilter APK-Speicher benutzt

Ein *Dokument* kann beliebig strukturierte Daten abspeichern. Abgespeichert wird es als Binary JSON (BSON), das ähnlich zu JSON ist, jedoch aus Effizienzgründen binär repräsentiert wird [TB11].
Das Fehlen eines festen Datenbankschemas ermöglicht die Speicherung der dynamischen Inhalte der Skript-Ergebnisse.

4.6.3 APK-Speicherung

Für die verteilte Analyse müssen außerdem die APKs verteilt werden. Dafür sendet der Client diese entweder mit oder die Arbeiter beziehen sie aus der *MongoDB*, indem nur die ID übertragen wird, mithilfe derer die Android-Applikation aus der Datenbank geladen werden kann.
Da *BSON-Dokumente* in *MongoDB* jedoch auf 16MB beschränkt sind, existiert die Spezifikation *GridFS*.
Durch Aufteilung der Datei in sogenannte *Chunks*, die unter diesem Limit liegen, wird die Beschränkung umgangen. Zu dem Zweck existieren zwei *Collections*. Die eine zur Speicherung der Metadaten, die andere für die *Chunks*. Durch Zusammenfügen der *Chunks* kann die ursprüngliche Datei rekonstruiert werden [monb].
ALL nutzt den Mechanismus einerseits zum Speichern der APKs, andererseits aber auch für große Analyseergebnisse. Jedoch wird ein zu großes Ergebnis nicht automatisch im *GridFS* abgelegt, da dadurch die Fähigkeit, Anfragen auf den Daten zu formulieren[34], verloren geht. Aus diesem Grund muss das Speichern in *GridFS* im *AndroScript* signalisiert werden (siehe 4.5.3).

4.6.4 Modellierung

Das Speichersystem ist in Abbildung 4.10 dargestellt. Dazu gehört zum einen die lokale Importdatenbank für die APK-Metainformationen (*ImportDatabaseStorage*). Zum anderen die Datenbank zur Speicherung der Analyseresultate (*ResultDatabaseStorage*) und Bereitstellung der APKs im verteilten Modus. Für die lokale Speicherung von Ergebnissen ist außerdem noch die Klasse *FileSysStorage* vorhanden.
Die verschiedenen Speicher implementieren dabei je nach Zweck ein oder mehrere Interfaces:

- *ApkCopyInterface*: Stellt Methoden zum Speichern und Empfangen der APKs bereit.
- *ImportQueryInterface*: Schnittstelle zum Abfragen der APK-Metadaten.
- *ImportStorageInterface*: Interface für das Importieren der APKs. Wird sowohl von *FileSysStorage* als auch *ImportDatabaseStorage* implementiert.
  Während erstere Klasse durch die implementierten Methoden das Anlegen der Verzeichnisstruktur für APK- sowie Ergebnisspeicherung erreicht, implementiert die *SQLite*-Datenbank dadurch das Importieren und Entfernen der APKs. Zusätzlich kann abgefragt werden, ob ein bestimmtes APK in der Datenbank vorliegt.
- *ResultWritingInterface* und *ResultStorageInterface* definieren den Umgang mit den Analyseresultaten. Ersteres beschreibt das Speichern der Ergebnisse während letzteres das Einsehen sowie Löschen erfasst.

[34] Daten in *GridFS* werden binär kodiert. Somit können nur noch Anfragen auf den Metadaten durchgeführt werden [monb]

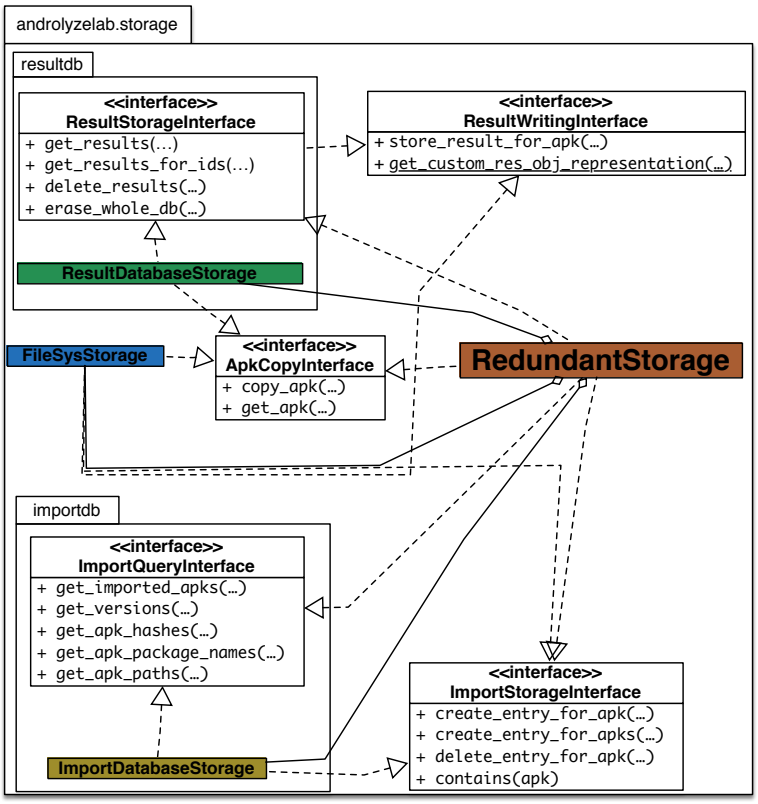

Abbildung 4.10: Architektur Storage System

Als übergeordnete Verwaltungseinheit dient die Klasse *RedundantStorage*, die jedes der aufgezählten Interfaces implementiert, indem es die durch die Schnittstellen definierte Funktionalität an die Klassen *ImportDatabaseStorage*, *ResultDatabaseStorage* und *FileSysStorage* delegiert.
Dabei wird jedoch erst auf Anfrage ein Objekt der jeweiligen Klasse instantiiert, da z.B. zum Abfragen der Importdatenbank, die Ergebnisdatenbank nicht benötigt wird. Zusätzlich wird das Importieren der jeweiligen Abhängigkeiten erst auf Anfrage eingeleitet. Folglich kann z.B. die Importdatenbank abgefragt werden, ohne, dass die Abhängigkeiten zur *MongoDB* installiert sind und der Datenbank-Service auch nicht gestartet sein muss.
Die Klasse *RedundantStorage* implementiert das redundante Speichern der Analyse-Resultate sowohl im lokalen Dateisystem als auch in der Ergebnis-Datenbank.
Zur Behandlung von Fehlern definiert ALL eine Vielzahl von Exceptions. Für jeden Fehlertyp, der bei der Speicherung auftreten kann, existiert eine entsprechende Exception. Somit kann speziell auf die Fehlertypen eingegangen werden und die jeweilige Fehlerbehandlung durchgeführt werden.

4.7 Analyse

Für die Analyse existieren drei verschiedene Implementierungen: eine nicht-parallele, eine parallelisierte Version mithilfe von *Threads* oder *Prozessen*[35] und eine verteilte Analyse, die Jobs an registrierte Arbeiter verteilt.
Die Basisklasse *BaseAnalyzer* abstrahiert die gemeinsame Funktionalität. Dafür stellt sie die Skripte und APKs sowie das Speichersystem zur Verfügung.

[35] Ob *Threads* oder *Prozesse* verwendet werden sollen kann vor der Analyse festgelegt werden

Damit die in Kapitel 4.5.1 beschriebenen Skript-Anforderungen die Performanz steigern können, muss der kleinste gemeinsame Nenner an Anforderungen gefunden werden, um nur die notwendigen Analyse-Objekte bereitzustellen.
Des Weiteren gibt die Oberklasse Feedback über den Fortschritt der Analyse. Die Fortschrittsdaten befinden sich dabei im *shared memory*, damit *Prozessen* von außen der Zugriff ermöglicht wird[36]. Zusätzlich werden die *identifier* der Analyseergebnisse zum Auffinden in der *MongoDB* nach außen präsentiert, um die Ergebnisse nach Beendigung der Analyse anzuzeigen.
Die APK-Daten werden über die *SQLite*-Datenbank bereitgestellt. Soll nicht das komplette Set an Applikationen analysiert werden, können durch Angabe von Paketnamen, Tags oder den Hashes der APKs, die Auswahl eingeschränkt werden. Die verschiedenen Analyseeinheiten werden im Folgenden näher erörtert.

4.7.1 Nicht parallel

Die nicht-parallele Analyse öffnet sequentiell die APKs und führt die Analyse mit den jeweiligen Skripten durch. Aus Effizienzgründen wird nicht für jedes Skript das APK erneut geöffnet. Tritt bei einem Skript ein Fehler auf, wird dieser auf *stderr* geloggt und mit dem nächsten Skript fortgefahren. Nach beendeter Analyse werden alle Ergebnisse gespeichert.

4.7.2 Parallel

Die parallele Abarbeitung wird durch den Einsatz von *Threads* oder *Prozessen* ermöglicht. Der *ParallelAnalyzer* verteilt mit *dynamic scheduling* die Analysejobs auf die Arbeiter. Jedem Arbeiter liegen dabei ebenfalls die Skripte und das Speichersystem vor. Die Jobs bestehen dabei lediglich aus den APK-Pfaden und sind via *shared memory* abrufbar. Damit die Worker nicht nach jeder Analyse für I/O bzw. Netzwerkaktivität unterbrechen müssen, haben sie zusätzlich einen Puffer fester Größe, der erst, wenn er vollständig gefüllt ist, die Ergebnisse speichert[37].
Die dynamische Lastverteilung wird dadurch erreicht, dass die Jobs nicht vorverteilt sind, sondern erst nach Bedarf aus dem gemeinsamen Speicher bezogen werden.

4.7.3 Verteilt

Die verteilte Analyse ist mithilfe des Frameworks *Celery* implementiert. Als *Nachrichten-Broker* wird *RabbitMQ* benutzt. Auch wenn *Celery* die Bibliothek *kombu* [celc], die zum Senden und Empfangen der Nachrichten zuständig ist, verwendet, und somit auch eine Reihe weiterer *Broker*[38] unterstützt, bietet RabbitMQ[39] im Gegensatz zu diesen den vollen Funktionsumfang von *Celery*.

[36] Die grafische Oberfläche z.B. startet die Analyse als eigenständigen *Prozess*, weshalb der gemeinsame Speicher notwendig ist

[37] Das Experiment 6.4.1 untersucht inwiefern sich die Performanz durch den Einsatz des Puffers steigern lässt

[38] Es werden u.A. *Redis*, *MongoDB*, *CouchDB*, *ZeroMQ* und *SQLAlchemy* als *Nachtrichten-Broker* unterstützt [celi]

[39] Redis bietet auch den vollen Funktionsumfang, allerdings wird *RabbitMQ* bzw. *AMPQ* aus Gründen der Performanz und Zuverlässigkeit empfohlen. Siehe `http://celery.readthedocs.org/en/latest/faq.html#do-i-have-to-use-amqp-rabbitmq`. Zuletzt eingesehen am 13.07.2014

4.7.3.1 Architektur

Um ein besseres Verständnis für die verteilte Analyse zu bekommen, muss zunächst dessen Architektur näher beschrieben werden. Diese ist in Abbildung 4.11 visualisiert.

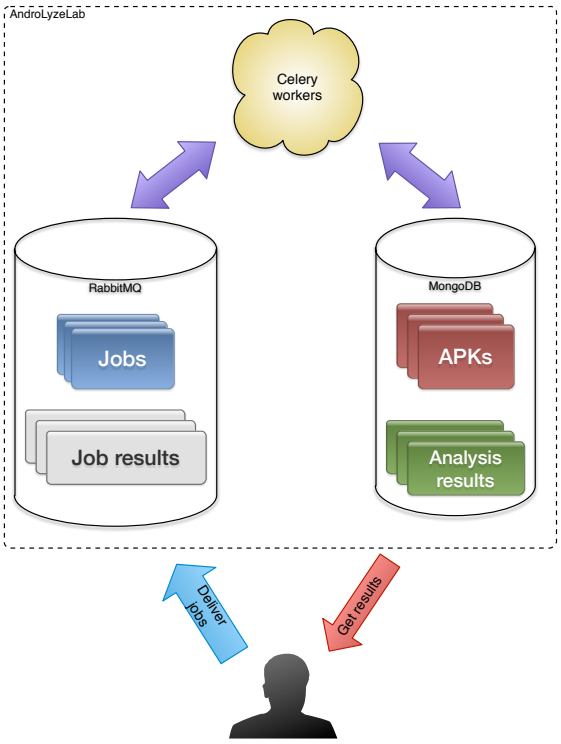

Abbildung 4.11: Architektur verteilte Analyse

Die *Analyse-Jobs* werden in das Warteschlangensystem *RabbitMQ* eingereiht. Von *Celery* bereitgestellte Worker empfangen diese und führen die Analyse durch.
Es stellt sich noch die Frage, woher die APKs bezogen werden. Dafür bietet ALL zwei verschiedene Modi an.
Zum einen können diese in die Datenbank *MongoDB* importiert werden. Dabei kann das Netzwerk zum Flaschenhals werden, wenn die APKs langsamer übermittelt werden, als die Worker die Analyse durchführen können. Das Problem zeigt sich mit wachsender Zahl von Arbeitern. Mit *sharding* [mond], d.h. der Verteilung der Daten auf mehrere Knoten, wird jedoch eine horizontale Skalierung erreicht und somit können Daten von mehreren Instanzen übertragen werden.
Zum anderen können die Apps auch einfach in die Nachricht integriert werden, die in der Warteschlange gespeichert wird. Auch bei diesem Szenario kann *Skalierbarkeit* durch Clustering [rab] erreicht werden, indem Daten zwischen mehreren *RabbitMQ*-Knoten geteilt werden. Trotz dieser Skalierung, kann jedoch das eigentliche Senden der Jobs (vom Client) und somit auch der APK-Daten, zum Flaschenhals werden.
Nichtsdestotrotz unterstützt ALL dieses Szenario, da nicht immer Skalierbarkeit und Effizienz im Vordergrund stehen. Möchte man komfortabel (d.h. ohne vorheriges Importieren der APKs) eine Analyse auf einem evtl. nicht so großen APK-Set durchführen, bietet sich dieses Szenario an.
Sind die Daten verteilt in der *MongoDB*, wird nur die ID der Android-Applikation übermittelt. Andernfalls die Rohdaten des APK.
Nach erfolgreichem Empfangen des Jobs, führen die *Celery*-Worker die Analyse durch

und speichern die Ergebnisse in der *MongoDB*. Zusätzlich wird ein Ergebnistupel pro Skript in einer speziellen Warteschlange, die für die Speicherung der Job-Ergebnisse zuständig ist, abgelegt. Bei dem Tupel handelt es sich um die *ID* des Ergebnisses sowie den Hinweis, ob das Resultat im *GridFS* vorliegt.
Der Client, der die Analyse initiiert, hat einen *Callback-Handler* registriert, der für jedes APK die erfolgreiche oder fehlgeschlagene Analyse signalisiert. Im Fehlerfall wird dem Benutzer der Fehler und der *Exception-Traceback* präsentiert. Im Erfolgsfall wird das Ergebnis der Analyse[40] aus der *MongoDB* geladen, das durch den Ergebnistupel eindeutig identifiziert ist.

4.7.3.2 Nachrichtenformat

Das Format, das ALL für die Nachrichten benutzt ist in Abbildung 4.12 dargestellt. Nachrichten müssen serialisiert werden. Das bedeutet, dass sie in ein Format konvertiert werden, das alle Parteien verstehen. *Celery* bietet dafür: *Pickle*, *JSON*, *MsgPack* etc. an. Es können außerdem eigene Serialisierer registriert werden.
Während *Pickle* zum Serialisieren von Python-Code benutzt werden kann, unterstützen

Argument:	Skripte	Skript Hashes	APK / APK-ID	is_hash	APK-Metadaten
Typ:	list<str>	list<str>	str	bool	FastApk

Abbildung 4.12: Nachrichtenformat AnalyzeTask

die beiden anderen dies nicht[41].
Die Abbildung zeigt, dass zunächst die Skripte übertragen werden. Auch wenn mit *Pickle* die Python-Skripte einfach serialisiert werden könnten, erfolgt dies nicht. Das hat den Grund, dass Skripte auf dem Worker importierbar sein müssen [pytc]. Kann ein Skript nicht importiert werden, wird zwar eine Exception ausgelöst, dass das Skript nicht deserialisiert werden konnte, jedoch wird dieser Fehler nicht an den Client, der die Analyse initiiert hat, propagiert.
Um dem Analysten ein besseres Feedback für diese Situation zu ermöglichen, wird lediglich der Paketname des Skriptes übertragen, sodass das Skript manuell importiert werden und im Falle eines Importfehlers dem Client ein Fehler signalisiert werden kann.
Wie bereits erwähnt, gibt es zwei Möglichkeiten um die APKs an die Arbeiter zu verteilen. Entweder werden die Rohdaten der *.apk*-Datei serialisiert oder nur der *identifier* zum Auffinden in der *MongoDB* übermittelt.
Das Feld *is_hash* beschreibt, welche Möglichkeit benutzt wird. Zusätzlich werden im Falle der APK-Serialisierung noch die Metadaten durch Serialisieren der Python-Klasse *FastApk* mitgesendet.

4.7.3.3 Fehlertoleranz

In verteilten Systemen muss man immer mit dem Auftreten von Fehlern rechnen. Fehler können dabei durch unzuverlässige Übertragungsmedien oder durch die Komplexität verteilter Systeme entstehen, indem beispielsweise einzelne Komponenten ausfallen.
Um dem zu entgegnen, implementiert ALL mithilfe von *Celery* Fehlertoleranz[42]. Feh-

[40] Nur falls der Benutzer die lokale Speicherung der Ergebnisse aktiviert hat

[41] Siehe `http://celery.readthedocs.org/en/latest/userguide/calling.html#serializers`. Zuletzt eingesehen am 13.07.2014

[42] Fehlertoleranz in Verbindung mit dem *RPC-Backend* für die Ergebnisspeicherung kann erst in künftigen Celery-Versionen benutzt werden, da noch ein Fehler in dem Framework gefunden wurde. Siehe

ler können mit den von Programmiersprachen gewohnten Konstrukten zur Fehlerbehandlung abgefangen werden und bei bestimmten Fehlern signalisiert werden, dass das erneute Ausführen des Jobs versucht werden soll.
Dabei können die Anzahl der Versuche sowie der Zeitpunkt gesteuert werden, an dem der Neuversuch unternommen werden soll [celf]. Die Parameter sind konfigurierbar und in Abschnitt 4.10 erläutert.
Die folgenden Fehler werden durch ALL behandelt:

- Broker Netzwerkfehler: Treten Fehler beim Publizieren der Jobs auf, wird dies solange versucht, bis die Aufgaben erfolgreich in der Warteschlange registriert sind[43].
- MongoDB Netzwerkfehler: Das Speichern der Ergebnisse in der Datenbank und ggf. dem Laden des APK aus der *MongoDB* kann erneut versucht werden.
- Workerausfall/Fehler: Stürzt ein Worker-*Prozess* ab oder tritt ein Systemausfall ein, ist der Job immer noch in der Warteschlange vorhanden und kann an einen anderen Knoten verteilt werden [celc].
- Skript-Import-Fehler: Sind die für die Analyse benötigten Skripte nicht importierbar, da sie auf einem Worker nicht deployed wurden, könnten sie evtl. auf einem anderen Knoten vorliegen.

Abbildung 4.13 zeigt den Ablauf der Analyse im verteilten Modus sowie die Implementierung der Fehlertoleranz. Das erneute Ausführen von Jobs wird dabei zeitlich durch einen *exponential backoff* bestimmt. Bei erneutem Scheitern des Jobs, wird bis zu einer, je nach Fehler festgelegten Obergrenze, die Wartezeit exponentiell erhöht[44].

4.7.4 Skript Deployment & Cluster management

Damit die verteilte Analyse überhaupt erst möglich ist, muss zunächst das Cluster von *Celery*-Workern eingerichtet und gestartet werden. Außerdem benötigen die Arbeiter den Code sowie die Skripte für die Analyse. Folglich muss beides vorher deployed werden.
Für all diese Zwecke bietet ALL mithilfe der Bibliothek *fabric*[45] eine komfortable Steuerung an. Die Hilfsfunktionen, die dafür zur Verfügung stehen, sind u.A:

- Worker Setup:
  - Initial: Die Worker müssen evtl. einmalig eingerichtet werden. Das bedeutet, dass ein eigener Benutzer angelegt wird[46], um die Rechte der Worker aus Sicherheitsgründen einzuschränken und u.A. Python installiert werden muss[47].

`https://github.com/celery/celery/issues/2113`. Mit früheren Versionen führt das Einschalten der Fehlertoleranz dazu, dass der Analyst kein Feedback darüber bekommt, wann die Analyse beendet ist.

[43] Siehe `http://celery.readthedocs.org/en/latest/userguide/calling.html#retry-policy`. Zuletzt eingesehen am 13.07.2014

[44] Die Steuerung der Wartezeit bis zum Neuversuch eines Jobs beginnt bei 2^0, wobei bei jedem Neuversuch der Exponent um eins erhöht wird, bis die Obergrenze erreicht wird. Dann bleibt die Wartezeit konstant

[45] Bibiliothek zur komfortablen und vereinfachten Nutzung von *SSH*-Funktionalität via Python-API. Siehe `http://www.fabfile.org`

[46] Das Erstellen eines extra Benutzers ist optional. Die Worker können unter jedem Benutzer ausgeführt werden

[47] Wird nur für Systeme mit der Paketverwaltung *Aptitude* unterstützt. Für den Vorgang werden Root-Rechte benötigt

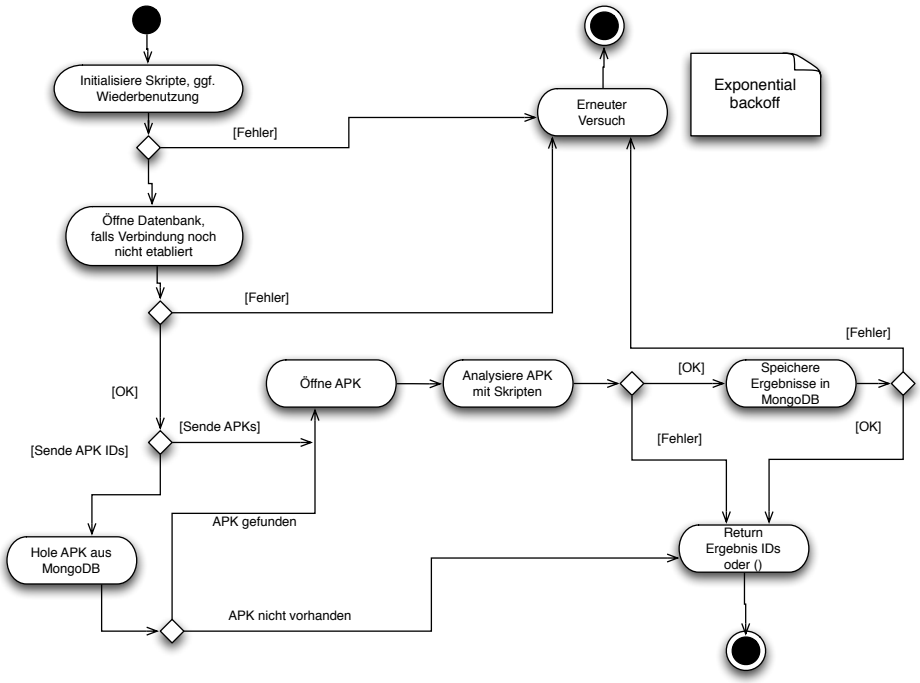

Abbildung 4.13: Aktivitätsdiagramm verteilte Analyse

  - Python-Umgebung: Zusätzlich müssen noch diverse Python Frameworks wie *Celery*, die Bindings für *MongoDB* etc. installiert werden.

- Deployment:

  - Code: Zum Übertragen des Quellcodes gibt es zwei Möglichkeiten. Für den offiziellen Code aus dem verwendeten *Git-Repository* wird der Updatemechanismus von *git* verwendet. Sowohl Benutzer/Passwort als auch Public-Key Authentifizierung wird unterstützt. Für das Deployment von Testingcode kann dieser einfach via *rsync* auf die anderen Knoten synchronisiert werden.
  - Skripte: Die Skripte werden ebenfalls mithilfe von *rsync* mit den anderen Knoten synchronisiert. Dabei können sowohl einzelne Dateien als auch komplette Ordner deployed werden.

- Worker management: Nicht fehlen darf außerdem ein Mechanismus zum Starten, Stoppen bzw. Neustarten der Worker. Den entsprechenden Kommandos kann zusätzlich noch die Anzahl der *Prozesse*, die gestartet werden sollen, mitgeteilt werden. Außerdem existiert eine *Autoscaling*-Option. Je nach Last kann der Prozesspool dynamisch vergrößert bzw. verkleinert werden, indem eine Unter- sowie Obergrenze definiert wird.

Damit *fabric* weiß, zu welchen Knoten es via *SSH* verbinden soll, müssen zunächst die Adressen der Knoten[48] in der verteilten Konfiguration hinterlegt werden. Auf die Konfiguration von ALL wird in 4.10 genauer eingegangen.

[48] ALL setzt vorraus, dass auf den Knoten bereits der *SSH*-Dienst gestartet ist. Die Authentifikation kann dabei entweder über die Kombination von Benutzernamen und Passwort sowie über *Public-Key-Authentifizierung* erfolgen und beim Ausführen des *fab*-Kommandos mitgeteilt werden

4.8 Sicherheit

Bei der Entwicklung von ALL wurde viel Wert auf Sicherheit gelegt. *MongoDB* und *RabbitMQ* können zwar ohne Authentisierung benutzt werden, jedoch ist es ratsam auf die Authentisierung via Benutzername sowie Passwort umzustellen.
Die Benutzung von *Pickle* ermöglicht die *Serialisierung* von Python-Code, jedoch kann dies unter Umständen auch zu Sicherheitsrisiken führen, da dieser von den *Celery*-Workern benutzt wird.
Verschlüsselung kann durch den Einsatz von *X.509* Zertifikaten, also der Benutzung einer Public Key Infrastructure (PKI), aktiviert werden[49]. Des Weiteren ist es möglich, die Authentisierung des Clients gegenüber des *Nachrichten-Broker* zu aktivieren. Somit können nur authentisierte Clients *Python-Code senden*.
Zusätzlich existiert eine optionale Validierung der Skript-Hashes. Der Analyst sendet zusammen mit den Skripten deren Hashes mit. Stimmen diese nicht mit den Hashes überein, die der Worker erhält, wenn er die Skripte lokal aus dem Dateisystem lädt, wird die Analyse abgebrochen. Das Experiment in Kapitel 6, Abbildung 6.5.1 zeigt außerdem, wie sich die Verschlüsselung auf die Performanz auswirkt.

4.9 Play Store Crawling

Ein noch so mächtiges Analysewerkzeug ist nutzlos ohne Daten, die analysiert werden können. Um dieses Problem zu adressieren, benutzt ALL das Framework *Google Play Crawler* [Akd], mit dem APKs aus dem *Play Store* heruntergeladen werden können. Zusätzlich ermöglicht es das Auflisten der Kategorien[50], die im Store vertreten sind. Somit lassen sich die neuesten sowie beliebtesten Applikationen in den verschiedenen Kategorien auflisten.
Um komfortabel z.B. die beliebtesten APKs aus einer bestimmten oder allen Kategorien herunterzuladen, stellt ALL einen Rahmen von Hilfsfunktionen um das Java-Programm in Form eines Python-Skriptes bereit. Es können außerdem einzelne Applikationen über den Paketnamen heruntergeladen und die gesamte Datenbanken aktualisiert werden. Gebunden an eine Importdatenbank wird für jeden Paketnamen aus der Datenbank zunächst überprüft ob eine aktuelle Version im *Play Store* vorliegt und nur dann aktualisiert. Um ein Blockieren der IP-Adresse zu verhindern, sind die Downloads nicht parallelisiert und mit einer Zeitverzögerung versehen.

4.10 Konfiguration

ALL ist durch die Bereitstellung von Konfigurationsdateien, die den Bedürfnissen des Analysten angepasst werden können, sehr flexibel.
Gerade das *verteilte System* benötigt einen gewissen Konfigurationsaufwand. Beispielsweise müssen die Adressen, Ports, Benutzernamen und Passwörter für *MongoDB* und *RabbitMQ* definiert werden.
Da ALL für die Analyse das Framework *androguard* benutzt, muss sichergestellt wer-

[49] Die Dokumentation von ALL beschreibt wie Benutzer sowohl für *MongoDB* als auch *RabbitMQ* mit einem Passwort erstellt werden können. Außerdem wird das Einrichten der *SSL*-Konfiguration für beide erklärt einschließlich des Erstellen von *X.509* Zertifikaten. Die Dokumentation ist als CD dieser Thesis beigefügt

[50] Das Auflisten von Applikation ist durch den *Play Store* auf 500 Elemente beschränkt [VGN14]

den, dass zur Laufzeit das Framework auch verfügbar ist, damit dessen Funktionalität verwendet werden kann.
Des Weiteren muss das Wurzelverzeichnis für die Speicherung der APKs sowie Analyseresultate spezifiziert werden. Die lokale Speicherung ist jedoch auch deaktivierbar, sodass die Resultate nur noch in der Ergebnisdatenbank abgelegt werden. Damit zu einem späteren Zeitpunkt dennoch lokal auf die Ergebnisse zugegriffen werden kann, bietet ALL eine Synchronisierung mit der Ergebnisdatenbank.
Außerdem kann die Anzahl der *Threads* bzw. *Prozesse*, die zur Analyse verwendet werden sowie der Analysemodus (nicht parallel, parallel oder verteilt), benutzerdefiniert werden.
Für die Verwendung des verteilten Systems stehen überdies Optionen zur Einstellung der Fehlertoleranz sowie Deployment zur Verfügung. Jobs, die evtl. einen zu langen Zeitraum zur Abarbeitung beanspruchen, können durch Zeitlimits in der Ausführungszeit beschränkt werden.
Konfigurationsdateien sind nach Sektionen gegliedert, in denen mehrere Werte stehen können, die jeweils einem Schlüssel zugeordnet sind. Alle Schlüssel und Sektionen sind in dem Modul *androlyzelab.settings* vorhanden.
Die Konfiguration ist in mehrere Dateien aufgeteilt und in Abbildung 4.14 dargestellt.

```
/
├── ~/.all
│   ├── script_settings.json
│   ├── config.conf
│   └── crawler.conf
└── androlyzelab
    └── conf
        └── distributed
            ├── distributed.conf
            └── ssl
```

Abbildung 4.14: Konfigurationsdateien

Konfigurationsdaten für das verteilte System befinden sich im Projektverzeichnis unter *conf/distributed*. Die Datei *distributed.conf* stellt die für die Worker notwendigen Informationen bereit. Zusätzlich beinhaltet sie die für das Management via *fabric* benötigten Informationen. Der Ordner *ssl* ist leer und kann für die Speicherung der *X.509* Zertifikate und Schlüssel benutzt werden.
Im Gegensatz zu der verteilten Konfiguration werden alle anderen Konfigurationsdateien im Homeverzeichnis des aktuellen Benutzers, genauer unter *~/.all/* gespeichert. Somit existiert für evtl. mehrere ALL-Versionen genau ein Konfigurationsset. Die Konfigurationsdatei lautet *config.conf*.
Damit komfortabel vordefinierte Skriptsets geladen werden können, ermöglicht die *JSON*-Datei *script_settings.json* die Definition verschiedener Skriptsets. Sets sind mit einem Schlüssel verbunden, sodass über diesen Schlüssel ein Skriptset festgelegt werden kann, das standardmäßig geladen wird.
Zum Herunterladen von APKs aus dem *Play Store* mithilfe des *Google Play Crawler* müssen Benutzername und Passwort eines zum *Play Store* verbundenen Google Account hinterlegt[51] werden. Für die Bereitstellung dieser Informationen kann die Datei *crawler.conf* benutzt werden.
Für alle Konfigurationen existieren Standardeinstellungen, die geladen werden, wenn

[51] Neben der Bereitstellung des *Google Play* Accounts muss initial ein Checkin mit dem Kommando *"java -jar googleplaycrawler.jar checkin"* ausgeführt werden. Die zurückgelieferte Android-ID muss ebenso in der Konfigurationsdatei hinterlegt werden

keine Konfigurationsdatei durch den User vorhanden ist.

4.11 UI

ALL bietet sowohl ein Command Line Interface (CLI) als auch ein Graphical User Interface (GUI) an. Die gesamte Funktionalität von ALL ist dabei über das Root-Package verfügbar, wodurch eine API nach außen propagiert wird.
Die Kommandozeilenversion kann zur Automatisierung benutzt werden, wohingegen die GUI einen höheren Bedienkomfort bietet.
Abbildung 4.15 zeigt eine laufende Analyse in der grafischen Oberfläche. Nachdem

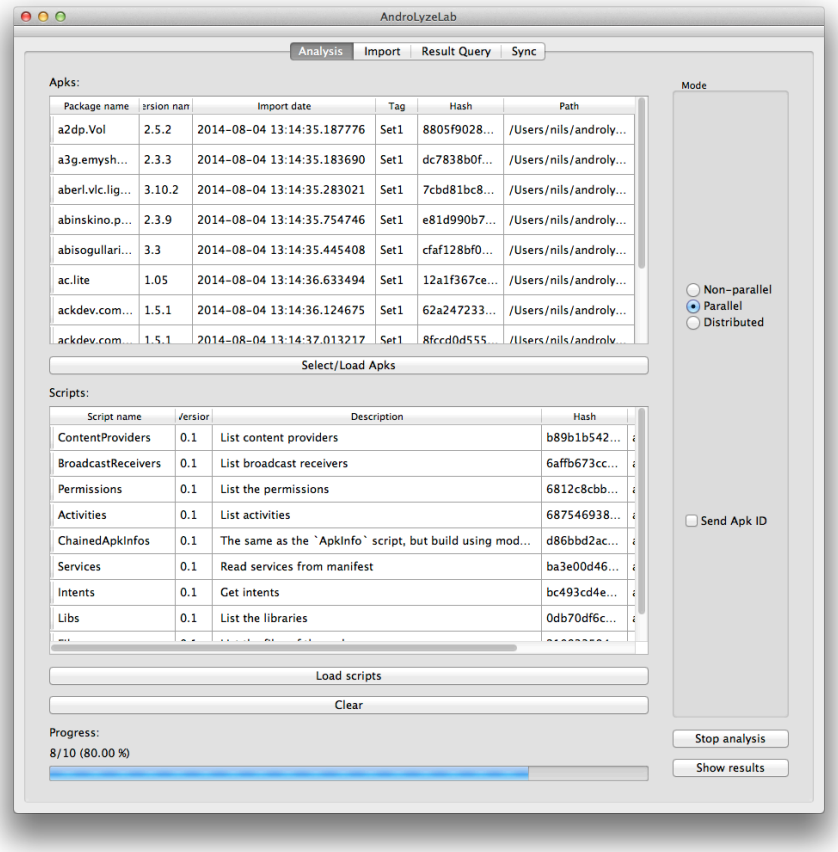

Abbildung 4.15: Grafische Oberfläche AndroLyzeLab

die Analyse erfolgreich beendet wurde, können die Ergebnisse in einem Dialog bequem erforscht werden (siehe Abbildung 4.16). Der Tab *Result Query* hilft bei der Formulierung komplexer Abfragen und Analysen der Resultate. Reichen die Optionen der GUI nicht aus, können außerdem Anfragen direkt in der Syntax von *MongoDB* formuliert werden. Die Ergebnisansicht basiert auf einem Datenbank-*Cursor*, sodass weitere Ergebnisse erst auf Anforderung angezeigt werden, denn die vollständige Anzeige der Ergebnisse ist bedingt durch die Größe nicht möglich.
Für ein detailliertes Feedback werden die Python-Loggingaufrufe in eine grafische Komponente weitergeleitet. Der Detaillgrad ist ebenfalls einstellbar.

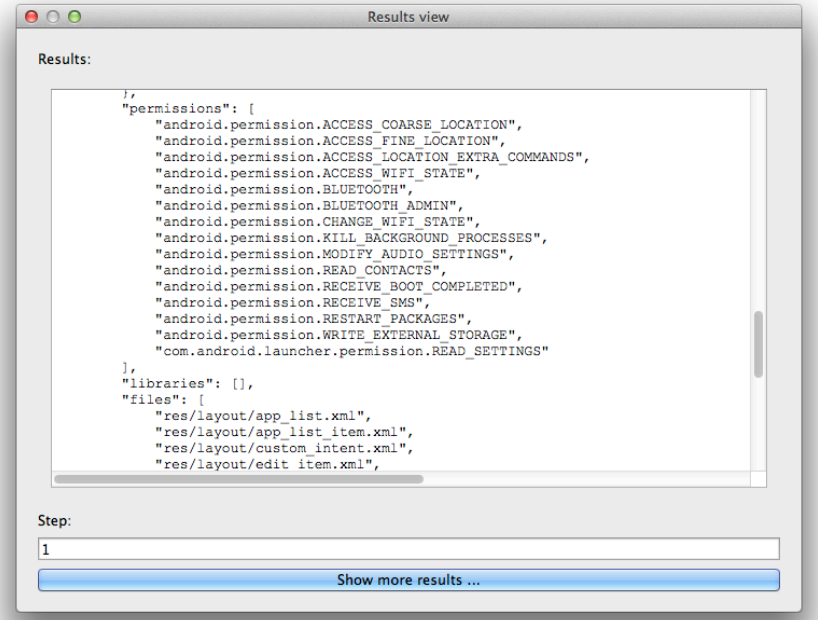

Abbildung 4.16: Ergebnisansicht

5 Implementierung

Nachdem im letzten Kapitel das Design von ALL vorgestellt wurde, wird im Folgenden auf die Details der Implementierung näher eingegangen.
ALL ist bedingt durch *androguard* in *Python* geschrieben. Als Entwicklungsumgebung wurde LiClipse[52] verwendet. Dabei handelt es sich um ein Paket, das basierend auf *Eclipse* Python Unterstützung durch das *PyDev*-Plugin bietet.
Nachfolgend wird zunächst das Logging der Ergebnisse näher von der Implementierungsseite beleuchtet. Danach folgt das Skript-Framework, die Implementierung sowohl der parallelen als auch verteilten Analyse und abschließend das Speichersystem.
Aus Übersichtsgründen werden Logging sowie Kommentare im Quellcode teilweise nicht gezeigt. Außerdem wurden zum Teil größere Codeabschnitte auf mehrere Abbildungen aufgeteilt.

5.1 Logging Ergebnisse

Das Speichern von Resultaten ist mit der Klasse *ResultObject* modelliert. Geknüpft an ein *Apk*-Objekt hält es die Metainformationen der Android-Applikation sowie des *AndroScript* bereit.
Die Ergebnisspeicherung wird implementiert, indem auf die Funktionalität der von Python bereitgestellten Klasse *OrderedDict* zurückgegriffen wird. Dabei handelt es sich um ein *Dictionary*, das die Einfügereihenfolge beibehält. Somit können Werte unter eindeutigen Schlüsseln gespeichert werden.
Strukturierung der Ergebnisse wird erreicht, indem beim Loggen eine Liste von Kategorien angegeben werden kann. Somit wird eine beliebige Verschachtelung innerhalb des *Dictionary* erreicht.

Code Listing 5.1: Logging

```
1 def __log(self, key, value, func, *categories, **kwargs):
2   register_key = kwargs.get("register_key", False)
3
4   if None in categories:
5     raise ValueError("You supplied an empty category: %s" % ', '.join([
          str(x) for x in categories]))
6
```

[52] Siehe http://brainwy.github.io/liclipse/index.html

```
7   def log2(_dict, key, value, *sub_categories):
8     cnt_categories = len(sub_categories)
9     if cnt_categories >= 1:
10      category = sub_categories[0]
11      category_name = str(category)
12      category_in_dict = category_name in _dict
13      category_val_none = category_in_dict and _dict[category_name] is
            None
14
15      if category_val_none or not category_in_dict:
16        self.__check_n_set_value_for_key(_dict, category_name,
              OrderedDict(), self.__dict_assignment, register_key, *
              categories)
17
18      sub_dict = _dict[category_name]
19      if cnt_categories > 1:
20        log2(sub_dict, key, value, *sub_categories[1:])
21      else:
22        self.__check_n_set_value_for_key(sub_dict, key, value, func,
              register_key, *categories)
23    else:
24      self.__check_n_set_value_for_key(_dict, key, value, func,
            register_key, *categories)
25
26  log2(self, key, value, *categories)
```

Die gesamte Funktionalität wird durch die in Abbildung 5.1 dargestellte Methode zur Verfügung gestellt. Sie wird sowohl zum Loggen von Werten sowie der Registrierung der Ergebnisstruktur genutzt. Prinzipiell muss dabei zwischen zwei Situationen unterschieden werden. Entweder wird direkt ein Wert in dem *Dictionary* gesetzt oder es wird ein Wert einer Liste hinzugefügt. Die Implementierungen der beiden Arten von Wertzuweisungen sind in den Abbildungen 5.2 und 5.3 dargestellt.
Basierend auf den bisher vorgestellten Funktionen können die Methoden zum Registrieren und Loggen implementiert werden. Abbildung 5.4 und 5.5 zeigen beispielsweise das Registrieren einer leeren Liste sowie das Hinzufügen von Werten in diese.

Code Listing 5.2: Dictionary Zuweisung

```
1 @staticmethod
2 def __dict_assignment(_dict, key, val):
3     _dict[key] = val
```

Code Listing 5.3: Hinzufügen eines Elementes in Aufzählung (gespeichert in Dictionary)

```
1 @staticmethod
2 def __dict_list_append(_dict, key, val):
3     _dict[key].append(val)
```

Die *__log* Methode aus Abbildung 5.1 funktioniert dabei wie folgt: Je nach Kategorietiefe wird entweder direkt der Wert geloggt oder es muss rekursiv weiter in die Struktur des *Dictionary* hineingegangen werden.
Sind mehrere Kategorien angegeben wird bei der Registrierung der Struktur auf jeder, außer der letzten Ebene, ein *OrderedDict* gespeichert. Somit wird die beliebige Strukturierung bzw. Verschachtelung der Ergebnisstruktur ermöglicht.
Die Methode *__check_n_set_value_for_key* führt dabei das Registrieren/Logging aus, indem die übermittelte Funktion (z.B. 5.2 oder 5.3) ausgeführt wird. Vorher wird jedoch überprüft, ob die Kategorie bereits registriert wurde, zumindest im Logging-Fall. Jedes *AndroScript* ist standardmäßig mit einem *ResultObject* ausgestattet, damit die

Metainformationen von Skript und APK im Ergebnis festgehalten werden. Es muss nicht, aber kann durch den Analysten zur Ergebnisspeicherung benutzt werden. Alternativ kann z.B. auch ein String benutzt werden, damit reiner Text als ".txt" abgespeichert wird.

Code Listing 5.4: Registrierung einer Aufzählung

```
1 def register_enum_keys(self, keys, *categories):
2     for key in keys:
3         self.__log(key, [], self.__dict_assignment, *categories,
              register_key = True)
```

Code Listing 5.5: Logging eines Wertes in eine Aufzählung

```
1 def log_append_to_enum(self, key, value, *categories):
2     self.__log(key, value, self.__dict_list_append, *categories)
```

5.2 AndroScript

Skriptoptionen sowie Anforderungen wurden bereits im Designkapitel vorgestellt. Jedoch fehlt noch die Analysemethode.

Code Listing 5.6: AndroScript

```
1  def analyze(self, apk, dalvik_vm_format, vm_analysis, gvm_analysis, *
       args, **kwargs):
2      res = self.res
3      self._log_script_meta_before_act_run(res)
4
5      # analyze and measure time
6      time_s = timeit(self._analyze,
7                      *((apk, dalvik_vm_format, vm_analysis, gvm_analysis
                           ) +  args),
8                      **kwargs)
9
10     if self.create_script_stats():
11         self._log_script_meta_after_act_run(res, time_s)
12
13     return self.res
```

Die öffentliche Methode *analyze*, die in Abbildung 5.6 dargestellt ist, zeigt die Kapselung um die *_analyze* Methode, die in Zeile 6 aufgerufen wird. Die gesamte Analysefunktionalität muss dort implementiert werden. Neben dem eigentlichen Aufruf werden noch die Metainformationen des *AndroScript* gespeichert. Dafür wird in Zeile 2 das *ResultObject* des Skriptes abgerufen. In der darauffolgenden Zeile werden Skript-Metainformationen wie Name, Hash und Version geloggt.
Sind Skript-Statistiken aktiviert, wird die Ausführungszeit der *_analyze* Methode gemessen und gespeichert (Zeile 11). Außerdem sind die Analyse-Objekte in der Methodensignatur zu sehen, die an *_analyze* weitergereicht werden.
Das Bündeln von Skripten mithilfe der Klasse *ChainedScript* funktioniert auf ähnliche Weise. Sequentiell wird die *_analyze* Methode der Skripte aufgerufen, die das *ChainedScript* zusammenfasst. Tritt bei einem Skript ein Fehler auf, wird die Analyse nur abgebrochen, wenn *continue_on_script_failure* True zurückliefert. Ist dies nicht der Fall kann durch Überschreiben von *log_script_failure_exception* die Exception im Resultat niedergeschrieben werden. Der Code ist allerdings nicht abgebildet, da dieser sehr umfangreich ist.

5.3 Analyse

5.3.1 Parallel

Die parallele Analyse ist durch Lösen des *Erzeuger-Verbraucher-Problems* implementiert und in Abbildung 5.7 nachvollziehbar. Der *ParallelAnalyzer* hat die Pfade zu den APKs, die Skripte und deren minimale Anforderungen, die zur Analyse benutzt werden sollen. Außerdem verfügt der *Analyzer* über eine Instanz des *RedundantStorage*-Objekts zur Speicherung der Ergebnisse.
Somit kann die Arbeit an Worker verteilt werden, die die eigentliche Analyse durchführen.
Die Skripte, die minimalen Anforderungen und das Storage-Objekt etc. werden bereits bei der Initialisierung der Arbeiter übermittelt, da diese nur einmalig benötigt werden und sich nicht ändern (Zeile 6).

Code Listing 5.7: ParallelAnalyzer

```
1  def _analyze(self):
2    try:
3      work_queue = self.work_queue
4
5      for _ in range(self.concurrency):
6        p = Worker.create_worker(self.threaded)(self.script_list, self.
             script_hashes, self.min_script_needs, work_queue, self.
             storage, self.queue_size, self.analyzed_apks, self.
             storage_results)
7        p.daemon = True
8        p.start()
9
10     for apk_stuff in AnalyzeUtil.apk_gen(self.apks_or_paths):
11       work_queue.put(apk_stuff)
12
13     for _ in range(self.concurrency):
14       work_queue.put(STOP_SENTINEL)
15
16     work_queue.join()
17
18     return self.analyzed_apks.value
19   except KeyboardInterrupt:
20     ...
```

Um eine dynamische Arbeitsverteilung (*dynamic scheduling*) je nach Last zu erreichen, werden die Jobs mithilfe einer Warteschlange (genauer *JoinableQueue*), die die Jobs über *shared memory* den Workern zur Verfügung stellt, verteilt. Außerdem synchronisiert sie den Zugriff und verhindert, dass der gleiche Job mehrfach abgerufen werden kann.
Die Methode *create_worker* erstellt die Arbeiter entweder als *Threads* oder *Prozesse*. Dies wird erreicht, indem dynamisch mithilfe von *Closures* zur Laufzeit die Oberklasse definiert wird. Standardmäßig wird für jede CPU bzw. jeden Kern ein *Thread*/*Prozess* gestartet, falls nicht anders durch den Benutzer vorgegeben.
Um den Arbeitern zu signalisieren, dass alle Jobs bereits erzeugt wurden, wird für jeden Arbeiter eine Stringkonstante in die Warteschlange eingereiht (Zeile 13-14). Ansonsten würden die *Threads*/*Prozesse* blockieren, da sie auf das Eintreffen neuer Arbeit warten.
Der *ParallelAnalyzer* wartet durch die Operation *join* auf der *JoinableQueue* darauf,

dass alle Jobs abgearbeitet werden. Über *shared memory* wird außerdem die Anzahl der analysierten APKs gezählt sowie die Ergebnistupel abgelegt, die die Analyseergebnisse in der Datenbank identifizieren.
Nachdem die Arbeitsweise des Erzeugers erklärt wurde, fehlt noch das Vorgehen der Verbrauchers. Diese beziehen den Pfad zum APK sowie ggf. Metainformationen aus der Arbeits-Warteschlange. Letzteres wird in Form eines *FastApk*-Objektes transportiert, das die Metainformationen aus der Import-Datenbank wie z.B. das Importdatum trägt. Zumindest, wenn vor der Analyse ein Import der APKs stattgefunden hat. Beide Elemente sind im Tupel *apk_stuff* abgespeichert (Zeile 11).
Der Arbeitsablauf der Arbeiter ist in Abbildung 5.8 zu sehen. Nachdem die Arbeitsanweisungen aus der *Queue* bezogen wurden, wird zunächst das *EAndroApk*-Objekt geladen, also das APK im Speicher repräsentiert (Zeile 8). Außerdem ist der Puffer für die Ergebnisse zu sehen. Ist dieser gefüllt, werden die Ergebnisse gespeichert und der Puffer geleert (Zeile 10-13).

Code Listing 5.8: Arbeitsablauf Worker

```
1  def run(self):
2    work_queue = self.work_queue
3    n = self.queue_size
4    cnt = 0
5    try:
6      for work in iter(work_queue.get, STOP_SENTINEL):
7        apk_path, _apk, _ = work
8        eandro_apk = AnalyzeUtil.open_apk(apk_path, apk = _apk)
9
10       if cnt == n:
11         self.__store_results(self.partial_results)
12         cnt = 0
13         self.partial_results = []
14
15       res = self.analyze_apk(eandro_apk, scripts = scripts)
16
17       if res is not None:
18         cnt += 1
19         self.partial_results.append(res)
20       else:
21         self.add_analyzed_apks_sm(1)
22
23       work_queue.task_done()
24
25     self.__store_results(self.partial_results)
26     work_queue.task_done()
27     if not threaded:
28       work_queue.close()
29   except KeyboardInterrupt:
30     ...
```

Anschließend werden die benötigten Analyse-Objekte durch *androguard* bereitgestellt und das APK mit allen *AndroScript*en analysiert. Im Erfolgsfall befinden sich in *res* die Ergebnisse. Andernfalls wurde ein Fehler auf *stderr* geloggt.
Die Ergebnisse werden anschließend in den Puffer eingefügt, der, wenn er geleert wird die Anzahl der analysierten APKs erhöht. Konnte die Applikation nicht geöffnet oder analysiert werden, wird lediglich der Zähler erhöht (Zeile 21).
Nach Beenden eines Jobs wird über die Methode *task_done* der *JoinableQueue* signalisiert, dass ein Task ausgeführt wurde. Intern besitzt diese einen Zähler, der, wenn

er mit der Anzahl der erzeugten Jobs übereinstimmt, den *ParallelAnalyzer* aus dem blockierten Status befreit.
Wird die Analyse vorzeitig durch den Benutzer abgebrochen, werden die Ergebnisse aus dem Puffer noch gespeichert.

5.3.2 Verteilt

Die verteilte Analyse besteht aus zwei wichtigen Klassen. Zum einen handelt es sich dabei um die Klasse *DistributedAnalyzer*, die die Initiierung des Jobs auf Clientseite beschreibt. Zum anderen die Klasse *AnalyzeTask*, die die Arbeitsanweisungen für die Arbeiter bereitstellt. Beide sind im UML-Diagramm in Abbildung 5.1 dargestellt.

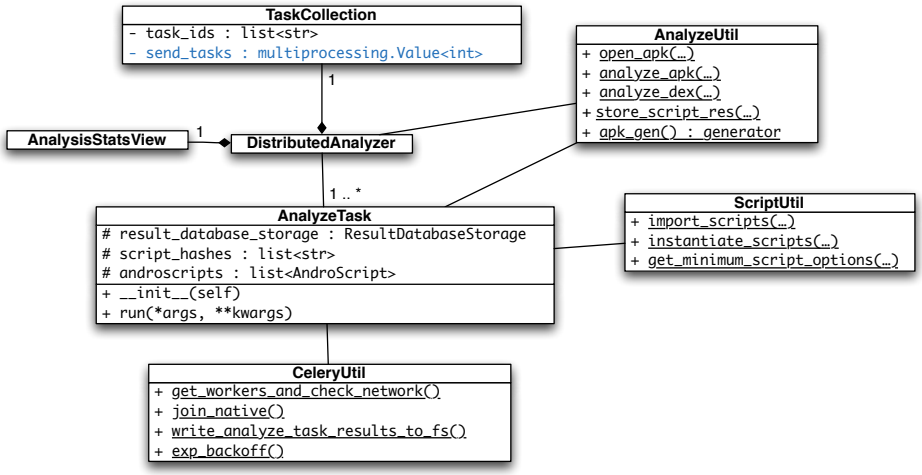

Abbildung 5.1: Klassendiagramm verteilte Analyse

Beim Veröffentlichen der Analysejobs behält die Klasse *TaskCollection* den Überblick und signalisiert über *shared memory* den Fortschritt.
Hilfsfunktionen zum Öffnen und Analysieren der APKs sowie Speichern der Ergebnisse in *MongoDB* stellt das Modul *AnalyzeUtil* bereit. Importieren der Skripte und Feststellen der minimalen Skript-Anforderungen implementiert das Modul *ScriptUtil*. Zum Installieren des *Callback-Handlers* in *RabbitMQ* sowie Empfangen und lokales Speichern der Analyseergebnisse ist *CeleryUtil* zuständig.
Um das System besser zu verstehen wird im Folgenden zunächst die Arbeit von der Seite der *Celery*-Worker und anschließend von der Seite des Clients betrachtet.

5.3.2.1 AnalyzeTask

Das verteilte Rechnen wird von *Celery* durch die *Task*-Klasse implementiert. Diese stellt die Ausführungseinheit der Worker dar.
Für jeden *Prozess* wird dabei ein Task initialisiert, der während der gesamten Laufzeit der Worker erhalten[53] bleibt. Somit können einmalige Initialisierungsaktionen vorgenommen werden. ALL nutzt diese Möglichkeit zum Anlegen von Instanzvariablen für die Datenbankverbindung sowie der Skripte.
Standardmäßig werden eine gewisse Anzahl von Aufgaben durch *Celery* für jeden Worker reserviert. ALL setzt diesen Wert so minimal wie möglich[54], sodass einerseits eine dynamische Lastverteilung erreicht wird und auf der anderen Seite trotzdem

[53] Zumindest wenn die Celery Option *CELERYD_MAX_TASKS_PER_CHILD* nicht auf einen Wert größer null gesetzt wurde [celh]
[54] Standardmäßig wird für jeden *Prozess* eine Aufgabe reserviert/vorgeladen

Nachrichten vorgeladen werden können, damit die Analyse so wenig wie möglich durch Warten auf Netzwerkdaten unterbrochen wird. Beim Senden der Rohdaten der APKs sind diese bereits vorgeladen, da sie Bestandteil der Nachricht sind, die die Worker vorladen.
Leider können zum jetzigen Zeitpunkt die APKs nicht vorgeladen werden, wenn nur die ID der Applikationen übertragen wird.
Celery besitzt verschiedene Signale mit denen auf Events reagiert werden kann. Der Code, der das Vorladen eines APKs aus der *MongoDB* implementieren könnte, wenn ein entsprechendes Signal zur Verfügung stände, ist in Abbildung 5.9 zu sehen[55]. Momentan ist die Methode mit dem Signal *task_prerun* verbunden und wird von *Celery* vor der *run*-Methode ausgeführt. Deshalb muss, falls die Datenbankverbindung noch nicht etabliert wurde, diese hergestellt werden (Zeile 3). Anschließend werden die Argumente des *AnalyzeTask* interpretiert. Wurde der APK-Hash mitgesendet, wird die Android-Applikation geladen und in einem *Dictionary* unter der ID abgelegt (Zeile 6-8).

Code Listing 5.9: APK Prefetch MongoDB

```
1  def prefetch_apk(self, task_id, task, *args, **kwargs):
2    try:
3      self.__setup_db()
4      args = kwargs["args"]
5      _, _, _, apk_zipfile_or_hash, is_hash, _ = args
6      if is_hash:
7        eandro_apk = self.__get_apk_from_mongodb(apk_zipfile_or_hash)
8        apk_prefetch_pool[apk_zipfile_or_hash] = eandro_apk
9    except (NoFile, DatabaseOpenError, DatabaseLoadException) as e:
10     log.exception(e)
```

Initialisiert werden die Instanzvariablen für die Datenbankverbindung und die Skripte, sobald der erste Analysejob ausgeführt werden muss.
Somit kann die Verbindung zur *MongoDB* wiederbenutzt werden, indem die Verbindungsinstanz als Klassenvariable hinterlegt werden kann und nicht jedes mal erneut aufgebaut werden muss.
Die zur Herstellung der Datenbankverbindung benötigten Informationen werden nicht über das Netzwerk übertragen. Dies wäre einerseits unsicher, da die *Credentials* übertragen werden müssen und auf der anderen Seite auch unnötig.
Jedem Worker liegt eine verteilte Konfiguration zugrunde, aus der die notwendigen Informationen ausgelesen werden können. Gebündelt sind diese durch die Klasse *Settings*, die die Funktionalität der Python-Klasse *ConfigParser* erweitert, abrufbar. Das Modul *celerysettings* stellt ein *Singleton* dieser Klasse bereit, die weiß, an welchem Ort die verteilte Konfiguration abgespeichert ist.
Der Task, der für die Analyse benutzt wird, ist in der Klasse *AnalyzeTask* implementiert. Genau wie bei *Threads* oder *Prozessen* wird dabei eine *run()*-Methode bereitgestellt, die für die eigentliche Ausführung des Jobs aufgerufen wird. Alle Informationen, die einmalig für den Job zur Ausführung notwendig sind werden hier übertragen und wurden im Designkapitel als Nachrichtenformat vorgestellt (Abbildung 4.12).
Abbildung 5.10 zeigt die *run()* Methode des Tasks. Zunächst werden die Argumente der Methode in einer Instanzvariable gespeichert. Diese sind nötig, um die Fehlertole-

[55] Es wurde bereits Kontakt mit dem Entwickler aufgenommen, damit das notwendige Signal bereitgestellt wird

ranz zu implementieren, da *Celery* diese für das erneute Ausführen des Jobs benötigt. Anschließend werden die Skripte initialisiert.

Code Listing 5.10: AnalyzeTask

```
1  def run(self, androscripts, min_script_needs, script_hashes,
       apk_zipfile_or_hash, is_hash = True, fast_apk = None):
2    try:
3      self.__retry_arguments = androscripts, min_script_needs,
           script_hashes, apk_zipfile_or_hash, is_hash, fast_apk
4      eandro_apk = None
5      do_script_hash_validation = settings.script_hash_validation_enabled
           ()
6      self.__open_db()
7
8      if do_script_hash_validation:
9          self.__setup_scripts_hash_validation(androscripts,
               script_hashes)
10     else:
11         self.__setup_scripts_reuse(androscripts, script_hashes)
12
13     if not is_hash:
14       eandro_apk = AnalyzeUtil.open_apk(apk_or_path =
             apk_zipfile_or_hash, apk = fast_apk, raw = True)
15     else:
16       eandro_apk = apk_prefetch_pool.get(apk_zipfile_or_hash, None)
17       if eandro_apk is None:
18         eandro_apk = self.__get_apk_from_mongodb_retry(
               apk_zipfile_or_hash)
19
20     if eandro_apk is not None:
21       result = AnalyzeUtil.analyze_apk(eandro_apk, self.androscripts,
             min_script_needs, propagate_error = False, reset_scripts =
             not do_script_hash_validation)
22       if result is not None:
23         fastapk, script_results = result
24         storage_results = self.__store_results(fastapk, script_results)
25         if storage_results:
26           return tuple(storage_results)
27
28     return ()
29   except (SoftTimeLimitExceeded, ScriptHashValidationError):
30     raise
31   finally:
32     if is_hash and apk_zipfile_or_hash in apk_prefetch_pool:
33       del apk_prefetch_pool[apk_zipfile_or_hash]
```

Neben den Paketnamen der Skripte werden außerdem deren Hashes übertragen, sodass die übermittelten Hashes mit denen aus dem Dateisystem geladenen Skripten abgeglichen werden können, um das Ausführen von eingeschleustem Code zu verhindern. Können die Hashes der Skripte nicht verifiziert werden wird mit der Exception *ScriptHashValidationError* der Fehler signalisiert und das Ausführen des Jobs verweigert (Zeile 9 und 29).

Die Skript-Validierung ist jedoch optional. Alternativ können die Hashes benutzt werden, um die Skripte wiederzuverwenden, falls sie mit den Skripten im Speicher übereinstimmen (Zeile 11).

Wurde nur der *identifier* des APKs übertragen, wird dies mit der Variable *is_hash*, die ebenfalls Teil des Nachrichtenformats ist, signalisiert. In dem Fall wurde die Android-Applikation bereits aus der *MongoDB* bezogen (*prefetch_apk*). Wurde die Datei nicht

gefunden, wird der Job abgebrochen und dem Benutzer, der die Analyse initiiert hat, der Traceback der Exeption angezeigt. Dieser wird in der Ergebniswarteschlange abgespeichert und durch *Celery* automatisch bewältigt.
Konnte die Applikation erfolgreich geladen werden, wird genau wie im parallelen Modus die Analyse ausgeführt, indem die App einmalig geöffnet wird und die Skripte nacheinander abgearbeitet werden (Zeile 21).
Das Vergleichen der Hashes führt allerdings auch dazu, dass jedesmal die Skripte erneut geladen werden müssen, damit die aktuellen Hash-Werte zur Verfügung stehen. Verzichtet man auf die Validierung der Skript-Hashes könnten die Skripte wiederverwendet werden.
Der Ergebnistupel, der die Resultate in der *MongoDB* identifiziert, wird im Erfolgsfall zurückgeliefert (Zeile 26). Ansonsten signalisiert ein leerer Tupel das Scheitern eines Analysejobs (Zeile 28).

Code Listing 5.11: Fehlertoleranz Speicherung Ergebnisse - Verbindungsfehler

```
1  @RetryDecorator(exception_tuple = (StorageException, ),
2    caused_by_tuple = CONNECTION_FAIL_ERRORS,
3    max_retries = CELERY_ANALYSIS_STORE_RES_RETRY_CNT,
4    max_retry_time = CELERY_DATABASE_STORE_RETRY_MAX_TIME
5    )
6  def __store_results(self, fastapk, script_results):
7    rds = self.result_database_storage
8    res = []
9    for script in script_results:
10     pres = rds.store_result_for_apk(fastapk, script)
11     if pres is not None:
12       res.append(pres)
13
14   return res
```

Die Fehlertoleranz wird durch einen dafür extra geschriebenen *Decorator* implementiert. Abbildung 5.11 zeigt am Beispiel der Ergebnisspeicherung, wie im Falle eines Netzwerkproblems ein erneutes Ausführen des Jobs initiiert wird. Dafür wird dem *Decorator* ein Tupel an *Exceptions* übergeben, die Fehler signalisieren, für die es sich lohnt einen Neuversuch zu starten. Denn nicht alle Fehler sind durch erneutes Ausführen des Jobs behandelbar.
Fast alle Exceptions, die ALL zur Fehlerbehandlung benutzt, sind eine Unterklasse von *WrapperException*. Dadurch besitzen diese ein Feld *caused_by*, das gesetzt wird, wenn ein Fehler abgefangen und in eine neue Exception verpackt wird. Somit sind die ursprünglichen *Exceptions* immer noch verfügbar. Der *RetryDecorator* kann diese Eigenschaft ausnutzen, indem auch auf dem ursprünglichen Fehler ein Matching durchgeführt werden kann.
Der Job wird bei der Ergebnisspeicherung also neu initiiert, falls der Fehler *StorageException* auftritt und der ursprüngliche Fehler in dem Tupel, der durch *CONNECTION_FAIL_ERROR* beschrieben wird, enthalten ist. Dahinter versteckt sich die Exception *ConnectionFailure*, die ein Verbindungsproblem seitens *MongoDB* aufzeigt.
Des Weiteren wird die maximale Anzahl an Versuchen, die unternommen werden sollen definiert. Zusätzlich wird noch eine zeitliche Verzögerung, die mithilfe von *exponential backoff* implementiert ist, hinzugefügt. Somit wird bis zu einer im Code festgelegten Obergrenze die Zeit exponentiell erhöht, um das System für einen längeren Ausfall der Datenbank nicht zu sehr zu belasten.
Das Initiieren eines Neuversuchs bedeutet, dass der Job mit denselben Argumenten

und der gleichen *Task-ID* erneut in die Warteschlange eingefügt wird. Jeder Task besitzt außerdem einen Status, der dessen Zustand beschreibt. Somit kann bei einem Neuversuch das vorherige Scheitern im Status festgehalten werden.

5.3.2.2 DistributedAnalyzer

Der *DistributedAnalyzer* ist der eigentliche Initiator der verteilten Analyse. Ihm liegt entweder eine Liste von Instanzen der Klasse *FastApk* oder eine Liste von Pfaden (zu den APKs) vor. Ersteres ist der Fall, falls eine Analyse basierend auf den Informationen der Importdatenbank vorgenommen wird. Letzteres ist nützlich, möchte man schnell eine verteilte Analyse ohne vorherigen Import in die Importdatenbank sowie ggf. *MongoDB* vornehmen.
In diesem Fall müssen die Rohdaten des APKs zwingend in die Nachricht integriert werden. Im ersten Fall ist dies optional, falls die Daten ebenfalls in der *MongoDB* vorliegen, denn dann ist das Senden des APK-*identifier* ausreichend.
Bei der Initialisierung der Klasse werden zunächst Signale registriert, damit das Veröffentlichen der Jobs im *Nachrichten-Broker* nachvollzogen werden kann, um den Benutzer über den Fortschritt der Nachrichtenübermittlung informieren zu können.
Wenn die Analyse durch Ausführen der *_analyze* Methode initiiert wurde, wird zunächst die Konnektivität zum *Nachrichten-Broker* überprüft. Ist dieser nicht erreichbar wird die Analyse abgebrochen. Das Verhalten ist im Quellcode in Listing 5.12 abgebildet.

Code Listing 5.12: Liste Worker und Teste Netzwerkonnektivität

```
1 try:
2     clilog.info(CeleryUtil.get_workers_and_check_network())
3 except NetworkError as e:
4     log.critical(e)
5     return 0
```

Außerdem wird ein *Ping* von registrierten Celery-Workern angefordert [celh], damit diese aufgelistet werden können. Somit wird ebenfalls über die Situation informiert, dass das Cluster von Arbeitern gar nicht gestartet wurde.
Anschließend wird zusätzlich noch die Verbindung zur Ergebnisdatenbank überprüft. Auch hier wird die Analyse im Fehlerfall abgebrochen. Die Analyse kann jedoch jederzeit erneut angestoßen werden.
Ist sowohl Konnektivität zu *RabbitMQ* als auch *MongoDB* gewährleistet, wird das Einreihen der Analyse-Jobs in die Warteschlange gestartet (dargestellt in Abbildung 5.13).

Code Listing 5.13: Verteilte Analyse

```
1  def _analyze(self):
2    storage = self.storage
3    try:
4      storage.create_or_open_sub_storages()
5      apk_gen = AnalyzeUtil.apk_id_or_raw_data_gen(self.apks,
           force_raw_data = self.serialize_apks)
6
7      if self.serialize_apks:
8        self.group_result = GroupResult(results = [])
9
10       for args in self.send_apk_args_generator(apk_gen):
11         task = analyze_task.delay(*args)
12         self.group_result.add(task)
13     else:
```

```
14        task_group = group((analyze_task.s(*args) for args in self.
              send_id_args_generator(apk_gen)))
15        self.group_result = task_group()
16
17      self.analyze_stats_view.start()
18
19      callback_func = self.get_callback_func(self.success_handler, self.
            error_handler)
20      CeleryUtil.join_native(self.group_result, propagate = False,
            callback = callback_func)
21
22      return self.stop_analysis_view()
23
24    except DatabaseOpenError as e:
25      log.critical(e)
26      return 0
27    except (KeyboardInterrupt, Exception) as e:
28
29      if celerysettings.CELERY_TASK_REVOCATION_ENABLED:
30        if self.group_result is None:
31          self.task_collection.revoke_all(terminate = True, signal = '
                SIGKILL')
32        else:
33          self.group_result.revoke(terminate = True, signal = 'SIGKILL')
34
35      return self.stop_analysis_view()
```

Celery bietet bereits eine Primitive [cela] zum Ausführen von Gruppenjobs an, die für das Senden der Nachrichten mit APK-*identifier* auch genutzt wird (Zeile 14-15). Beim Serialisieren der rohen *APK*-Daten hingegen wurde festgestellt, dass dies zu einem Speicherproblem führt, weshalb auf eine eigene Lösung zurückgegriffen wird, die die Jobs der Reihe nach serialisiert (Zeile 8-12).
Für eine Fortschrittsanzeige auf dem CLI wird ein *Thread* verwendet, damit neben dem eigentlichen Fortschritt die Dauer der Analyse kontinuierlich aktualisiert und dargestellt werden kann.
Nachdem die Nachrichten serialisiert und an *RabbitMQ* übermittelt wurden, wird ein *Callback-Handler* in der Ergebniswarteschlange registriert, der aufgerufen wird, sobald ein Worker einen Job beendet hat. Der Vorgang ist im Quelltext unter dem Methodenaufruf *CeleryUtil.join_native* ersichtlich, dem als Argument der *Callback-Handler* übergeben wird (Zeile 19-20).
Dabei existiert sowohl ein Handler für den Erfolgsfall als auch einer für das erfolglose Ausführen des Jobs. Ersterer empfängt aus der Ergebniswarteschlange einen Ergebnistupel, der das Analyseergebnis in der Datenbank identifiziert, woraufhin die Ergebnisse abgerufen und lokal im Dateisystem niedergeschrieben werden[56]. Somit werden die Ergebnisse schon während der Analyse gespeichert.
Scheitert der Analysejob in der Ausführung, wird die Exception sowie der Traceback aus der Warteschlange empfangen und dem Benutzer präsentiert.
Die beiden Callback-Handler sind in den Abbildungen 5.14 und 5.15 dargestellt.

Code Listing 5.14: Success callback handler

```
1 def success_handler(self, task_id, result):
2   if result is not None:
3     for res in result:
4       self.add_storage_result(res)
```

[56] Zumindest wenn die lokale Speicherung aktiviert ist. Siehe Kapitel 4.10

```
5     self.storage.fetch_results_from_mongodb(result, wait_for_db = True)
```

Es stellt sich allerdings noch die Frage wie vorgegangen werden soll, wenn die Analyse vorzeitig abgebrochen wird. Auch dafür bietet ALL Flexibilität über die verteilte Konfiguration[57] an.

Code Listing 5.15: Error callback handler

```
1 def error_handler(self, task_id, error_msg, state, traceback = None):
2     log.error(str(error_msg))
3     if traceback:
4         log.error(traceback)
```

Entweder kann die Analyse fortgesetzt oder abgebrochen werden. Für letzteren Zweck bietet *Celery* das Zurückziehen von Tasks [celh]. Die *Prozesse* werden mit dem Signal *SIGKILL* explizit zum Beenden gezwungen, denn ein Beenden mit *SIGINT* oder *SIGTERM* führt dazu, dass die *Worker* noch die aktuellen Jobs beenden und wurde nicht verwendet, da je nach Skript die Analyse sehr lang dauern kann.

5.4 Storage

In diesem Abschnitt wird auf die Implementierung des Speichersystems näher eingegangen. Dabei wird nur die Speicherung von Analyseergebnissen und das Kopieren der APKs in *MongoDB* vorgestellt, da das Abspeichern der APK-Metainformationen via *SQL* in der Import-Datenbank nicht sonderlich interessant ist, jedoch aber das Auslesen der Metainformationen aus dem Manifest. Denn diese wurde so optimiert, dass nur die notwendigen Daten aus der Zip-Datei entpackt werden.

5.4.1 Import

Das Importieren der APKs in die Importdatenbank erfordert die Kenntnis über Informationen wie Paketname sowie die Version der Applikation. Zusätzlich wird die Größe der *classes.dex* als Metrik erfasst, damit ein besseres *scheduling* implementiert werden kann. Während *androguard* die Funktionalität zwar bereitstellt, geschieht dies nicht so effizient wie möglich, da die gesamte Zip-Datei entpackt wird. Die von ALL optimierte Importfunktion ist in Abbildung 5.16 visualisiert.

Code Listing 5.16: FastApk - Auslesen der Metainformationen aus AndroidManifest.xml

```
1  def fast_load_from_io(file_like_object = None, apk_file_path = None,
       calculate_hash = True):
2    flo, file_open_from_path = ..., ...
3    flo.seek(0)
4    _hash = None
5
6    if calculate_hash:
7      _hash = Util.sha256(flo.read())
8      flo.seek(0)
9    try:
10     if zipfile.is_zipfile(flo):
11       z = zipfile.ZipFile(flo)
12       binary_manifest = z.read(MANIFEST_FILENAME)
13       ap = androapk.AXMLPrinter(binary_manifest)
14       dom = minidom.parseString(ap.get_buff())
15       manifest_tag = dom.getElementsByTagName(MANIFEST_TAG_NAME)
16       if len(manifest_tag) > 0:
```

[57] Konfigurierbar über die Einstellung *task_revocation* in der Sektion *Analyse*

```
17          size_app_code = z.getinfo(COMPILED_APP_CODE).file_size
18          manifest_items = manifest_tag[0].attributes
19          version_name = manifest_items.getNamedItemNS(MANIFEST_NS,
                MANIFEST_VERSION_NAME).nodeValue
20          package = manifest_items.getNamedItem(MANIFEST_PACKAGE).
                nodeValue
21          return FastApk(package, version_name, path = apk_file_path,
                _hash = _hash, size_app_code = size_app_code)
22      raise CouldNotOpenManifest(apk_file_path), None, sys.exc_info()[2]
23    except Exception as e:
24      raise CouldNotOpenApk(apk_file_path, caused_by = e), None, sys.
            exc_info()[2]
25    finally:
26      if file_open_from_path:
27        flo.close()
```

Die Methode kann dabei entweder auf einem *file-like-object* oder durch Angabe des Dateipfades zum APK aufgerufen werden. Der Sachverhalt wird abstrahiert, indem je nach Parameter ein *file-like-object* in der Variable *flo* abgelegt wird. Der Quelltext hierfür wurde aus Übersichtsgründen gekürzt.
Die Abbildung zeigt, dass die *.apk*-Datei einmalig in den Speicher geladen wird, um sowohl das Hashing als auch das Auslesen der Informationen ohne erneute I/O-Aktivität durchführen zu können. Das *file-like-object* bietet das Interface eines *file* in Python, das jedoch im Speicher abgebildet ist.
Nach Überprüfung, ob es sich um eine valide *Zip*-Datei handelt, wird nur die Datei *AndroidManifest.xml* extrahiert. Da diese binär kodiert ist muss mithilfe von *androguard* eine Konvertierung vorgenommen werden (Zeile 12-13).
Anschließend werden der Paketname und die Version aus dem Manifest gelesen. Beide sind im Tag *manifest* als Attribut enthalten, wie in Abbildung 2.2 bereits dargestellt wurde.
In Zeile 17 ist zu sehen wie die unkomprimierte Größe der *classes.dex* zur Verwendung als Metrik für die Größe des Quelltextes ohne Entpacken ausgelesen wird. Nachdem alle Informationen gesammelt wurden, werden diese in der Klasse *FastApk* gebündelt zurückgeliefert (Zeile 21). Je nach Fehler wird entweder die Exception *CouldNotOpenManifest* oder *CouldNotOpenApk* ausgelöst.

5.4.2 Ergebnisdatenbank

Das Speichern der Analyseergebnisse ist in Abbildung 5.17 dargestellt.

Code Listing 5.17: Speicherung Analyseergebnis - Implementierung ResultStorageInterface

```
1  def store_result_for_apk(self, apk, script):
2    try:
3      res_obj_dict = MongoUtil.escape_keys(script.result_dict())
4      _id = script.gen_unique_id()
5      if script.uses_custom_result_object() or script.is_big_res():
6        result = self.get_custom_res_obj_representation(script)
7        gridfs = self.grid_fs
8
9        if gridfs.exists(**{RESOBJ_ID : _id}):
10         gridfs.delete(_id)
11       gridfs.put(result, metadata = res_obj_dict, filename = script.
             get_file_name(), _id = _id)
12       return _id, True
13     else:
14       res_obj_dict[RESOBJ_ID] = _id
```

```
15        self.res_coll.update({RESOBJ_ID : _id}, res_obj_dict, upsert =
              True)
16        return _id, False
17
18    except (PyMongoError, BSONError) as e:
19      raise DatabaseStoreException(self, "script: %s" % script, caused_by
              = e), None, sys.exc_info()[2]
```

Diese werden über die Methode *result_dict()* des *AndroScript* abgefragt und über *gen_unique_id()* wird ein *identifier* für das Ergebnis erstellt (Zeile 3 und 4). Der *identifier* wird berechnet, indem der SHA-256 Hash des APK zusammen mit dem Skriptnamen erneut mit SHA-256 gehasht wird. Ferner werden alte Ergebnisse desselben APK überschrieben, wenn der Skriptname gleich ist.

Bei der Speicherung muss zwischen zwei Situationen unterschieden werden. Je nachdem wie groß das Ergebnis sein kann, kann via *is_big_res()* im *AndroScript* signalisiert werden, dass die Ergebnisse im *GridFS* abgelegt werden sollen. Die Benutzung eines eigenen Objekts zur Speicherung der Ergebnisse impliziert die automatische Benutzung von *GridFS*. Abgefragt wird dies über die Methode *uses_custom_result_object()* (Zeile 5).

In beiden Situation wird das *ResultObject* mindestens für das Loggen der Apk- sowie Skript-Metainformationen benutzt. Diese werden in der Variable *res_obj_dict* abgelegt. Zum Speichern kann dem *MongoDB*-Treiber direkt ein *Dictionary* übergeben werden.

Dadurch, dass der Benutzer die Daten sowie das Format vorgibt, mit dem die Daten in der Datenbank abgelegt werden, muss vorher eine Überprüfung und Ersetzung von reservierten Zeichen erfolgen. Nicht erlaubt sind "." sowie "$" [mona] und werden von ALL durch "_" sowie "_$" ersetzt. Betroffen sind allerdings nur Schlüssel in dem *Dictionary*. Folglich betrifft dies nur die Kategorien, die beim Loggen benutzt werden. Nicht aber die Werte, die geloggt werden, außer man loggt ein *Dictionary*.

Das Ersetzen der reservierten Zeichen wird durch die Methode *escape_keys* durchgeführt (Zeile 3), indem jede iterierbare Struktur außer Strings daraufhin überprüft wird, ob ein *Dictionary* enthalten ist. Anschließend werden in allen *Dictionaries* die Zeichen ersetzt.

Da für *GridFS* keine Updatefunktion existiert, muss ein altes Ergebnis, wenn vorhanden, zunächst gelöscht werden. Anschließend kann das Ergebnis zusammen mit den Metainformationen in der Datenbank abgelegt werden (Zeile 9-11). Auf den Metainformationen können immer noch Anfragen formuliert werden, denn diese werden nicht in *Chunks* zerlegt.

Das Format in dem die Daten gespeichert werden, kann durch das Interface *CustomResultObjInterface* definiert werden. Ist dieses nicht implementiert wird die *__str__()*-Methode des Ergebnisobjekts benutzt.

In dem dargestellten Listing wird bei einem Fehler seitens *MongoDB*s und falls ein Fehler beim Konvertieren in das *BSON*-Format auftritt (z.B. Überschreitung der Dateigröße von max. 16MB [monc]), eine *DatabaseStoreException* zum Signalisieren des Fehlers benutzt (Zeile 19).

Das Importieren der APKs in die *MongoDB* verläuft analog, weshalb nicht extra darauf eingegangen werden muss.

6 Evaluation und Messungen

Im Folgenden werden einige Messungen und Experimente vorgestellt und anschließend eine Evaluierung der Ergebnisse vorgenommen.
Für die Experimente standen insgesamt vier Computer zur Verfügung. Davon sind drei mit *KVM* virtualisierte Maschinen.
Die Hardware ist allerdings homogen. Die technischen Daten des physikalischen Testsystems können Abbildung 6.1 entnommen werden. Die virtuelle Hardware der VMs unterscheidet sich einerseits im Arbeitsspeicher von nur 16GB sowie einer Festplatte ohne *RAID* mit einer Größe von 128GB. Auf allen Systemen ist Ubuntu als OS installiert. Jedoch unterscheiden sich diese in ihren Versionen.
Für die Messungen wurden die 500 beliebtesten Apps[59] aus allen Kategorien aus dem *Play Store* heruntergeladen. Die verschiedenen APK-Sets sind in Abbildung 6.2 zu sehen.

OS	Ubuntu 13.10 Saucy
Kernel	3.11.0-23-generic
RAM	4x 8GiB DIMM DDR3 Synchronous 1600 MHz (0,6 ns)
SSD	Toshiba MKNSSDCR240GB
HDD	RAID1: 2x 3TB Seagate ST3000DM001-1CH1
CPU	Intel(R) Core(TM) i7-4771 CPU @ 3.50GHz[58]
Cores	4
Cache sizes L1/L2/L3	256KiB, 1MiB, 8MiB
Network	Ethernet Connection I217-LM 1Gbit/s

Abbildung 6.1: Technische Daten physikalisches Testsystem

Setname	***Beschreibung***	***Anzahl APKs***	***Größe in MB***
Apkset1	Top Free 4	102	1.159
Apkset2	Top Free 100	2.519	22.315
Apkset3	Top Free 500	12.689	91.764

Abbildung 6.2: APK-Testsets

Außerdem wurden verschiedene Sets an Skripten zusammengestellt, die in Abbildung 6.3 visualisiert sind. Die Sets haben außerdem unterschiedliche Skript-Anforderungen.

[59] Die Applikationen wurden am 07.09.2014 heruntergeladen

Setname	*Anforderungen*	*Skripte*
Manifest	Keine	ChainedApkInfos, Files, Libs, Activities, Intents, ContentProviders, Services, BroadcastReceivers, Permissions
Manifest + SSL	XREF	Alle aus Skriptset Manifest + SSL
Misc1	XREF	Manifest + SSL + ClassListing, ClassDetails, AnalyzeFrameworks, GVMAnalysisExample
Misc2	XREF	Misc1 + Decompile

Abbildung 6.3: Skript-Testsets

6.1 Durchführung

Damit die Experimente aussagekräftige Werte zeigen, wurde jeder Wert, der in einem Diagramm eingetragen ist, durch mindestens zehnmaliges Wiederholen erreicht. Bei jedem Testlauf wurde die jeweilige Zeit gespeichert, sodass die Standardabweichungen berechnet werden können. Im Gegensatz zu den Mittelwerten sind sie in den Diagrammen aus Übersichtsgründen jedoch nicht eingetragen. Die y-Achse beschreibt bei jedem Experiment die Zeit, die für die Messung benötigt wurde und ist in Sekunden gemessen. Gerundet wurde jeweils auf zwei Nachkommastellen.
Die Diagramme wurden mit dem Online-Service plot.ly erstellt, der auch für die Berechnung der Standardabweichung benutzt wurde.

6.2 Import

Das Importexperiment testet einerseits wie weit sich das Importieren der APKs in die Importdatenbank parallelisieren lässt indem mehrere *Prozesse* verwendet werden und auf der anderen Seite, ob der selbst entwickelte Importmechanismus gegenüber *androguard* Geschwindigkeitsvorteile erzielen kann. Dafür wurden die beiden Importmechanismen gegeneinander getestet und des Weiteren untersucht, inwiefern das gleichzeitige Strukturieren und Kopieren der Applikationen durch ALL die Importzeit beeinflusst.
Dafür wurde *Apkset2* verwendet. Die Import-Datenbank sowie das Importverzeichnis, in das die APKs einsortiert werden, wurden nach jedem Durchlauf gelöscht. Die Applikationen wurden von der HDD geladen und die APKs auch auf diese kopiert. Die Datenbank wurde hingegen auf der SSD angelegt.
Die Ergebnisse in Abbildung 6.4 zeigen, dass der selbst entwickelte Importmechanismus effizienter im Vergleich zum Öffnen der APKs mit *androguard* ist. Bei der Verwendung von nur einem Prozess werden nur 245,37 Sekunden benötigt. *androguard* benötigt im Mittel 496,98 Sekunden. Somit konnte eine Optimierung von 251,61 Sekunden erzielt werden. Also eine Steigerung um 50,62%.
Mit steigender Anzahl von *Prozessen* nimmt der Geschwindigkeitsvorteil jedoch ab. Der schnellste Import mit *androguard* wurde mit 19 *Prozessen* erreicht. Mit einer Zeitdifferenz von 15,93 Sekunden ist der eigene Import jedoch immer noch um 16,14% schneller.
Der Performanzunterschied lässt sich dadurch erklären, dass *androguard* die gesamte

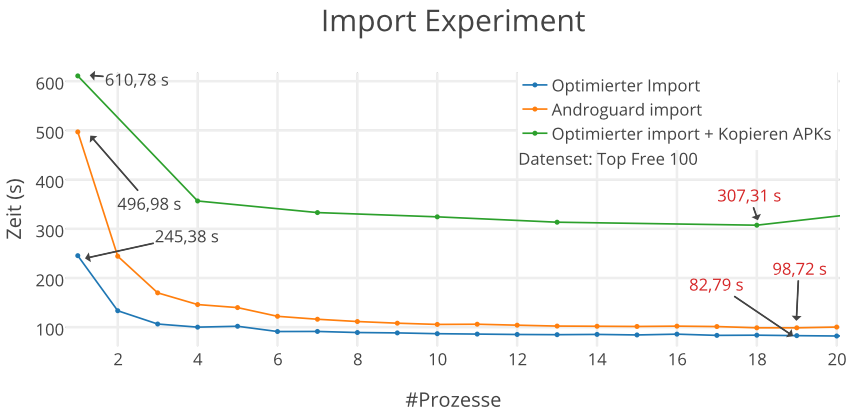

Abbildung 6.4: Import Experiment

Zipdatei, während ALL lediglich das Manifest entpackt. Bei beiden wurde das gezippte APK vorher in den Speicher geladen, indem mithilfe der Klasse *BytesIO* ein *file-like-object* im Speicher erzeugt wurde, das das Interface bereitstellt, das durch die *file* Klasse bekannt ist.

Der Unterschied der beiden Importverfahren wird durch die Methoden *FastApk.fast_load_from_io* sowie *FastApk.androguard_load_from_io* implementiert und kann in der Klasse *FastApk* nachvollzogen werden.

Das Experiment wurde mit dem eigenen Importverfahren noch bis zu 200 *Prozessen* ausgeführt. Aus Gründen der Übersichtlichkeit aber nicht mehr in der Abbildung dargestellt [60]. Das beste Ergebnis konnte mit der Verwendung von 172 *Prozessen* erreicht werden. Mit 72,01 Sekunden wurde gegenüber der Verwendung von nur einem Prozess eine Steigerung um 3,41 erzielt.

Außerdem ist zu sehen, dass das zusätzliche Kopieren der APKs den Import deutlich verlangsamt. Die Parallelisierung ermöglicht hier mit 18 *Prozessen* immer noch eine Effizienzsteigerung um 49,69% mit einer Importdauer von 307,31 Sekunden. Auch der Import mit Kopieren wurde bis zu 200 *Prozessen* durchgeführt. Jedoch zeigt sich im Unterschied zu dem durch ALL optimierten Import ohne Kopieren ab einer Grenze von 21 *Prozessen* wieder eine Verschlechterung der Importdauer. Das lässt sich wahrscheinlich durch die zu hohe Anzahl von gleichzeitigen Zugriffen auf die Festplatte erklären.

6.3 Skript-Anforderungen

Im Design-Kapitel (4.5.3) wurden für eine effizientere Analyse die Anforderungen, die ein Skript benötigen kann aufgeteilt, sodass diese nur nach Bedarf bereitgestellt werden. Das nachfolgende Experiment zeigt in einer isolierten Testumgebung wie diese Anforderungen die Analysezeit beeinflussen. Dafür wurde für jede Skriptanforderung ein *AndroScript* erstellt, das die jeweilige Methode überschreibt, um die Anforderung zu signalisieren. Zusätzlich wurde noch ein Skript erstellt, das alle Anforderungen benötigt. Die *_analyze* Methode wurde dabei leer gelassen, sodass keine Analyse durchgeführt wird.

ALL öffnet die APKs und stellt die je nach Anforderungen benötigten Analyse-Objekte durch *androguard* bereit. Auch wenn der Rumpf der Analysemethode nicht gefüllt wurde, wurden trotzdem die Ergebnisse sowohl im Dateisystem als auch in der Ergeb-

[60] Online verfügbar unter `https://plot.ly/~nachtmaar/114`

nisdatenbank gespeichert. Das passiert automatisch durch ALL, indem das *ResultObject* zum Loggen der Metainformationen benutzt wird.

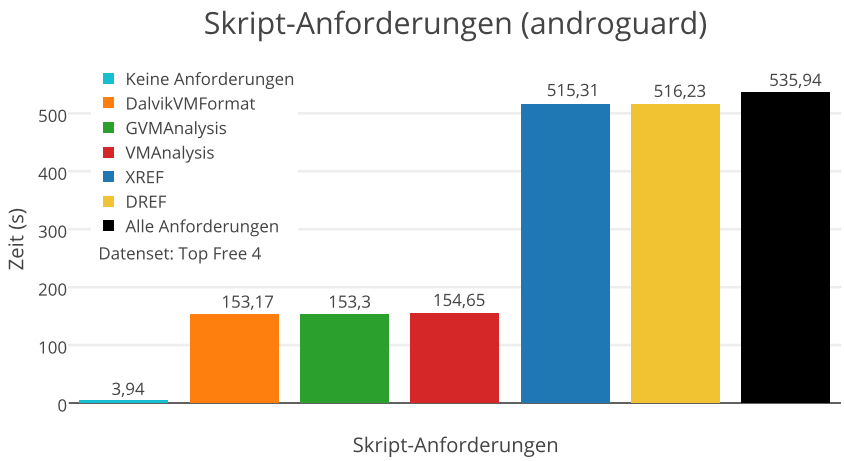

Abbildung 6.5: Skript Anforderungen

Die Ergebnisse[61] sind in Abbildung 6.5 zu sehen. Ein Skript ohne Anforderungen benötigt nur einen Bruchteil der Zeit, die die anderen Anforderungen beanspruchen. Ohne Skript-Anforderungen wird lediglich der Zugriff auf die Klasse *EAndroApk* zur Verfügung gestellt.

Schon mit der ersten Anforderung (*DalvikVMFormat*) wird die gesamte Analyse um 149,23 Sekunden und somit um den Faktor 38,88 langsamer. Die nächsten Anforderung, die die Objekte *VMAnalysis* sowie *GVMAnalysis* zur Verfügung stellen, ändern an der benötigten Zeit nichts. Mit einem Unterschied von maximal +/- 1,5 Sekunden kann es sich um Ungenauigkeiten in der Messung handeln, auch wenn die Standardabweichungen gering sind[62].

Abbildung 6.6 zeigt die Abhängigkeiten zwischen den Analyse-Objekten. Das *GVMAnalysis*-Objekt benötigt das Objekt *VMAnalysis*, welches wiederum das *DalvikVMFormat*-Objekt benötigt. Sowohl *XREF* als auch *DREF* benötigen alle drei Objekte.

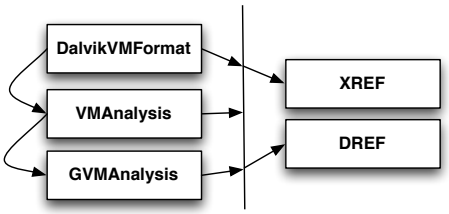

Abbildung 6.6: Abhängigkeiten zwischen Skript-Anforderungen

Das Erstellen des *GVMAnalysis*-Objektes müsste folglich langsamer als das des *VMAnalysis* sein, da ersteres dieses für die Initialisierung benötigt.

Am zeitintensivsten ist das Erstellen der Querverweise zwischen Methoden und Feldern (respektive *XREF* und *DREF*). Der Mehraufwand lässt sich dadurch erklären, dass für die Verweise alle anderen bisher gezeigten Analyse-Objekte benötigt werden. Durch die Standardabweichungen und den Unterschied von nur einer Sekunde zwischen dem Erstellen von *XREF* bzw. *DREF* lässt sich auch hier keine Aussage darüber treffen, welches die längere Zeit benötigt und kann je nach Quellcode und APKs evtl. variieren.

Das Experiment zeigt also auf, dass das Bereitstellen der Analyse-Objekte nur auf Anforderung einen deutlichen Geschwindigkeitsvorteil erbringt.

[61] Das Diagramm ist online unter `https://plot.ly/~nachtmaar/122` interaktiv einsehbar

[62] Standardabweichungen sind respektive der im Diagramm gegebenen Reihenfolge: 0.038, 0.43, 0.35, 0.63, 0.58, 0.65, 0.56

Um z.B. nur die Metainformationen aus dem Manifest auszulesen wird keine Anforderung benötigt, denn die *EAndroAPK* Klasse wird dem Analysten standardmäßig bereitgestellt. Folglich können Informationen wie Komponenten, Intents oder benötigte App-Berechtigungen effizient ausgelesen werden.
Insgesamt lässt sich aus den Ergebnissen schließen, dass bei sorgfältiger Benutzung der Skript-Anforderungen eine deutliche Leistungssteigerung erzielt werden kann. Außerdem zeigen sie, dass eine gröbere Unterteilung der Anforderungen für die Objekte *DalvikVMFormat*, *VMAnalysis* sowie *GVMAnalysis* vorgenommen werden sollte. Genau eine, anstelle von aktuell drei Anforderungen sollte die Bereitstellung der drei Objekte ermöglichen, da die benötigte Zeit zur Bereitstellung nur sehr geringfügig variiert. Des Weiteren wird durch weniger Anforderungen, die definiert und unterschieden werden müssen, das Schreiben der Skripte vereinfacht.

6.4 Parallele Analyse

In diesem Abschnitt wurden Experimente rund um die parallele Analyse durchgeführt. Es wurde versucht die optimalen Einstellungen zu finden, sodass die parallele Analyse als Referenzimplementierung genutzt werden kann, um die Performanz der verteilten Analyse zu messen. Der Vergleich beider Modi wird in 6.5.3 durchgeführt.
Um die besten Einstellungen für den parallelen Modus zu finden, wurden diverse Optimierungen ausprobiert. Zum einen wurde das Interface *Resetable* eingeführt, sodass *AndroScript*e zurückgesetzt werden können und kein neues Objekt der Klasse für das nächste APK erzeugt werden muss. Außerdem wurde ein Puffer eingeführt, der die Ergebnisse im Speicher behält bis die maximale Speicherkapazität des Puffers erreicht ist. Erst dann werden die Ergebnisse lokal im Dateisystem festgehalten und in der Ergebnisdatenbank abgelegt. Die Idee dahinter ist, dass die Arbeiter die Analyse nicht nach jedem APK unterbrechen müssen, um I/O auszuführen.
Des Weiteren wurde für Jobs, die in der Laufzeit stark variieren, versucht ein besseres *Scheduling* zu erreichen, indem lange Jobs bereits am Anfang bearbeitet werden. Somit wird eine bessere Ausnutzung der Parallelität erreicht. Denn sonst kann es vorkommen, dass im *worst case* die Analyse, die am längsten benötigt, erst am Ende ausgeführt wird und die anderen Arbeiter bereits ihre Aufträge abgearbeitet haben. Eine weitere Überlegung war die Granularität der Aufgaben speziell für diesen Fall feiner zu gestalten, sodass ebenfalls möglichst lange alle Arbeiter Analysen durchführen können.

6.4.1 Ergebnispuffer

Mit dem Einführen eines Ergebnispuffers für die parallele Analyse wurde versucht, I/O-Aktivitäten zu bündeln, indem nicht nach jeder Analyse die Ergebnisse gespeichert werden. Das bedeutet allerdings auch, dass, je nach Größe des Ergebnisses viel Speicher benutzt werden kann, der folglich bei einer speicherintensiven Analyse nicht zur Verfügung steht.
Die nachfolgenden Experimente untersuchen mit zwei unterschiedlichen Skriptsets wie sich die Größe des Puffers auf die Analysedauer auswirkt.
Bei der Größe, die auf der x-Achse aufgetragen ist, handelt es sich jeweils um die Puffergröße pro *Prozess*/*Thread*. Ein Element in dem Puffer beinhaltet die Ergebnisse der Analyse eines APK mit allen Skripten.

Für die Analyse wurden acht *Threads*/*Prozesse* verwendet. Die Messwerte sind alle Zweierpotenzen zwischen eins und 128.

Das verwendete Skript-Set *Manifest* besteht aus Skripten, die Informationen aus dem Manifest auslesen und keine Skript-Anforderungen benötigen[63]. Das in Abbildung 6.7 unten rechts dargestellte Experiment[64] benutzt *Prozesse* und zeigt, dass bei einer Puffergröße von eins die Analyse am längsten dauert. Mit diesem Wert wird der Puffer bei der Bearbeitung des nächsten Job geleert, sodass der Wert dem Deaktivieren des Puffers gleichkommt. Die schnellste Zeit wurde mit einer Puffergröße von 16 erzielt. Ab diesem Wert sieht man eine Verschlechterung bezogen auf die Analysedauer.

Mit mehr als 120 Ergebnissen pro *Prozess*/*Thread* und jeweils neun Skripten, wobei

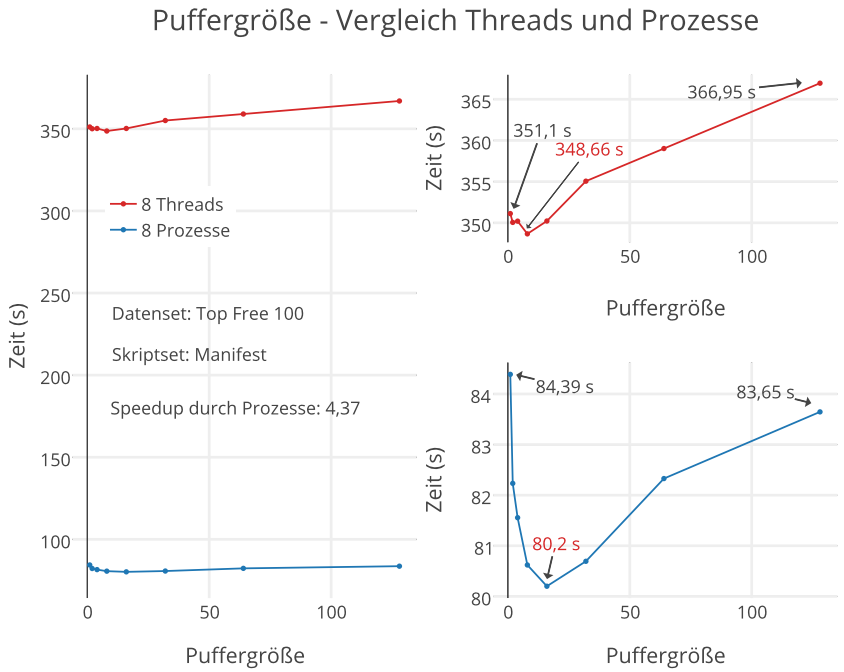

Abbildung 6.7: Puffergröße - Vergleich Threads und Prozesse

ChainedApkInfos eine Zusammenfassung aller Manifest-Skripte als *ChainedScript* ist, befinden sich bei einer Anzahl von 2.519 APKs alle Ergebnisse im Speicher des jeweiligen Arbeiter, sodass die gesamte I/O-Aktivität der Worker erst am Ende der Analyse erfolgt. Dabei muss allerdings berücksichtigt werden, dass nicht alle Arbeiter gleich viele Ergebnisse im Puffer am Ende der Analyse haben, da bedingt durch die Größe des jeweiligen APK die Analyse-Jobs unterschiedlich schnell abgearbeitet werden können und folglich manche Puffer bereits geleert sein können.

Wenn man bedenkt, dass die Analyse mit dem Skriptset *Manifest* so schnell erfolgt und sehr viel I/O-Aktivität in kurzer Zeit erfolgt, könnte die Analyse evtl. auch mit *Threads* durchführbar sein, da der *GIL* in *CPython* bei I/O-Zugriffen den Lock des *Interpreters* freigibt, wodurch paralleler I/O möglich wird.

Die Ergebnisse dieser Idee sind ebenfalls in Abbildung 6.7 zu finden. Vergleicht man die benötigten Analysezeiten zwischen *Threads* und *Prozessen*, so sieht man, dass der Einsatz von *Threads* die Leistung deutlich verschlechtert.

Die Analyse mit *Prozessen* ist 268,49 Sekunden, folglich 4,37-mal schneller. Dabei wurden die beiden kleinsten Messwerte voneinander subtrahiert.

[63] Standardabweichungen: 0.92, 0.79, 0.82, 0.9, 0.72, 0.75, 1.34, 1.16

[64] Siehe `https://plot.ly/~nachtmaar/164`

Das zweite Experiment[65], das in 6.8 abgebildet ist, benutzt das Skriptset *Manifest + SSL*. Im Gegensatz zum Skripset *Manifest* benötigt dieses deutlich mehr Zeit für die Analyse, weshalb nicht wie im vorherigen Experiment die *Top Free 100*, sondern lediglich die *Top Free 4* analysiert wurden. Das Ergebnis zeigt auf, dass zwischen

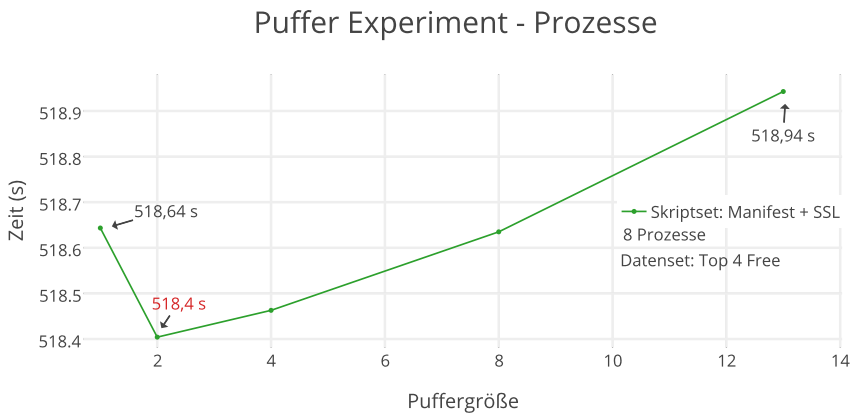

Abbildung 6.8: Puffer Experiment Manifest + SSL - Prozesse

schnellster und langsamster Analyse nur ein Unterschied von 0,54 Sekunden liegt.
Der Zeitgewinn von 4,19 Sekunden im vorherigen Experiment lässt sich wahrscheinlich darauf zurückführen, dass ca. 10.000 APKs mehr analysiert wurden und somit auch mehr I/O nötig war. Durch die schnelle Analyse eines einzelnen APKs müssen vermutlich mehrere Arbeiter gleichzeitig ihre Ergebnisse speichern, als dies der Fall bei einem rechenintensiveren Skriptset ist.
Zusammengefasst lässt sich also sagen, dass der Ergebnispuffer lediglich eine Verbesserung bei Skripten ohne Anforderungen erzielt. Somit könnte man anhand der minimalen Skript-Anforderungen, die sich aus den Anforderungen aller eingesetzten Skripte ergeben, dynamisch die Größe des Puffers festlegen.
Standardmäßig ist dieser mit einem Wert von eins deaktiviert.

6.4.2 Performanz in Relation zur Anzahl der Prozesse

Interessant ist außerdem die Relation zwischen der Anzahl von *Prozessen* und der benötigten Analysezeit. Ebenfalls mit dem Skriptset *Manifest* durchgeführt, zeigen die Ergebnisse[66] in Abbildung 6.9, dass mit steigender Anzahl die Analysedauer sinkt. Die schnellste Analyse erfolgte mit 16 *Prozessen*. Insgesamt konnte also im Vergleich zur Verwendung von nur einem Prozess eine Verbesserung um den Faktor 4,78 erreicht werden.
Folglich könnte man auch hier anhand der Skript-Anforderungen eine Optimierung vornehmen, indem die Anzahl der *Prozesse* für Analysen ohne Skript-Anforderungen erhöht wird. Das mit 16 *Prozessen* immer noch eine Verbesserung erzielt werden kann, ist damit erklärbar, dass die Analyse des Manifests nicht sehr CPU-intensiv ist und deshalb mit mehr *Prozessen* schneller Daten von der Festplatte gelesen werden können. ALL würde auf dem Testsystem standardmäßig acht *Prozesse* verwenden, da dies die Anzahl der Kerne ist, die von *multiprocessing.cpu_count()* zurückgeliefert wird (*Hyperthreading* mit eingeschlossen).
Das Verhalten wurde jedoch nicht für CPU-intensivere Skripte analysiert, da die Dau-

[65] Siehe `https://plot.ly/~nachtmaar/82`
[66] `https://plot.ly/~nachtmaar/109`

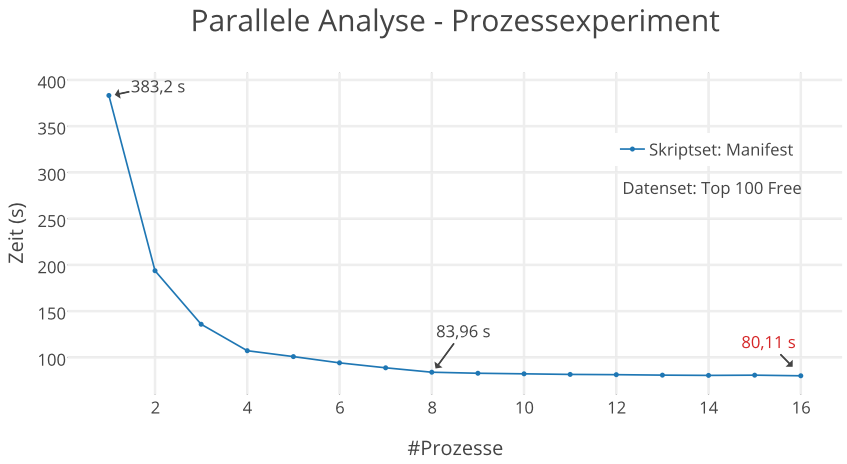

Abbildung 6.9: Parallele Analyse - Prozessexperiment

er der Skripte sehr stark schwankt. Kapitel 6.4.4 zeigt auf, dass je nach Skript die optimale Anzahl an *Prozessen* variiert.

6.4.3 Skript-Reset Experiment

Die *AndroScript*e implementieren das Interface *Resetable*, wodurch sie die Methode *reset()* bereitstellen und somit in ihren Initialisierungszustand zurückversetzt werden können.
Das nachfolgende Experiment versucht aufzuzeigen inwiefern sich dieses Verhalten auf die Performanz der Analyse auswirkt, indem das Zurücksetzen der Skripte mit dem Neuinitialisieren, daher dem Erstellen eines neuen Objekts, verglichen wird.
Dafür wurde ein *ChainedScript* erstellt, das aus zehn *AndroScript*en besteht. Jedes Skript loggt dabei 100-mal einen String mithilfe des *ResultObject*. Um Werte nicht zu überschreiben und das *ResultObject* immer gleich groß zu halten, werden zufällige Schlüssel generiert, die als Kategorie beim Loggen der Ergebnisse verwendet werden.
Die Messpunkte, also die Anzahl der Analysen, die mit dem *ChainedScript* durchgeführt wurden, sind alle Zweiterpotenzen zwischen eins bis einschließlich einer maximalen Anzahl von einer Millionen.
Das Ergebnis[67] in Abbildung 6.10 zeigt eine Performanzverbesserung, die jedoch in Anbetracht der Anzahl der Analysen nicht ins Gewicht fällt. Bei 131.721 Analysen mit dem *ChainedScript* ist ein Zeitunterschied von 16,57 Sekunden messbar. Bei der maximalen Anzahl von einer Millionen immerhin 127,94 Sekunden. Somit ist ein Geschwindigkeitsvorteil vom Faktor 1,25 an dieser Stellte messbar.
Auch wenn das Experiment in Relation zur Anzahl der Analysen zwar keinen enormen Zeitunterschied impliziert, wurde immerhin gezeigt, dass das Zurücksetzten der Skripte die Performanz nicht negativ beeinflusst.

6.4.4 Metrik für besseres Scheduling

Bei der Parallelisierung von Programmen ist neben der Identifikation der Komponenten, die überhaupt parallelisiert werden können, außerdem die Granularität der Aufgaben wichtig [SSP06]. Denn sind Aufgaben nicht weit genug zerlegt, kann trotz *dynamic scheduling* eine ungleichmäßige Lastverteilung vorkommen.
Während der Entwicklung wurde mit dem Skriptset *Misc2* und dem Datenset *Top 4*

[67] `https://plot.ly/~nachtmaar/80`

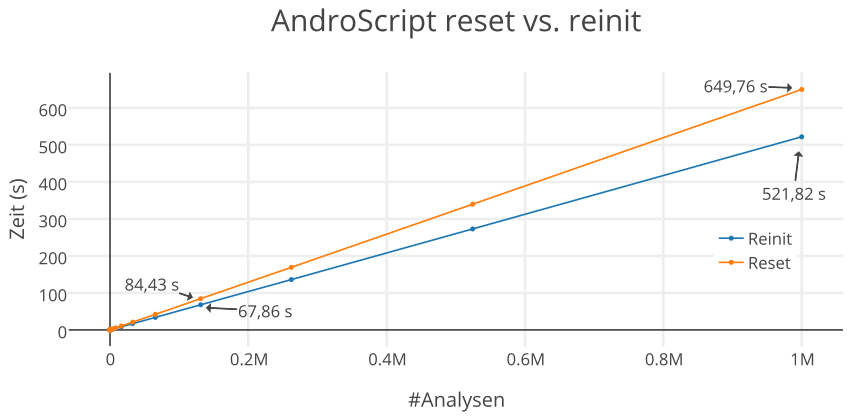

Abbildung 6.10: Skript Reset vs. Reinit Experiment

Free festgestellt, dass die Effizienz der Analyse sehr gering ist. Im Vergleich zur Verwendung von nur einem Prozess ist lediglich eine Geschwindigkeitsverbesserung um den Faktor 3,19 feststellbar (das Maximum ist vier und bedingt durch vier CPU-Kerne des Test-Systems). Abbildung 6.11 zeigt die Ergebnisse[68] dieser Analyse.

Der Grund hierfür liegt in einem ineffizienten *Scheduling*. Das Diagramm zeigt außer-

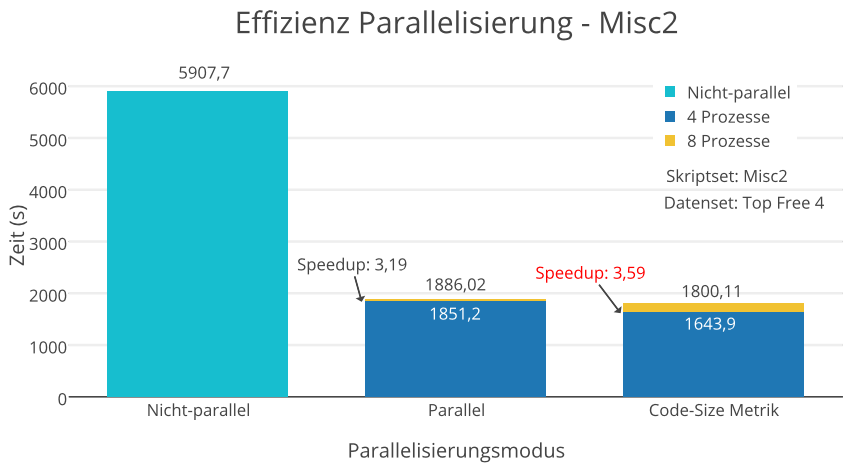

Abbildung 6.11: Effizienz Parallelisierung - Misc2

dem auf, dass mit dem verbesserten *Scheduling*, das nachfolgend vorgestellt wird, der Faktor von 3,19 auf 3,59 verbessert werden konnte.

Abbildung 6.12 vergleicht das normale *Scheduling* mit dem verbesserten *Scheduling*[69]. Zu sehen ist die Verteilung der Analysejobs auf vier Arbeiter bzw. *Prozesse*. Auf der linken Seite ist die Verteilung der Arbeit in ihrer ursprünglichen Reihenfolge visualisiert. Auf der rechte Seite wurde eine Metrik benutzt, sodass die Jobs, die am längsten zur Ausführung benötigen zuerst abgearbeitet werden. Da die meisten Skripte auf dem Quellcode der Applikation basieren, wurde sich für die Größe der *classes.dex* entschieden. Diese wird beim Importieren in die Datenbank berechnet.

Der Grund für die schlechte Lastverteilung ist ein langer Analysejob (Dekompilieren des Quellcodes), der zur Abarbeitung die Hälfte der gesamten Analysezeit benötigt und als letzte Aufgabe an Prozess1 verteilt wird. Die Granularität kann jedoch ohne Veränderung des *DAD-Decompiler* von *androguard* nicht beeinflusst werden. Durch Benutzung der Metrik wird dieser Job jedoch früher an einen Arbeiter verteilt und somit Parallelität länger ausgenutzt.

[68] `https://plot.ly/~nachtmaar/134`
[69] `https://plot.ly/~nachtmaar/158`

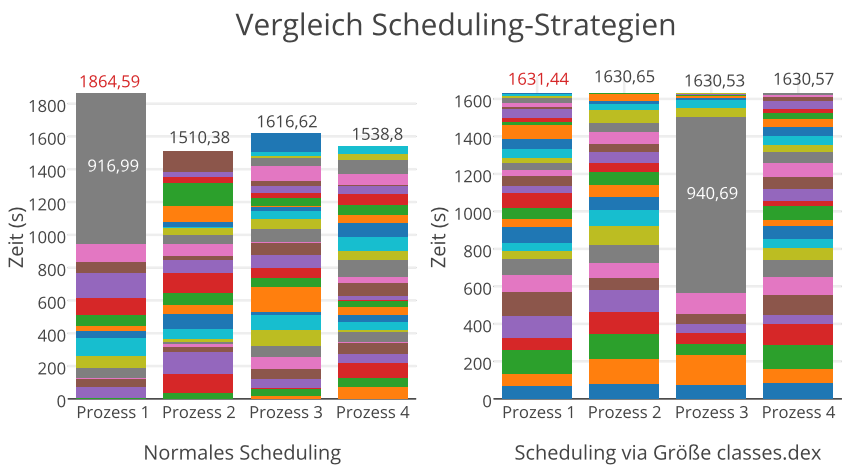

Abbildung 6.12: Vergleich Scheduling-Strategien

Da es sich allerdings auch um einen Zufall handeln kann, dass das *Scheduling* effizienter ist, wurde zusätzlich das gleiche Datenset noch mit dem Skript *Misc1* getestet. Dabei handelt es sich um das gleiche Skriptset wie *Misc2*, jedoch erfolgt keine Dekompilation, die den langen Analysejob hervorruft.

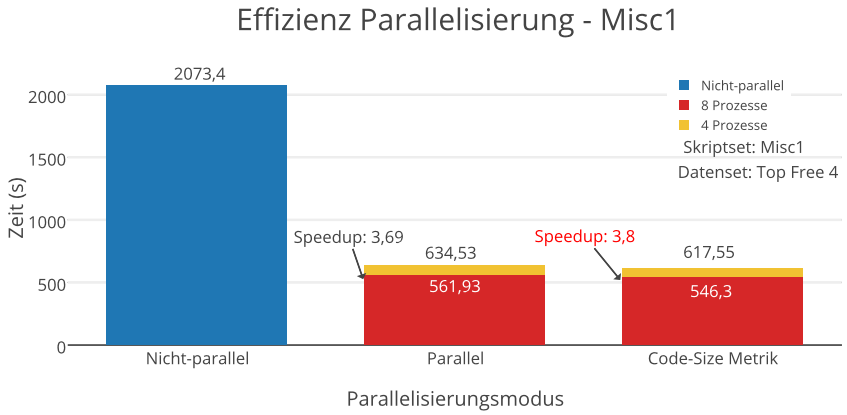

Abbildung 6.13: Effizienz Parallelisierung - Misc1

Abbildung 6.13 zeigt, dass auch hier eine Verbesserung[70] erzielt werden konnte (15,63 Sekunden). Insgesamt wurde ein Speedup von 3,8 erreicht, sodass die Parallelisierung sehr effizient ist. Das normale *Scheduling* erreicht dagegen lediglich den Faktor 3,69. Interessant ist außerdem, dass für das Skriptset *Misc2* die Verwendung von nur vier *Prozessen* effizienter ist, während Skriptset *Misc1* mit acht *Prozessen* schneller ist.
Eine andere Idee, um das *Scheduling* weiter zu verbessern, ist, die Analysejobs feinkörniger zu gestalten. Das bedeutet, dass pro Skript ein Job und nicht ein Job für ein APK mit allen Skripten definiert wird. Jedoch müsste die Bereitstellung der Analyse-Objekte nur einmalig geschehen, da dieser Vorgang sehr zeitintensiv ist. Ferner müssten die Analyse-Objekte und das jeweilige APK pro Skript via *shared memory* verfügbar sein. Jobs mit mehreren zeitintensiven Skripten könnten so eine bessere Lastverteilung erreichen. Allerdings könnte die Benutzung von *shared memory* und dem mehrfachen Bereitstellen der *Apk*-Klasse zusätzliche Ressourcen benötigen. Dieser Ansatz wurde nicht ausgetestet, ist also für zukünftige Arbeiten interessant.

[70] `https://plot.ly/~nachtmaar/137`

6.5 Verteilt ausgeführte Experimente

Die verteilt ausgeführten Experimente weisen das folgende Setup auf: Auf dem physikalischen Testsystem, dessen technische Daten in 6.1 beschrieben wurden, lief der *MongoDB*-Service für die Ergebnisspeicherung und APK-Verteilung.
RabbitMQ wurde auf einer virtuellen Maschine als *Nachrichten-Broker* installiert, damit das Senden der APK-Rohdaten ebenfalls in die Experimente mit einbezogen werden konnte, denn zum Initiieren der Analyse wurde das physikalische Testsystem ausgewählt.
Auf jedem der vier Computer wurde ein *Celery-Worker* mit acht *Prozessen* gestartet.

6.5.1 Nachrichtenpersistenz und SSL

RabbitMQ unterscheidet bei Nachrichten zwischen flüchtigen und persistenten Nachrichten. Letzere werden zwingend auf der Festplatte niedergeschrieben und sind folglich nach einem Absturz des Servers immer noch vorhanden.
Flüchtige Nachrichten hingegen können, müssen aber nicht, zwischengespeichert werden, sodass diese teilweise aus dem Speicher beantworten werden können.
Abbildung 6.14 zeigt ein Experiment auf, bei dem die Performanz in Relation zur Nachrichtenform gemessen wurde. Außerdem wurde untersucht inwieweit Verschlüsselung die Leistung verschlechtert.

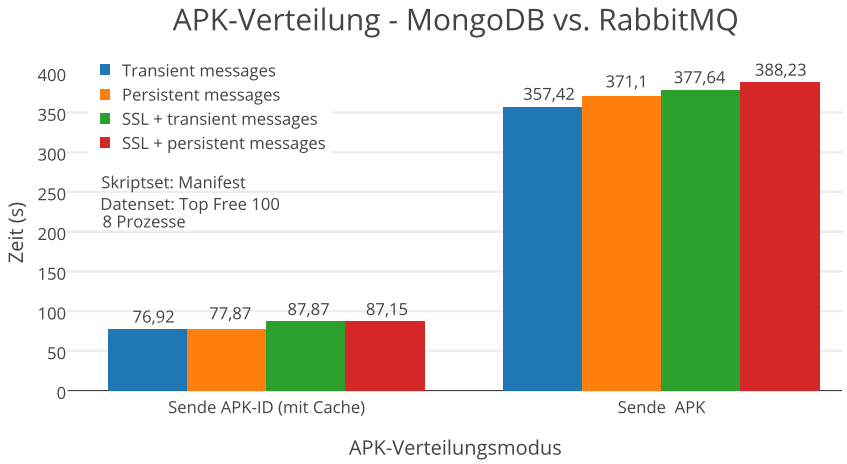

Abbildung 6.14: Vergleich APK-Verteilungsmodus - MongoDB vs. RabbitMQ

Gemessen wurde die Geschwindigkeit des Nachrichtenaustauschs, indem lediglich das Skriptset *Manifest* benutzt wurde, sodass ein einzelnes APK sehr schnell analysiert ist (das APK muss lediglich entpackt und Daten aus der Datei *AndroidManifest.xml* geparsed werden). Außerdem wurde das Empfangen der Ergebnisse aus der Datenbank ausgeschaltet, sodass lediglich der Datendurchsatz gemessen werden kann. Das Datenset besteht aus den 100 beliebtesten Apps aus allen Kategorien und weist eine Gesamtgröße von 22.315 Megabyte auf.
Verglichen wurde außerdem der Unterschied zwischen Senden der APK-Rohdaten und der APK-ID. Dabei wurde festgestellt, dass *MongoDB* bereits bei der zweiten Wiederholung des Experimentes einen deutlichen Geschwindigkeitsschub erzielen kann. Dieser kann durch *Caching-Verhalten* erklärt werden, denn *MongoDB* benutzt sogenannte *memory mapped files*[71]. Dabei handelt es sich um Dateien, die im Arbeitsspeicher ab-

[71] Siehe `http://docs.mongodb.org/manual/faq/storage`. Eingesehen am 28.07.2014

gebildet werden. Da die Größe des APK-Sets in den Speicher des *MongoDB*-Servers passt, zeigt sich der Effekt.
Des Weiteren wurde versucht die Geschwindigkeit der APK-Verteilung durch *MongoDB* ohne den *Caching-Effekt* zu untersuchen, erwies sich jedoch als schwierig. Das *Caching* wird durch den *Kernel* implementiert, sodass versucht wurde den *MongoDB*-Dienst zu stoppen, die Speicherseiten freizugeben[72] und wieder zu starten. Die Verteilung wurde so zwar deutlich langsamer und der Speicher wurde auch tatsächlich freigegeben, aber dennoch konnte nach mehreren Wiederholungen noch eine Performanzsteigerung festgestellt werden, sodass im Diagramm nur das Verhalten mit *Caching* abgebildet ist.
Der erste Messpunkt (21-fache Wiederholung) wurde verworfen, damit das *Cache*-Verhalten sichtbar wird.
Bei der Verwendung von flüchtigen Nachrichten ohne *SSL* konnte mit *Caching* ein Speedup von 4,65 im Vergleich zur APK-Serialisierung festgestellt werden.
Des Weiteren ist ersichtlich, dass das Zwischenspeichern der Nachrichten auf dem *RabbitMQ*-Server zu Geschwindigkeitseinbußen bei großen Nachrichten (mit Rohdaten) führt, da diese persistent auf der Festplatte gespeichert werden müssen.
Beide Modi zeigen auf, dass der Einsatz von *SSL* die Analysedauer negativ beeinträchtigt. Denn einerseits sind Nachrichten größer und andererseits impliziert das Verschlüsseln einen zusätzlichen Verbrauch von CPU-Ressourcen.
Beim Senden der APK-Rohdaten wurde vermutlich die Serialisierung, die nicht parallelisiert wurde, sowie die Festplatte zum Flaschenhals. Denn direkt nach Veröffentlichung aller Analysejobs waren alle Aufgaben abgearbeitet.
Die optimale Einstellung für maximale Performanz sind also flüchtige Nachrichten ohne die Benutzung von *SSL*, weshalb die nachfolgenden Experimente diese benutzen.

6.5.2 Datendurchsatz

Das nachfolgende Experiment versucht den Datendurchsatz bei der APK-Verteilung zwischen *RabbitMQ* und *MongoDB* noch weiter zu untersuchen. Dafür wurde ein so großes APK-Set gewählt (*Top Free 500*, 91.764 Megabyte), dass dieses nicht vollständig in den Speicher passt und folglich *Caching* nicht erfolgen kann. Dennoch wurde vor jedem Durchlauf *MongoDB* gestoppt und alle unbenutzten Seiten aus dem Kernel-Cache entfernt. Außerdem wurde ebenfalls das Empfangen der Ergebnisse aus der Datenbank ausgeschaltet, sodass lediglich der Datendurchsatz gemessen werden kann.
Es wurde wieder das Skriptset *Manifest* benutzt. Das Ergebnis in Abbildung 6.15 veranschaulicht[73] einen großen Unterschied in Bezug auf die Analysedauer. Die Verteilung via *MongoDB* ist um den Faktor 1,59 langsamer.
RabbitMQ lädt die Nachrichten und somit auch die Rohdaten im Falle der APK-Serialisierung vor. Wird lediglich der APK-Hash gesendet, kann erst bei Ausführen des Analysejobs das Laden der APK-Rohdaten initiiert werden. Da Nachrichten jedoch langsamer ankommen als sie bearbeitet werden können (siehe Erkenntnis aus Experiment 6.14), muss die Verteilung durch *MongoDB* generell langsamer sein.
Die APKs wurden bereits mit der maximalen Größe für *Chunks* importiert, sodass die Rekonstruktion des APKs so schnell wie möglich erfolgen kann.

[72] Siehe `https://www.kernel.org/doc/Documentation/sysctl/vm.txt`. Eingesehen am 28.07.2014
[73] `https://plot.ly/~nachtmaar/181`

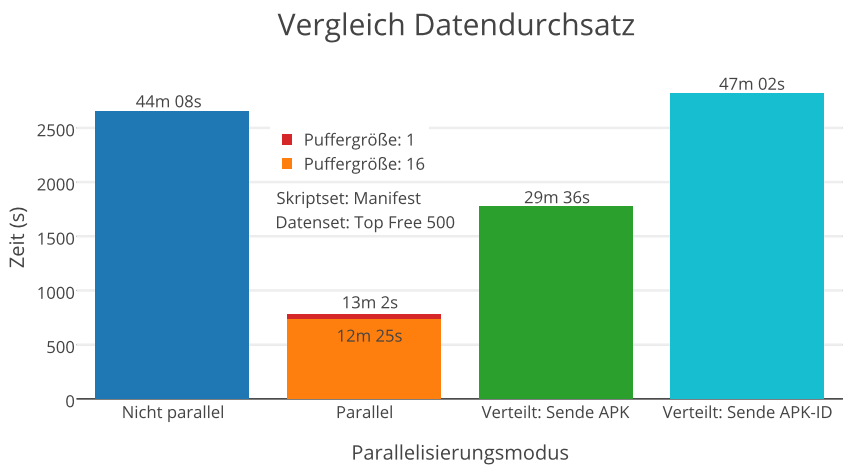

Abbildung 6.15: Vergleich Parallelisierung - Manifest

Die Gründe für die schlechte Performanz seitens *MongoDB* müssen folglich im Rahmen zukünftiger Arbeiten weiter untersucht werden. Denn im Gegensatz zur Serialisierung der APKs sind diese bei Verwendung der *MongoDB* bereits vorverteilt, sodass das eigentliche Senden der Nachrichten nicht zum Flaschenhals werden kann und ggf. mehrere *MongoDB*-Instanzen die Daten verteilen können.
Zusätzlich zeigt die Abbildung, dass die schnellste Bearbeitung des gesamten Analysejobs im parallelen Modus durchgeführt werden konnte. Mit den Resultaten aus den parallel ausgeführten Experimenten, konnte mit einer Puffergröße von 16 ein zusätzlicher Geschwindigkeitszuwachs von 37,36 Sekunden erreicht werden.
Im Vergleich zum Senden der APK-Rohdaten ist die lokale parallele Analyse 2,27-mal schneller und mit einem größerem Ergebispuffer sogar 2,38-mal so schnell. Die verteilte parallele Analyse ist jedoch immer noch schneller als eine lokale, nicht parallelisierte Analyse.

6.5.3 Parallel versus verteilt

Das nachfolgende Experiment zeigt wie effizient die verteilte Analyse ist. Als Referenzimplementierung dient dabei die parallele Analyse. Mit vier Computern wäre maximal eine Verbesserung von vier möglich. Dazu kommt allerdings noch der *Overhead* des verteilten Systems und der drei virtuellen Maschinen, die nicht die gleiche Leistung wie das physikalische Testsystem erbringen können.
Sowohl die parallele als auch die verteilte Analyse wurde mit acht Prozessen und dem Skriptset *Manifest + SSL* durchgeführt. Abbildung 6.16 zeigt[74] den Vergleich beider Modi.
Auf der linken Seite sind die Ergebnisse mit dem APK-Set *Top Free 4* zu sehen, welche aufzeigen, dass der verteilte Modus lediglich einen Speedup von 2,17 und 2,41 erzielen kann. Der erste Wert zeigt die Verbesserung ohne Benutzung einer Metrik und der zweite die Steigerung mit Benutzung der Codegröße als Metrik.
Mit einem größeren APK-Set (*Top Free 100*) kann jedoch ein Speedup von 3,26 und 3,31 erreicht werden.
Das Ergebnis lässt sich durch das Reservieren von Aufgaben durch *Celery* erklären, das nicht für eine so kleine Anzahl an Jobs optimiert ist. Ausgestellt werden kann das *Feature* allerdings nicht und ist für das Vorladen der APK-Daten für größere Analysejobs auch sinnvoll. ALL setzt den Wert so klein wie möglich, sodass jeder *Prozess*

[74] `https://plot.ly/~nachtmaar/179`

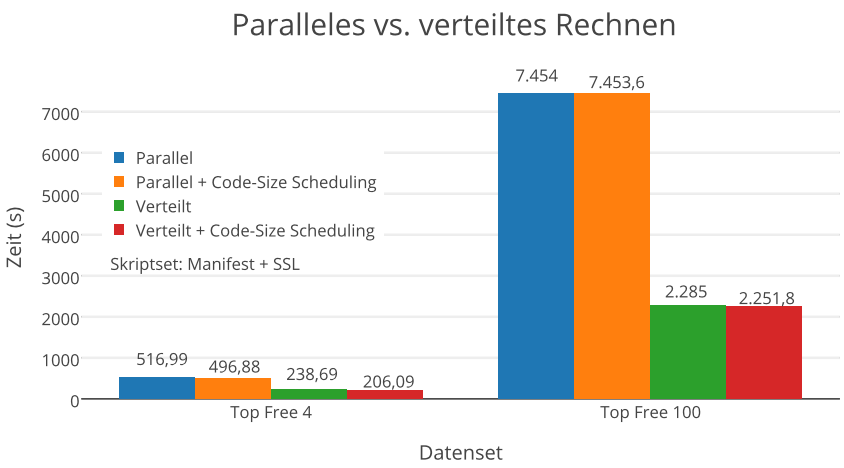

Abbildung 6.16: Vergleich paralleles versus verteiltes Rechnen

eine Aufgabe reserviert. Das kann dazu führen, dass ein Arbeiter bereits alle Aufgaben bearbeitet hat, jedoch keine Neuen vorhanden sind, da ein beschäftigter Arbeiter im *worst case* weitere acht Aufgaben reserviert hat.
Zusätzlich zeigt sich mit steigender Anzahl an Arbeitern und sinkender Anzahl an Analysejobs umso mehr, dass die vollständige parallele Ausführung gegen Ende nicht aufrecht erhalten werden kann. Die letzten 31 Jobs werden nicht optimal parallelisiert, da sie von weniger als 32 Prozessen bearbeitet werden.
Das Experiment zeigt außerdem, dass das *Scheduling* durch die Codegröße wieder Geschwindigkeitsvorteile erzielen konnte. Sowohl im parallelen als auch verteilten Modus. Das Experiment wurde jedoch nur zweimal[75] durchgeführt.

6.5.4 Analyse Top Free 500

Zum Schluss wird noch eine Zeitmessung verschiedener Skripte basierend auf dem APK-Set *Top Free 500* präsentiert, um einen besseren Eindruck einerseits über die Leistung von ALL und andererseits über die Laufzeit der Skripte zu bekommen.
Da sich in den Experimenten das Senden der Rohdaten als schneller erwies, wurde dieser Modus für das Experiment genutzt, dessen Ergebnisse in Abbildung 6.17 visualisiert sind. Die Messung mit dem jeweiligen Skript wurde nur einmal durchgeführt.

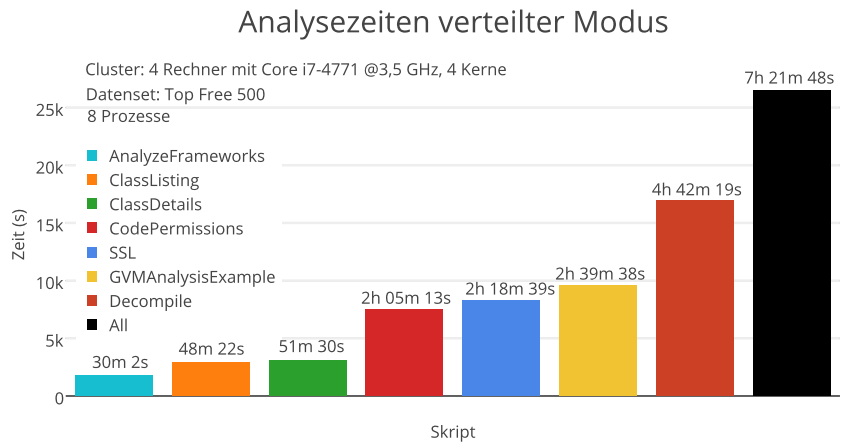

Abbildung 6.17: Analysezeiten verteilter Modus - Top Free 500

[75] Die Standardabweichungen sind respektive der im Diagramm gegebenen Reihenfolge: 1.81, 0.43, 1.58, 13.56, 1.77, 4.56, 13.26, 2.18

Serialisierung und Laden der APKs von der Festplatte werden dem Skript *Analyze-Frameworks* zum Verhängnis, das keine Skript-Anforderungen stellt. Denn kurz nach Veröffentlichung aller Analysejobs hatten die Worker bereits alle Aufgaben ausgeführt. *ClassListing* und *ClassDetails* benötigen das *DalvikVMFormat*-Objekt und erfordern folglich nur das Erstellen des *Disassembly*.
Das Skript *CodePermissions* benötigt zwar nur das *VMAnalysis*-Objekt, der zusätzliche Overhead lässt sich allerdings durch das selektive Dekompilieren der Methoden erklären, die Android-Berechtigungen benötigen. *SSL* benötigt die Querverweise zwischen Methoden (*XREF*), wodurch zusätzliche Ressourcen benötigt werden.
Das Erstellen eines Graphen, der die Methodenaufrufe visualisiert, dauert knapp 31 Minuten länger. Mit über 4 Stunden dauert das vollständige Dekompilieren von 12.689 APKs mit einer Größe von 91,76 Gigabyte am längsten.
Addiert man die Zeiten der Analysen mit nur einem Skript zusammen, lässt sich feststellen, dass die gebündelte Analyse aller Skripte 6 Stunden und 34 Minuten weniger Zeit benötigt. Der Sachverhalt lässt sich durch die Bereitstellung der Analyse-Objekte erklären, die pro Analyse erstellt werden müssen.

7 Fazit

Mit ALL wurde zusätzlich zu *PlayDrone* ein neues skalierbares Analysewerkzeug für Android eingeführt. Unterstützt durch die Funktionalität von *androguard* können APKs disassembliert und dekompiliert werden. Außerdem können Querverweise genutzt werden, um Beziehungen zwischen Methoden und Feldern aufzufinden.
Durch die Einbettung der Analysefunktionalität, in ein einfaches und auf Performanz optimiertes Skript-Framework, können effiziente statische Analysen durchgeführt werden. Dynamische Analyse ist prinzipiell auch möglich, da in dem Skript der Zugriff auf die Rohdaten des APK möglich ist. Jedoch stehen keine Hilfswerkzeuge zur Unterstützung der dynamischen Analyse zur Verfügung.
Ergebnisse können strukturiert geloggt und zum einfachen Durchstöbern lokal als JSON-Dateien und zusätzlich in einer *NoSQL*-Datenbank abgelegt werden, sodass komplexe Abfragen auf den Resultaten durchgeführt werden können.
Das Verwalten selbst großer APK-Kollektionen ist durch Filterung via Tags, Hashes und Paketnamen oder separaten *SQLite*-Datenbanken möglich.

7.1 Kritische Bewertung

Die Evaluierung der Messungen zeigt die erfolgreiche Steigerung der Performanz durch verschiedene Methoden.
Der wichtigste Punkt ist die Parallelisierung durch die Verwendung von *Prozesssen* und das Bereitstellen eines verteilten Systems, um horizontale Skalierung zu implementieren.
Durch Einführen von Skript-Anforderungen und somit einer bedarfsgesteuerten Bereitstellung der Analyse-Objekte, konnte eine enorme Steigerung der Effizienz erreicht werden. Zusätzlich wurde gezeigt, dass im lokalen parallelen Modus die Benutzung von mehr *Prozessen*, als standardmäßig durch ALL vorgesehen, für Skripte ohne Anforderungen, einen weiteren Leistungsschub hervorbringen kann. Zusätzlich ist ein größerer Ergebnispuffer zur Bündelung von I/O-Aktivitäten sinnvoll.
Basierend auf den Skript-Anforderungen könnte man optimierte Einstellungen vor der Analyse festlegen. Außerdem wurde gezeigt, dass das Scheduling mithilfe der Größe der *classes.dex* eine gleichmäßigere Lastverteilung erzielen kann, weshalb die Metrik

standardmäßig benutzt werden sollte.
Die Ergebnisse der verteilten Experimente haben aufgezeigt, dass flüchtige Nachrichten (*transient messages*) zu einer schnelleren Übermittlung an die Worker führt und genau dann praktikabel ist, wenn dem Analysten Performanz wichtiger ist als persistente Jobs, die selbst bei einem Systemausfall des Nachrichten-Brokers noch verfügbar sind.
Festgestellt wurde, dass Festplatte und wahrscheinlich das Serialisieren der Nachrichten, zumindest wenn die Rohdaten der APKs enthalten sind, als erstes zum Flaschenhals werden und nicht etwa das Netzwerk. Somit ist für zukünftige Arbeit eine parallele Serialisierung der Nachrichten interessant. Für die Benutzung der *MongoDB* als APK-Verteiler sind für einen effizienten Einsatz noch weitere Optimierung, wie z.B. das Vorladen der Android-Applikationen, essentiell. Auch hier muss weiter geforscht werden, wie die APK-Verteilung schneller implementiert werden kann, sodass für Anwendungsfälle, bei denen das Senden der APKs durch den Client zum Flaschenhals wird, duch Vorverteilung auf mehrere *MongoDB-Shards* gelöst werden kann.

7.2 Ausblick

Neben den Verbesserungen, die durch Auswertung der Messungen erzielt werden konnten, bieten sich außerdem noch eine Reihe weiterer Optimierungen und zukünftige Arbeiten an:

- **Skripte**: Die Messungen haben aufgezeigt, dass die Skript-Anforderungen vereinfacht werden können, indem *DalvikVMFormat*, *VMAnalysis* und *GVMAnalysis* zusammengeführt werden.
  Außerdem könnte ein Cache für dekompilierte Methoden eingeführt werden, sodass dies nicht mehrfach erfolgen muss. Interessant wäre außerdem eine benutzerdefiniert Auswahl des *Decompiler*. Des Weiteren könnten auf *androguard*-basierte Projekte wie z.B. *Androwarn* leicht in ALL integriert werden. Nicht fehlen darf außerdem die Entwicklung weiterer Skripte. Eine integrierte Umgebung zur Entwicklung würde das Schreiben noch weiter vereinfachen.

- **Logging**: Genau wie man Werte in eine Aufzählung loggen kann, könnte eine Set-Funktionalität nützlich sein, sodass Ergebnisse einer Aufzählung nur hinzugefügt werden, wenn sie nicht bereits in dieser vorliegen.

- **Import-Datenbank**: Zum besseren Filtern der APKs könnte man reguläre Ausdrücke erlauben.

- **Celery**: Durch Erstellen mehrerer Queues, die der Leistung der Worker entsprechen, könnte ein besseres Routing durch Code -Größe und Skript-Anforderungen vorgenommen werden. Somit werden lange Aufgaben nicht an Arbeiter mit wenig Ressourcen verteilt.
  Um ein besseres Feedback über die Analyse zu geben, könnten eigene *remote control commands* für *Celery* geschrieben werden, damit Ressourcen wie CPU, Speicher und Netzkwerk der Arbeiter im User Interface visualisiert werden können. Die Verwendung von *msgpack* als Serialisierer könnte zu kleineren Nachrichten und einer schnelleren Serialiserung beitragen[76]

[76]Siehe Vergleich verschiedener Serialisierer in Python: `http://jmoiron.net/blog/python-serialization/`

- **GUI**: Die grafische Oberfläche kann noch benutzerfreundlicher gestalten werden. Außerdem sind einige Funktionen wie der Update-Mechanismus oder das Scheduling via Codegröße noch nicht integriert.

- **Analyse**: Die lokale Analyse könnte ebenfalls mit Zeitlimits für Jobs ausgestattet werden, indem der von *Celery* benutzte *Fork* der Bibliothek *multiprocessing* namens *billiard*[77] benutzt wird.
  Interessant wäre außerdem die Benutzung von *Celery* für periodische Analysen sowie der Einführung eines Event-Systems, sodass, wenn ein definiertes Event ausgelöst wird, z.B. eine E-Mail an den Analysten gesendet wird. Das Feature könnte zum Auffinden besonders kritischer Bugs benutzt werden, sodass der Analyst die Analyse nicht jedes mal selbst anstoßen sowie anschließend die Ergebnisse auswerten muss.

Zusammengefasst lässt sich also sagen, dass mit ALL ein skalierbares Analysewerkzeug für Android-Applikationen entwickelt wurde, das sowohl für statische als auch dynamische Analyse benutzt werden kann.

[77] `https://github.com/celery/billiard`

Abbildungsverzeichnis

Code Listings

Literatur

Wissenschaftliche Artikel

[Bra10] Stefan Brahler. Analysis of the android architecture. *Karlsruhe institute for technology*, 2010.

[DG11] Anthony Desnos and Geoffroy Gueguen. Android: From reversing to decompilation. *Proc. of Black Hat Abu Dhabi*, 2011.

[Dos14] David Dossot. *RabbitMQ Essentials*. Packt Publishing Ltd, 2014.

[DPLR] Roberto Di Pietro, Flavio Lombardi, and Sara Rossicone. Modeling mobile resource security.

[EBFK13] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 73–84. ACM, 2013.

[FBV09] Rune Møllegaard Friborg, John Markus Bjørndalen, and Brian Vinter. Three unique implementations of processes for pycsp. In *CPA*, pages 277–292, 2009.

[FHM+12] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why eve and mallory love android: An analysis of android ssl (in) security. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 50–61. ACM, 2012.

[Gar11] Marko Gargenta. *Einführung in die Android-Entwicklung*. O'Reilly Germany, 2011.

[Gun12] Sheran Gunasekera. *Android Apps Security*. Springer, 2012.

[HUHS13] Johannes Hoffmann, Martin Ussath, Thorsten Holz, and Michael Spreitzenbarth. Slicing droids: program slicing for smali code. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1844–1851. ACM, 2013.

[Kri04] Jens Krinke. Advanced slicing of sequential and concurrent programs. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 464–468. IEEE, 2004.

[KYY+12] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, Junbum Shin, and SWRD Center. Scandal: Static analyzer for detecting privacy leaks in android applications. *MoST*, 2012.

[LBH13] Monte Lunacek, Jazcek Braden, and Thomas Hauser. The scaling of many-task computing approaches in python on cluster supercomputers. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–8. IEEE, 2013.

[Nol12] Godfrey Nolan. Inside the dex file. In *Decompiling Android*, pages 57–92. Springer, 2012.

[PS12] Étienne Payet and Fausto Spoto. Static analysis of android programs. *Information and Software Technology*, 54(11):1192–1201, 2012.

[SBS+09] A-D Schmidt, Rainer Bye, H-G Schmidt, Jan Clausen, Osman Kiraz, Kamer A Yuksel, Seyit Ahmet Camtepe, and Sahin Albayrak. Static analysis of executables for collaborative malware detection on android. In *Communications, 2009. ICC'09. IEEE International Conference on*, pages 1–5. IEEE, 2009.

[SSP06] M Sasikumar, Dinesh Shikhare, and P Ravi Prakash. *Introduction to parallel processing*. Prentice Hall of India, 2006.

[Tan09] Andrew S Tanenbaum. *Moderne Betriebssysteme*, volume 7342. Pearson Deutschland GmbH, 2009.

[TB11] Bogdan George Tudorica and Cristian Bucur. A comparison between several nosql databases with comments and notes. In *Roedunet International Conference (RoEduNet), 2011 10th*, pages 1–5. IEEE, 2011.

[TvS08] Andrew S Tanenbaum and Maarten van Steen. Verteilte systeme-prinzipien und paradigmen (2. aufl.)., 2008.

[VGN14] Nicolas Viennot, Edward Garcia, and Jason Nieh. A measurement study of google play. In *The 2014 ACM international conference on Measurement and modeling of computer systems*, pages 221–233. ACM, 2014.

[WNL+14] Lukas Weichselbaum, Matthias Neugschwandtner, Martina Lindorfer, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. Andrubis: Android malware under the magnifying glass. *Vienna University of Technology, Tech. Rep. TRISECLAB-0414-001*, 2014.

Online-Ressourcen

[Akd] Akdeniz. Google play crawler java api. `https://github.com/Akdeniz/google-play-crawler`. Zuletzt eingesehen am 13.07.2014.

[andb] Android kitkat. https://developer.android.com/about/versions/kitkat.html. Zuletzt eingesehen am 14.06.2014.

[andc] Android managing projects. https://developer.android.com/tools/projects/index.html. Zuletzt eingesehen am 15.06.2014.

[andd] Android manifest element. https://developer.android.com/guide/topics/manifest/manifest-element.html. Zuletzt eingesehen am 15.06.2014.

[ande] App manifest. https://developer.android.com/guide/topics/manifest/manifest-intro.html. Zuletzt eingesehen am 11.06.2014.

[andf] Application fundamentals. https://developer.android.com/guide/components/fundamentals.html. Zuletzt eingesehen am 14.06.2014.

[andg] Content provider basics. https://developer.android.com/guide/topics/providers/content-provider-basics.html. Zuletzt eingesehen am 14.06.2014.

[andh] Intents and intent filters. https://developer.android.com/guide/components/intents-filters.html. Zuletzt eingesehen am 15.06.2014.

[andi] Meet android. http://www.android.com/meet-android/. Zuletzt eingesehen am 14.06.2014.

[Bea10] David Beazley. Understanding the python gil. http://www.dabeaz.com/python/UnderstandingGIL.pdf, 2010. Zuletzt eingesehen am 17.06.2014.

[cela] Celery designing workflows. http://docs.celeryproject.org/en/latest/userguide/canvas.html#guide-canvas. Zuletzt eingesehen am 07.07.2014.

[celb] Celery: Distributed task queue. http://www.celeryproject.org. Zuletzt eingesehen am 07.07.2014.

[celc] Celery frequently asked questions. http://celery.readthedocs.org/en/latest/faq.html.

[celf] Celery tasks. http://celery.readthedocs.org/en/latest/userguide/tasks.html. Zuletzt eingesehen am 13.07.2014.

[celg] Celery time limits. http://docs.celeryproject.org/en/latest/userguide/workers.html#worker-time-limits. Zuletzt eingesehen am 07.07.2014.

[celh] Celery workers. http://celery.readthedocs.org/en/latest/userguide/workers.html#max-tasks-per-child-setting. Zuletzt eingesehen am 19.07.14.

[celi] Introduction to celery. http://docs.celeryproject.org/en/latest/getting-started/introduction.html. Zuletzt eingesehen am 07.07.2014.

[D.] Thomas D. Yet another static code analyzer for malicious android applications. https://github.com/maaaaz/androwarn.

[Des] Anthony Desnos. Reverse engineering, malware and goodware analysis of android applications ... and more (ninja !). https://code.google.com/p/androguard/. Zuletzt eingesehen am 28.07.2014.

[dex] Dexter documentation. http://dexter.dexlabs.org/static/docs/. Zuletzt eingesehen am 11.06.2014.

[ipy] Ipython parallel intro. http://ipython.org/ipython-doc/stable/parallel/parallel_intro.html. Zuletzt eingesehen am 02.07.14.

[Mas] Mashable. Google play hits 1 million apps. http://mashable.com/2013/07/24/google-play-1-million/. Zuletzt eingesehen am 27.06.2014.

[mona] Mongodb faq. http://docs.mongodb.org/manual/faq/developers. Zuletzt eingesehen am 21.07.14.

[monb] Mongodb gridffs. http://docs.mongodb.org/manual/core/gridfs/. Zuletzt eingesehen am 12.07.2014.

[monc] Mongodb limits and thresholds. http://docs.mongodb.org/manual/reference/limits/. Zuletzt eingesehen am 10.07.2014.

[mond] Mongodb sharding. http://docs.mongodb.org/manual/sharding/. Zuletzt eingesehen am 13.07.2014.

[pyta] Python 2 threading bibliothek. https://docs.python.org/2/library/threading.html. Zuletzt eingesehen am 01.07.2014.

[pytc] Python pickle. https://docs.python.org/2/library/pickle.html. Zuletzt eingesehen am 13.07.2014.

[rab] Rabbitmq clustering. https://www.rabbitmq.com/clustering.html. Zuletzt eingesehen am 13.07.2014.

[Sva14] Vanja Svajcer. Sophos mobile security threat report. http://www.sophos.com/en-us/medialibrary/PDFs/other/sophos-mobile-security-threat-report.pdf, 2014. Zuletzt eingesehen am 30.07.2014.

Appendix

Beispiel AndroScript

Code Listing 1: ClassDetails - Listet Klassen zusammen mit ihren Methoden und Feldern auf

```
1  from androlyzelab.model.script.AndroScript import AndroScript
2
3  # categories
4  CAT_CLASS_DETAILS = "class details"
5  CAT_METHODS = "methods"
6  CAT_FIELDS = "fields"
7
8  class ClassDetails(AndroScript):
9    ''' Retrieve all classes and their methods and fields '''
10
11   VERSION = "0.1"
12
13   def _analyze(self, apk, dalvik_vm_format, vm_analysis, gvm_analysis,
         *args, **kwargs):
14     # get `ResultObject`
15     res = self.res
16
17     # get classes
18     classes = dalvik_vm_format.get_classes()
19
20     # run over classes
21     for c in classes:
22       ROOT_CAT = (CAT_CLASS_DETAILS, c.name)
23       res.register_keys([CAT_METHODS, CAT_FIELDS], *ROOT_CAT)
24
25       # log methods
26       res.log(CAT_METHODS, [mn.name for mn in mc.get_methods()], *
             ROOT_CAT)
27
28       # log fields
29       res.log(CAT_FIELDS, [fn.name for fn in c.get_fields()], *ROOT_CAT
             )
30
31   def needs_dalvik_vm_format(self):
32     ''' The need the `DalvikVMFormat` object '''
33     return True
34
35 if __name__ == '__main__':
36   # use test method during script development
```

```
37  for res in AndroScript.test(ClassDetails, ["apks/a2dp.Vol.apk"]):
38    print res.write_to_json()
```

Built-in Skripts

Manifest

Name	*Beschreibung*
Activities	Listet alle Acitivies sowie die Main-Activity auf
BroadcastReceivers	Listet alle BroadcastReceivers auf
ContentProviders	Listet alle ContentProvider auf
Services	Listet alle Services auf
Files	Zeigt alle Datein an, die in dem APK vorhanden sind
Libs	Zeigt die benutzen Bibliotheken
Permissions	Welche Berechtigungen werden im Manifest definiert
ApkInfo	Fast alle oben beschriebenen Manifest-Informationen in einem *AndroScript* zusammen

DVM

Name	*Beschreibung*
ClassListing	Listet alle Klassen auf
ClassDetails	Zeigt alle Klassen inklusive Methoden und Feldern an
Decompile	Dekompiliert alle Methoden und speichert den Quellcode unter der Kategorie Klasse und Methodennamen
DecompileText	Dekompiliert und speichert als Text-Datei ab. Ergebnisse werden im *GridFS* von *MongoDB* abgelegt

ChainedScript

Name	*Beschreibung*
DVM	Kombiniert ClassListing, ClassDetails und Decompile
ChainedApkInfos	Fügt alle Manifest-Skripte zusammen. Bietet die Gleiche Funktionalität wie *ApkInfo*, jedoch als *ChainedScript*

Misc

Name	*Beschreibung*
AnalyzeFrameworks	Sucht nach den Frameworks *Titanium*, *Xamarin*, *Rhomobile* und *Phonegap* (Ursprüngliches Skript von Pablo Graubner)
SSL	Überprüft ob Fehler bei der Verwendung von *SSL/TLS* gemacht werden (Ursprüngliches Skript von Lars Baumgärtner)
GVMAnalysisExample	Erstellt einen Graphen, der die Methodenaufrufe visualisiert abgelegt
CodePermissions	Überprüft an welchen Code-Stellen Berechtigungen benutzt werden und dekompiliert diese

Eidesstattliche Erklärung

Ich versichere hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, ganz oder in Teilen noch nicht als Prüfungsleistung vorgelegt und keine anderen als die angegebenen Hilfsmittel benutzt habe. Sämtliche Stellen der Arbeit, die benutzten Werken im Wortlaut oder dem Sinn nach entnommen sind, habe ich durch Quellenangaben kenntlich gemacht. Dies gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen sowie für Quellen aus dem Internet. Bei Zuwiderhandlung gilt die Bachelorarbeit als nicht bestanden. Ich bin mir bewusst, dass es sich bei Plagiarismus um schweres akademisches Fehlverhalten handelt, das im Wiederholungsfall weiter sanktioniert werden kann.

Nils Tobias Schmidt

Marburg, 14. August 2014