Master's thesis

Predictive Identification of Android Malware through Hybrid Analysis

created by
Johannes Thon
Student number: 325369

Date of submission: 08.06.2018

Surveyors: Prof. Dr. Dr. h.c. Sahin Albayrak, DAI Laboratory,
Prof. Dr. habil. Odej Kao, Complex and Distributed IT Systems

Advisor: M. Sc. Minh Hoang Nguyen, DAI Laboratory

Technische Universität Berlin, Fakultät IV Elektrotechnik und Informatik

Statutory Declaration

I hereby declare that the thesis submitted is my own, unaided work, completed without any unpermitted external help. Only the sources and resources listed were used.

The independent and unaided completion of the thesis is affirmed by affidavit:

Berlin, June 6, 2018

_____

Signature

Acknowledgement

I would like to express my gratitude to Nisi, my family and friends who supported me throughout my years of study and throughout writing this thesis.

I would also like to acknowledge Minh Hoang Nguyen for his support, remarks and comments. Furthermore I like to thank the DAI Laboratory for the equipment and support.

Many thanks as well to Anis and Tobias for their valuable input.

Without your help I would not have been able to write this thesis.

Abstract

With a still increasing amount of mobile devices running the mobile operating system *Android*, their extensive usage and various application possibilities, those devices have become valuable targets for malicious apps.
With advanced static and dynamic analyses researchers gain valuable insights into the mechanics of malware, where machine learning is often utilized to detect unknown malicious apps. The Android operating system and related malware are continuously evolving. So training a machine learning model using outdated malware could negatively affect the performance of the predictive identification of more recent malware. Though several scientific publications used outdated malware collections.
This thesis focuses on the central research question: *How precise is the predictive identification of recent Android malware by comparing two hybrid approaches?* In this thesis recent malicious and benign Android apps from suitable repositories were collected. Two hybrid analyses were implemented to extract static and dynamic information from Android apps. Both approaches attempt to increase the code coverage of the dynamic analysis, which was executed on physical devices.
A *random forest* ensemble approach was utilized to perform a binary classification on malicious and benign Android apps using a total of 163559 features of the hybrid analyses. An antivirus aggregator service was applied to reduce the teacher noise. To provide a categorization of the analyzed malware an unsupervised clustering approach with *k-means* was utilized.
The classification of both hybrid approaches and the clustering were applied to over 9000 Android apps. The prediction of unknown malware results in an accuracy of approximately 90%. The clustering approach reveals 30 clusters with different malware labels.

Kurzfassung

Die Verbreitung und Einsatzmöglichkeiten mobiler Endgeräte, die das mobile Betriebssystem Android nutzen, nimmt stetig zu. Die Geräte erweisen sich als wertvolle Ziele für Malware, die in Form von Apps auf das Gerät gelangen.
Wissenschaftler haben ausgereifte statische und dynamische Analyseverfahren entwickelt, um Einblicke in die Funktionsweise von Malware-Apps zu erhalten. Oftmals werden maschinelle Lernverfahren verwendet, um unbekannte bösartige Apps zu erkennen. Software wie das Android-Betriebssystem oder Android-Malware entwickeln sich kontinuierlich weiter. Das Training eines maschinellen Lernmodells mit veralteten Malwaredaten, kann zu negativen Auswirkungen auf die Leistungsfähigkeit der Vorhersage von aktueller Malware führen. Mehrere Publikationen nutzen veraltete Malware-Apps als Grundlage.
Diese Masterarbeit behandelt die folgende zentrale Forschungsfrage: *Wie genau ist die Erkennung von unbekannter aktueller Malware im Vergleich von zwei hybriden Analyseverfahren?* In dieser Arbeit wurden aktuelle bösartige sowie harmlose Android-Apps von geeigneten Quellen erfasst. Es wurden zwei hybride Analysen implementiert, die in der Lage sind statische und dynamische Informationen aus Android-Apps zu extrahieren. Beide hybride Ansätze verfolgen das Ziel die Programmcodeerfassung zu maximieren. Die dynamische Analyse wurde auf physikalischen Geräten durchgeführt. Ein *random forest*-Ansatz wurde verwendet, um eine binäre Klassifikation von bösartigen und normalen Apps durchzuführen. Dazu wurden insgesamt 163559 Features der hybriden Analysen verwendet. Ein Antivirus-Aggregator-Dienst wurde genutzt, um Störungen der Daten für das Modelltraining zu minimieren. Um eine Kategorisierung der verwendeten Malware zu erhalten wurde ein Clustering-Ansatz mit Hilfe von *k-means* umgesetzt. Für die Klassifizierung und das Clustering wurden über 9000 Apps verwendet.
Unbekannte Malware wurde mit einer Genauigkeit von etwa 90% erkannt. Der Clustering-Ansatz deckt 30 Cluster mit verschiedenen Malware-Bezeichnungen auf.

Contents

1 Introduction

This master's thesis is originated at *Technische Universität Berlin* and is supervised by M. Sc. Minh Hoang Nguyen at DAI-Laboratory in the competence center *Security (CC SEC)*. It is focused on the central research question: *How precise is the predictive identification of recent Android malware by comparing two different hybrid analysis approaches?*
The following section 1.1 will make the reader familiar with the motivation of this thesis, followed by a complete structure overview in section 1.2.

1.1 Motivation

With decreasing overall trading prices the amount of mobile devices worldwide is still increasing. Particularly the portability of mobile devices makes them attractive to users [FASW15]. Nowadays, Android is by far the most popular mobile operating system and is dominating the market. According to the *International Data Cooperation (IDC)* it had an overall market share of approximately 85 % worldwide at the beginning of 2017 [Int18].
Besides running mobile devices with Android, the operating system gains importance within the automotive division as Android Auto [1] and in the Internet of Things (IoT) market as Android things [2].
Since the beginning, Android was always a target of attacks because users store valuable information about their personal identity, user credentials and payment information on their devices [PÇA16]. Furthermore the amount of mobile apps increases. Especially malicious apps are in the focus of security researchers all over the globe. Many publications regarding mobile malware detection address the Android platform.

[1] https://www.android.com/auto/ (Last access on 01.05.2018)
[2] https://developer.android.com/things/index.html (Last access on 01.05.2018)

In the last years tools with different analysis techniques for Android apps were developed. For example the DAI laboratory provides the static analysis tool *Androlyzer* [3]. As malware is continuously evolving, it is important to permanently monitor its activities. Within this field a lot of research papers were published. Various machine learning techniques were used to recognize new malware apps. Often approaches result in very good detection accuracy. Nevertheless many scientific publications use outdated malware data from years 2010 to 2012. For realistic prediction it is mandatory to take also newer malware into account. This master's thesis is focused on hybrid analysis and detection of recent malware of 2017 for the Android mobile platform using machine learning techniques. Additionally this work intends to enrich the capabilities of Androlyzer and the DAI laboratory to improve analysis capabilities against malicious mobile applications.

1.2 Overview

In order to get familiar with the structure of this thesis, this section will outline its content. Chapter 2 introduces the required background information and focuses on knowledge as well as relevant tools. Chapter 3 describes the given problem and requirements. Chapter 4 contains the concept of this thesis as well as its design decisions. The main realization can be found in the chapters 5, 6 and 7. Afterwards an evaluation of the results is outlined in chapter 8. This thesis ends with the conclusion and outlook in chapter 9.

[3] https://androlyzer.com (Last access on 15.05.2018)

2 Background

This chapter contains background information for this master's thesis. The required knowledge will be outlined in the following sections. The first section 2.1 imparts fundamental knowledge about the Android operating system and Android applications. Because this work deals with Android applications and the Android operating system, it is recommended to understand the basics. In order to reveal how to extract information from Android apps, the thesis discusses the fundamentals of Android app analysis fundamentals. It also explains the term *hybrid analysis*.
This thesis implements specific approaches to realize predicting behavior. Therefor essential knowledge about machine learning techniques is required, which is the subject of section 2.2.
Section 2.3 introduces important tools that are used in this thesis. They are primarily utilized for data storage, analysis of Android applications and machine learning.
Section 2.4 deals with the used online services, which contains an online file analysis service and a general view of Android app collections. The final section 2.5 outlines an overview of related scientific publications.

2.1 Android

In 2005 Google acquired the startup *Android Inc.* based in Palo Alto, California, which at this time was developing an early version of the mobile operating system named Android [Elg05]. In the last decade the Android operating system was developed by Google Inc. and a group of companies known as the *Open Handset Alliance (OHA)* [And17b].
Nowadays, a lot of other corporations have invested in Android [And17b]. The system is continuously developed and released under the *Android Open Source Project (AOSP)* [1] using open source licenses. The majority of Android software is

[1] https://source.android.com/ (Last access on 01.05.2018)

licensed under the Apache Software License version 2.0 with exceptions like the Linux kernel, which uses the *GNU General Public License (GPL)* [And17a]. In most cases *Google Mobile Services (GMS)* - a collection of Google applications - is added to the system. This collection uses a different licensing approach called *Google licensing model* and contains apps like Google Maps, Youtube, Play Store and Google Play Services [2]. As the Android operating system evolves different versions like Android 4.4 or Android 8.0 exist. In general, new versions introduce new features, changes in the API and security patches. Android's major API versions are named by a candy-word, whereas developers more often use the precise API-level numbers. They are more precise because sometimes a name like *Lollipop* stands for Android API level 22 and also 23. The current Android version 8.1 has an API level of 27 and is named *Oreo*. The distribution of platform versions of the Android ecosystem is significant for the development of apps. The distribution from December 2017 is illustrated in figure 2.1.

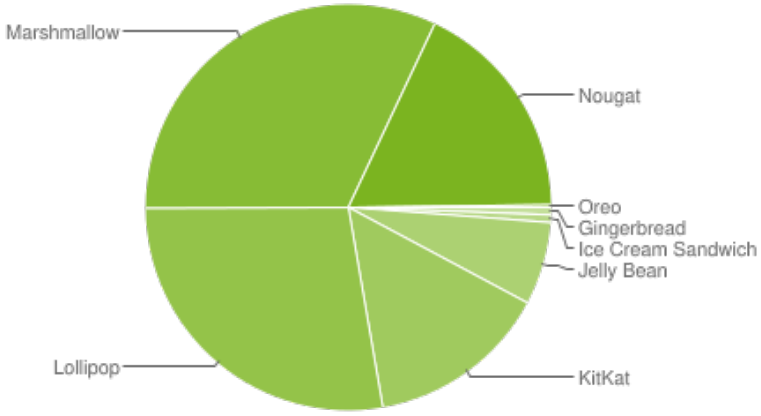

Figure 2.1: Platform version distribution of Android [Goo18d]

As shown the older Android versions *Gingerbread* (API level 9 and 10) and *Ice Cream Sandwich* (API level 14 and 15) represents a tiny fraction of the ecosystem. The same applies to the newest Android release *Oreo*. If an Android app is distributed for *Lollipop*, *Marshmallow* (API level 23) and *Nougat* (API level 24 and 25), it is compatible to nearly 75% of the devices.

[2] https://www.android.com/gms/ (Last access on 07.05.2018)

2.1.1 Android Operating System

In general Android is a software stack, which is illustrated in figure 2.2. At the core it contains a Linux kernel in a modified version. This kernel is known for well-designed security features and enables device manufacturers to develop hardware drivers for a well-known kernel [And18b]. The overlying *Hardware Abstraction Layer (HAL)* mainly contains APIs for upper layers in order to use hardware in a simple and unified way. The upcoming layer is composed of two modules. The left one contains native libraries like OpenGL, Webkit or SSL/TLS, which provide important features for applications. The right one contains *Android Runtime (ART)*, a modified *Java Virtual Machine (JVM)*, in order to run user applications not implemented in native code. ART executes *Dalvik Executable Format (Dex)* and uses the Dex bytecode specification [And18d]. The predecessor is called Dalvik. Apps constructed for Dalvik will also run on ART. Eventually some techniques will not be available. ART comes with *Ahead-of-time compiling (AOT)*, which will perform complete bytecode translation after installing but before running the application. ART also provides improved garbage collection and new debugging features [And18d]. The next layer *Java API Framework* contains the entire feature set of the Android operating system accessible through APIs. These APIs are basic components for building Android applications. One example is the *View System* to build the app's *user interface (UI)*. Another one is the *activity manager*, which manages the life cycle of apps. A *content provider* allows access to data from other apps. At top-level system apps are located in order to provide basic functionality like calendar, contacts and e-mail [And18b].
Each of the described components relies on the security of their underlying component. Almost every system component located above the kernel is restricted by an application sandbox technology. Only a small amount of the Android OS is running with root privileges by default [And18e].

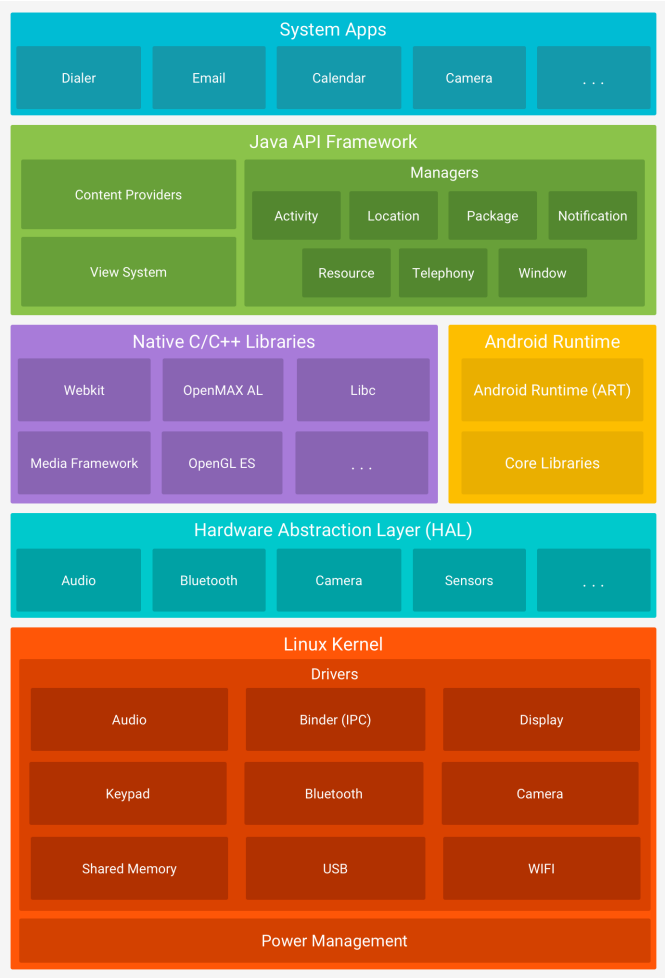

Figure 2.2: The Android stack [And18b]

Sandbox technology means, that for every application Android assigns a *unique user ID (UID)* and runs it within an own process. With this approach memory and file system isolation can be enforced. This stands in contrast to Linux, which runs several processes under the same user. The kernel application sandbox prevents uncontrolled mutual interaction between applications. Access to other apps or resources will be made available solely via *Inter-process communication (IPC)* through the kernel. There is no distinction between native code and byte code applications. All user apps are restricted by an enforced Sandbox. Without access to specific features these apps would have a very restricted functionality. Therefor a permission system is established, which grants apps access to specific features. The only way to bypass this system is to compromise the Linux Kernel by itself [And18f]. The Android operating system runs on various devices or emulators, where the target architecture for the OS is emulated on the host system.

The Android operating system is booted by using a bootloader. This can vary on devices manufactured by different vendors. The bootloader is capable of booting into a recovery image, which will provide a debug mode for the device. Vendors often encrypt and lock their bootloaders to prevent modifications of the booted Android operating system. Unlocking a bootloader will enable altering or replacing the operating system or recovery image. One way of rooting a device will use an unlocked bootloader to install custom recovery images, which are able to modify the system and installing the `su` command. Rooting in general is used to gain superuser privileges on a system, which is usually prevented by default. This approach will allow the user to change the system. Popular modifications are bloatware removal or user interface improvements. Rooting comes with great responsibility and can decrease the device security. If malware gains root access, it can become quite powerful, hard to detect or removed.

2.1.2 Android Apps

The market share and customization abilities of the Android ecosystem are mainly the result of the ability to install applications of 3rd-party providers. A developer or interested person is able to create an Android application and publish it via the Google Play Store. Due to the openness of Android it is possible to install apps via alternative market places or through a direct connected device. To transfer and install mobile applications on an Android device the *APK (Android Package)* file format is used. This is basically a Zip file containing several different files and folders. Android applications can be written using Java, Kotlin or C++. Kotlin runs on the JVM, is developed by the JetBrains developers and licensed under Apache 2.0 [Goo18b]. C++ can be integrated as native program code.
Every application runs in an own virtual machine. It is enforced to run in a sandbox in order to ensure secure isolation [Goo18b]. In order to enable specific functionality of the app a permission system is used. Basically every app has to declare a permission list to access specific features like Bluetooth, GPS or the contact list. Before Android 6.0 (API level 26) permissions need to be granted by the user before installing. In order to install an app, the user had to grant all permissions. Only a hidden feature was able to allow and reject individual permissions in Android 4.3. The so called *App Ops* feature was later removed in Android 4.4. Since API level 23 the permission system was changed by introducing *Runtime permissions*. Permissions are split into two groups: normal and dangerous permissions. Normal

permissions are automatically granted by the system, whereas dangerous permissions must be approved by the user explicitly [And18c]. Instead of requesting upfront, this dialog happens during runtime. Developers have to deal with denying the request. The following figure 2.3 contains the four core components of an Android app.

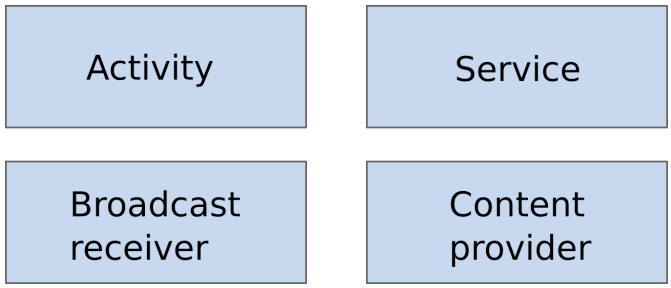

Figure 2.3: Android core components

Every component fulfills a specific purpose and comes with an own life cycle. An *activity* interacts with the user and represents a single screen with a user interface [Goo18b]. To use an activity developers can subclass the activity class and implement app-specific program code in the appropriate life cycle methods. Activities are used as a main entry point for applications.
A *service* is a component to process a specific task in the background. It can run for a long time and does not require a user interface. To process data or wait for network data, developers are encouraged to start a service component to prevent the UI thread from blocking. A blocked UI thread can lead to a noticeably negative user experience. Furthermore it is possible to run services, even if the app is not visible to the user. Since Android API level 26 the `JobScheduler` allows a more intelligent way to save battery resources, by using Doze and Android Standby. A *broadcast receiver* is a component which is responsible for transmitting messages between Android applications. The operating system uses broadcast receiver to signal status messages like screen status. *Content providers* are used to manage data in Android apps. The component allows an app to read or write data like contact information through an API. In contrast to other programming languages, Android has no single main entry point like the main-method. Communication between components or apps is established through asynchronous messaging called *intents*. This enables components to communicate with each other or interact with other applications in the system. Since every application is running in a sandbox, using intents is Inter-process communication. This allows to start the camera application in a messenger

application for example [Goo18b].
The common Android archive format APK contains the components depicted in figure 2.4.

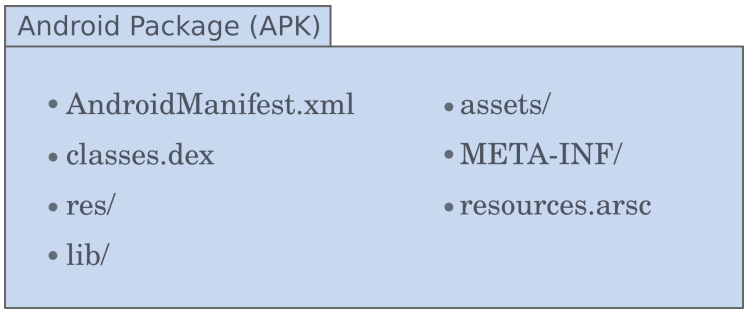

Figure 2.4: Contents of an *Android Package (APK)*

The `AndroidManifest.xml` contains important information about the application. It is formatted in XML and contains basic information like app name, app version or target API level of the application. It also contains the minimum API which is suitable for the app. If the host Android system is lower, the application will not run on the device. The manifest also contains the permissions required by the app. An entry in the manifest is required for activities, services, broadcast receivers and content providers. However, a broadcast receiver can also be declared in a dynamic way [Goo18b].
The `classes.dex` file contains the bytecode of the Android application. The general design of the bytecode is described on the website of the Android source code [3]. The Dex file has a reference limit of 65,536 methods. If an app exceeds this limitation the multidex feature can be used, which creates additional numbered Dex files [4].
The Android UI is defined by special resource files in the XML format. They are separated from the program code. User interface elements for an activity, for example text fields or buttons, are defined within the XMLs. All resources are located in the folder `res`. The interconnection between UI elements in the program code and the definition file is realized by special ids and the Android R class. Details regarding the R class are stored in the `resources.arsc` file. The `assets` folder contains additional files, which can be accessed through the application. If native code is used in the application, the compiled files are stored in the `lib` folder. The folder `META-INF`

[3] https://source.android.com/devices/tech/dalvik/dalvik-bytecode (Last access 15.05.2018)
[4] https://developer.android.com/studio/build/multidex (Last access on 15.05.2018)

contains information like hash sums for the files in the APK and the appropriate digital signature.

2.1.3 Analysis of Android Apps

This subsection deals with the basics of analyzing Android applications and provides insight into libraries and tools.
In general, program code analysis of applications can be divided into static and dynamic analysis. It describes a research field which tries to extract information about the behavior of an application. Usually static analysis involves an automated tool that takes the source code of a program as input and examines the code without executing it. Then it returns results after checking the code structure [LBP+17]. This often involves sequences of statements and how variables are processed throughout different API calls. The main advantage of static analysis is that it often covers the entire code, while requiring less resources [LBP+17]. Disadvantages are the lacking detection of dynamic code loading and issues with program code obfuscation.
The dynamic analysis runs the application in a prepared environment to examine its behavior during runtime. It is often costly to implement and expensive on resources during run, but API calls, network traffic and other measurements can be gathered. Dynamic analysis lacks code coverage, because only code is executed, that meets specific conditions in the analysis environment [LBP+17].
Several libraries and complete toolkits exists to analyze Android apps.
For the static analysis the APK can be disassembled with tools like *Apktool* [5]. It helps to disassemble the `classes.dex` to human readable syntax and resources like XML-files or assets nearly to their original format. An assembler/disassembler library for the Dex format is *SmaliBaksmali* [6]. It also contains `DexLib2`, a component which provides a Java presentation of the Dex bytecode. It has its own syntax and fully supports Dex functionality. It is written and maintained by Ben Gruver. These tools help to statically analyze Android applications, but do not provide a complete static analysis. Another tool is *APKParser* [7], which provides easy access to APK details like the manifest. It is able to decode the binary translated XML files into readable strings.
A mature static analysis is performed by FlowDroid, a research project of *EC*

[5] https://ibotpeaches.github.io/Apktool/ (Last access on 04.05.2018)
[6] https://github.com/JesusFreke/smali (Last access on 04.05.2018)
[7] https://github.com/hsiafan/apk-parser (Last access on 04.05.2018)

SPRIDE et. al [ARF+14]. This analysis is context-, flow-, field- and object sensitive. It models life cycles of Android components to gain precision. It is able to model static callbacks defined within resource XMLs. FlowDroid is released as an open Source project for PC usage. *Androlyzer* [8] is an Android app, which runs a comprehensive static analysis on the device. It is able to find privacy leaks within an application by modeling the program code using control- and data flow graphs. It uses an entry point and life cycle sensitive approach. Large applications can be uploaded on the Androlyzer server. A complete overview about static analysis tools and scientific work can be found in the publication *Static Analysis of Android Apps: A Systematic Literature Review* [LBP+17].

To perform a dynamic analysis of Android applications a suitable environment is necessary. This is realized by using an emulator or a real device. Because Android applications are compiled into an intermediate bytecode, they are able to run on various processor architectures, if a suitable virtual machine like ART or Dalvik is available. If an Android application contains native program code, it needs specific compilation for one or more target architectures, so called *Application binary interface (ABI)*. Due to size reasons APKs are often compiled only for ARM architectures. This will prevent an installation on a x86 emulator, which is often used on commodity machines due to performance reasons. The usage of an ARM emulator on a x86 machine results in slow emulation of the ARM target architecture. Performing a dynamic analysis on an x86 emulator is therefor not always applicable, due to mismatching ABIs of Android apps. Using an ARM emulator on an x86 machine therefor results in poorer performance. Another disadvantage using an emulator is that apps can determine, if they are currently running in an emulator. These methods are often used by malware, to circumvent the detection through a Google analysis prior to being available in the Google Play repository.
An example for a dynamic analysis tool is the *Mobile Security Framework (MobSF)* [9]. This framework provides security analysis for Android, iOS and Windows applications. It is released under GPL3. For Android, MobSF supports static and dynamic analysis. It uses Python 2.7 and Java 1.7 and provides a web interface and a simple API. The main functionality is provided by the web user interface, it is easy to choose a file and start an analysis. The results for static analysis are comprehensive. The dynamic analysis will be executed on a VirtualBox *virtual machine (VM)* or

[8] https://androlyzer.com/ (Last access on 03.05.2018)
[9] https://github.com/MobSF/Mobile-Security-Framework-MobSF (Last access on 03.05.2018)

a connected device. Static analysis can be performed using the API, however a dynamic analysis is not controllable through the API.

CuckooDroid is an open source software to perform automatic analysis of Android applications. It is developed by Check Point Software Technologies and is an extension module for the *Cuckoo Sandbox* tool [10]. It is designed to run untrusted Android applications within an isolated environment, called a Sandbox. A virtual machine provides the Android Operating System environment. The file is executed within the VM and monitored in real-time [Che18]. The system uses an Android emulator with ARM architecture. Modifications to the system image provides dynamic analysis functionality. Realization is done by using the XPosed Framework and Droidmon, which are described in the later section 2.3.4.
In order to study the behavior of an application, the key is a good code coverage. This can be reached by running the application and performing several UI interactions. In this way the application can be forced to run most of implemented code. Android UI interactions can be performed by using test tools like `MonkeyRunner`. The *UI/Application Exerciser Monkey* [11]. is able to perform random user interface events on applications. In order to analyze the UI of an Android application the command line tool `dumpsys` [12] can be utilized. It provides various diagnostic information like memory allocation details, network diagnostics and is able to provide details about running applications and user interface elements. The GitHub project *AndroidViewClient* [13] by Diego Torres Milano used the *Android Debug Bridge* (described in chapter 2.3.3) and `dumpsys` to provide convenient access to the view hierarchy of apps with a python script named *dump*. It is released under Apache License 2.0.

2.2 Machine Learning

This section deals with basic machine learning knowledge. Therefor two general approaches are described in subsection 2.2.1 and 2.2.2. Fundamental information about feature engineering is described in section 2.2.3, followed by details regarding

[10]https://cuckoosandbox.org/ (Last access on 03.05.2018)
[11]https://developer.android.com/studio/test/monkey.html (Last access on 04.05.2018)
[12]https://developer.android.com/studio/command-line/dumpsys (Last access on 02.05.2018)
[13]https://github.com/dtmilano/AndroidViewClient (Last access on 02.05.2018)

the improvement of a machine learning model and its evaluation in chapter 2.2.4. A detailed machine learning approach is described in the last section 2.2.5 of this chapter. The upcoming lines will motivate and introduce the topic.

Within the last decades the production of data has increased, due to ubiquitous cloud computing, mobile- and IoT devices. Therefor the last years one term showed up more often: Big data. To solve various tasks often a set of algorithms exist, that can be used. Not every task can be effectively solved by an algorithm. One example is the prediction of customer behavior. In this case collected data can be useful. A machine could automatically extract the algorithm for a task. Maybe this is not perfect realizable, but perhaps it is possible to construct a useful approximation, recognize patterns and extract knowledge from the data. This helps to understand the process and to make predictions, which can be right for the future. This approach is called machine learning. This topic is part of the bigger picture of artificial intelligence [Alp14].
Today machine learning models are used for recommender systems for example in online shops, for fraud detection in credit card companies or for medical diagnosis in hospitals. Machine learning, especially the field deep learning, is used for speech recognition and face recognition. As today many scientific communities use machine learning for their research and scientific progress. The machine learning algorithms can be divided into two main groups, which will be outlined in the next subsections.

2.2.1 Supervised Learning

The *supervised learning* approach is able to automate a decision process from generalization of known examples and specific input data. It can be used to predict a specific result [MG17]. In general the data is labeled and split into training and testing data. The training data is fed into a supervised machine learning algorithm to train the model. Subsequently the test data is used to verify the effectiveness of the model by comparing the predicted label with the test label of the data. Supervised learning can furthermore be divided into classification and regression algorithms. Classification is used to predict a specific class out of a given choice. An example is spam prediction, where two classes are defined: *spam* and *not spam*. Regression in contrast is used to predict a continuous value like the income of a person [MG17].
To introduce common terms in machine learning a simple data approach will be

shown. The example is taken from the book of *Ethem Alpaydin* named *Introduction to Machine Learning - Third edition* [Alp14]. The upcoming task should classify family cars out of a set of cars. Therefor a training set with cars and the appropriate class membership is provided. Family cars are labeled as a positive example, whereas other cars are labeled as negative examples. Provided attributes for determining the class are *price* and *engine power*. These input variables or features are defined as the *input representation*. Other features may be useful for this classification, but are ignored due to complexity [Alp14].
A car is represented by a vector x and its label r as shown in formula 2.1 and 2.2. Vector x contains the price as x_1 and the engine power as x_2. The label r is set to 1 in case of a positive example, otherwise it is set to 0.

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{2.1}$$

$$r = \begin{cases} 1 & \text{if } x \text{ is a positive example} \\ 0 & \text{if } x \text{ is a negative example} \end{cases} \tag{2.2}$$

The total training set is named X and shown in formula 2.3. Every car has an index t with the appropriate vector x^t and label r^t. The training set contains N examples in total and starts with index $t = 1$.

$$X = \{x^t, r^t\}_{t=1}^{N} \tag{2.3}$$

The data of a sample training set for the cars and their appropriate label is depicted in figure 2.5. The feature *price (x_1)* can be found at the x-axis, whereas feature *engine power (x_2)* is represented by the y-axis. Every car is depicted as a circle, which is filled with a + if the car is labeled as family car and − if not.

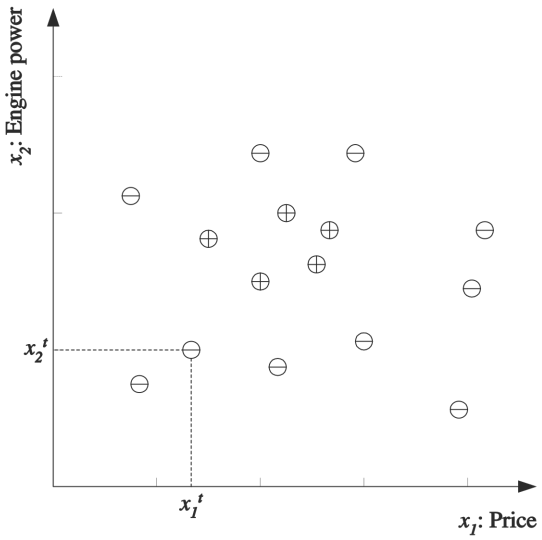

Figure 2.5: Training set with labels [Alp14]

After discussions with experts and further analysis of the data, it is assumed that the price and engine power of a family car will be in the range shown in formula 2.4.

$$(p_1 \leq price \leq p_2)\ AND\ (e_1 \leq enginepower \leq e_2)\ \text{[Alp14]} \tag{2.4}$$

The class of family cars C can be represented as a rectangle in the two-dimensional space, as shown in figure 2.6.

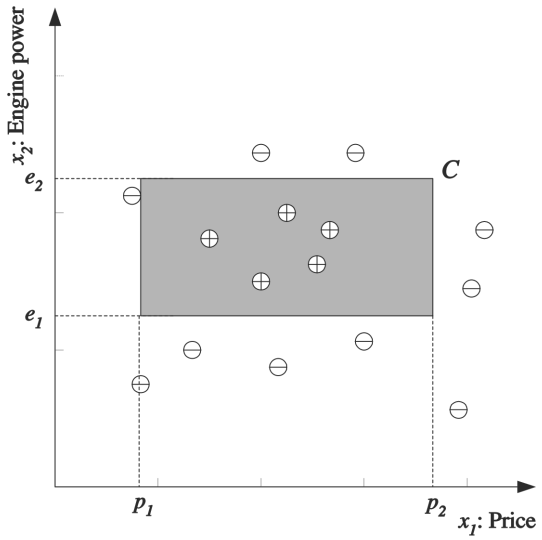

Figure 2.6: Highlighted class of family cars C in the training set [Alp14]

C is selected from a set of rectangles H, the so called *hypothesis class*. Within this class the machine learning algorithm will find a particular *hypothesis*, $h \in H$, to approximate C as closely as possible [Alp14]. How well this hypothesis h will classify unknown examples correctly is called *generalization*. A good generalization is required to get best prediction results. As depicted in figure 2.7, the most specific hypothesis S and the most general hypothesis G are valid hypothesises for the given training set. The class of family cars C lay in between.

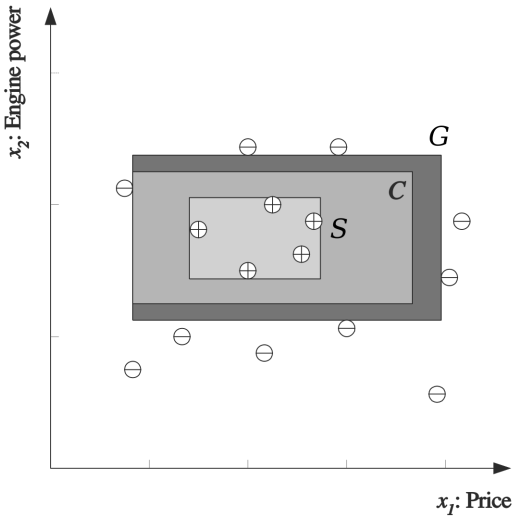

Figure 2.7: Most specific and most general hypothesis [Alp14]

If S is chosen as hypothesis for family cars, the amount of false negatives is high. If G is chosen, the amount of false positives is high. For best generalization the complexity of the hypothesis class H should match with the complexity of the function deduced from the data [Alp14]. A less complex choice of H is called *underfitting*, whereas a too complex H is named *overfitting*. This can happen for example if the model is to strong adjusted to the training set or the training set is noisy, as depicted in figure 2.8. Optimal is a reduction of the teacher noise before the labeling takes place.

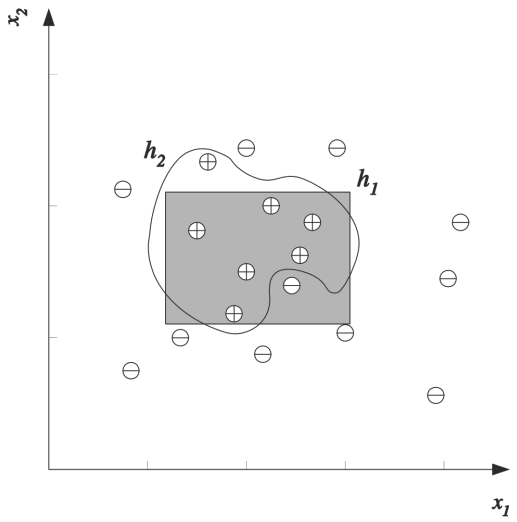

Figure 2.8: Example for noisy training data affecting the model complexity [Alp14]

In summary there is a trade off between three factors, the *triple trade off* [Alp14]:

- capacity of the hypothesis class
- amount of training data
- generalization error on new data

As the size of the training set increases the generalization error decreases. If the model complexity will increase, the accuracy increases as well and will decrease after the sweet spot is hit [Alp14].

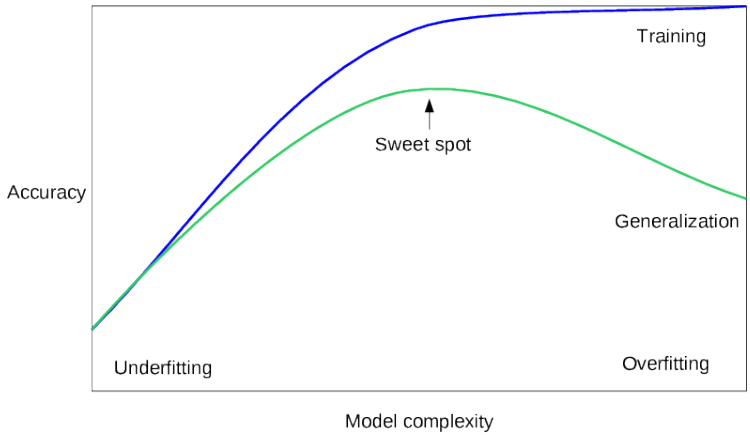

Figure 2.9: Relationship between model complexity and accuracy [MG17]

If the model complexity is too high, it will concentrate too much on the data points of the training set. This results in high accuracy for the training set, but a low generalization ability [MG17]. This relationship is depicted in figure 2.9. A concrete classification algorithm can be found in section 2.2.5.

2.2.2 Unsupervised Learning

This section deals with the second group of machine learning approaches called *unsupervised learning*. First basic information is provided followed by presenting a specific algorithm.
Unlike supervised learning the results of the output data are unknown. There is no teacher for training the algorithm and no labels that are assigned. The machine learning algorithm receives only input data and the task to extract knowledge out of the data [MG17]. Imagine a collection of blog entries and the need to retrieve the topics. For this problem the text can be fed into an unsupervised machine learning algorithm, which will identify the topics. In this case one can limit the amount of topics, but it is not possible to predict the allocations of the individual blog entries [MG17].
Another use case of these algorithms is the unsupervised transformation on a data set. Therefor the representation of the data is changed in order to improve understandability for machines or humans. Popular approaches are dimensionality reduction or cluster algorithms [MG17].
They have the ability to cluster data into previously unknown groups. It is difficult to evaluate unsupervised learning algorithms, because the correct output is unknown. Therefor often manual evaluation is applied. Unsupervised learning is frequently used in the exploring phase or for preprocessing of supervised algorithms. Thus scaling the train and test data can be useful to improve the results of the algorithms [MG17].
A simple and popular cluster algorithm is *k-means*. Let $X = \{x_i\}, i = 1, \ldots, n$ with n data points, which should be clustered into a set of K clusters. The overall set of clusters is defined as $C = \{c_k, k = 1, \ldots, K\}$, whereas c_k stands for one cluster with index k. The algorithm will find a partition, such that the squared error between the empirical mean m_k of the cluster c_k and the points of the cluster is minimized [Jai10]. The squared error between m_k and the points in the cluster c_k is defined as

$J(C_k)$ as shown in formula 2.5.

$$J(C_k) = \sum_{x_i \in C_k} \|x_i - m_k\|^2 \text{ [Jai10]} \tag{2.5}$$

$$J(C) = \sum_{k=1}^{K} J(C_k) \text{ [Jai10]} \tag{2.6}$$

The overall goal is to minimize the sum of the squared errors of all K clusters defined as $J(C)$ (see formula 2.6). The algorithm takes the amount of clusters, the initialization method and distance metrics.

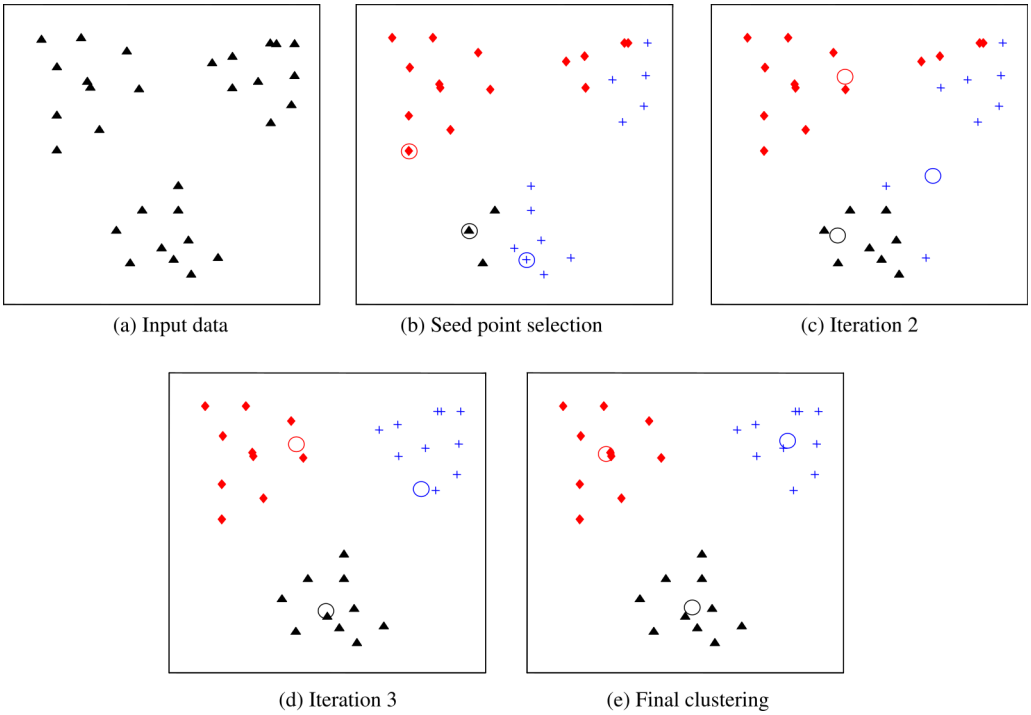

Figure 2.10: An example of *k-means* [Jai10]

An example procedure is depicted in figure 2.10. Random initialization on three clusters is assumed. The sample input data is depicted in a 2-dimensional space (a). The algorithm will first add three initialization points at random, uses a distance measurement (e.g. euclidean distance) and assigns the closest data points to its cluster c_k (b). Afterwards the centroids are moved to the mean position of every assigned cluster c_k. The distance measurement and the assignment of the closest data points are repeated, until the minimum sum of squared errors $J(C)$ is reached.

2.2.3 Feature Engineering

In order to use machine learning, access to an application-specific data collection is mandatory. Every record in the collection contains information in the form of data, generally in a specific format. It is important to understand the data and, for effective usage of machine learning, to represent data in the correct way, so it fits best for the application. This is done by the so called *feature engineering*.
A feature or an attribute of a data point is some specific information. In the example of cars this could be the car model, engine power, age, build year and so on. Because machine learning algorithms require specific input, a conversion of this information is necessary. This can be done by transforming this information into feature vectors. Typically the main features types are: continuous and categorical features. The family car classification in section 2.2.1 uses two continuous features: the price and the engine power. These features can grow theoretically infinite. Whereas categorical features for a car could be color, fuel type or transmission type. A continuous feature value is able to grow and can be represented for example by an integer or float variable. Categorical features could also be represented by numbers. Imagine the seats in a car. They cannot be increased boundlessly, so a categorical features is the desired representation. The question is how to represent categorical data in an appropriate way.
A popular way is *one-hot-encoding*, where a categorical variable is replaced by features for their containing categories. They contain a numeric one for the chosen category, otherwise a zero. The transmission type for a car could have for example three different categories: manual, classic automatic and dual clutch automatic. For example in this case this one variable can be transformed into three features called: *transmission_manual*, *transmission_classic_automatic* and *transmission_dualclutch_automatic*. For the machine learning algorithm a car has these features and one of them is set to one (true) and the others to zero (false).
Increasing the amount of features will also increase the complexity of the model and overfitting is more likely to happen. So it would be really useful to know, which are the most relevant features for the desired scenario. This can help to reduce the model size to a more general one and eventually prevent overfitting. Three methods to perform this selection automatically are the following [MG17]:

- univariate statistics
- model-based selection

- iterative selection

The first simple selection technique uses statistics between a single feature and the target set to evaluate the statistical coherence. Features with the strongest confidence are elected after iterating. It should be emphasized, that no relation between multiple features is considered [MG17].
A model-based selection approach uses a supervised learning algorithm to separate important features from less important ones. This technique is more sophisticated than univariate statistics, because it can handle coherence between multiple features. Iterative selection uses several models to reduce the feature set. One approach is to start with no features and add them step by step. Another could start with all features and reduce them in an iterative way. This can also be called a recursive elimination. This technique is really resource intensive, because of retraining the model for each iteration [MG17].

In order to use machine learning on text documents some feature engineering techniques and preprocessing is required as well. For instance to extract words from documents and cluster them regarding topics. One approach is to use the *Bag-of-words* approach, which is typically divided into the following three steps depicted in figure 2.11.

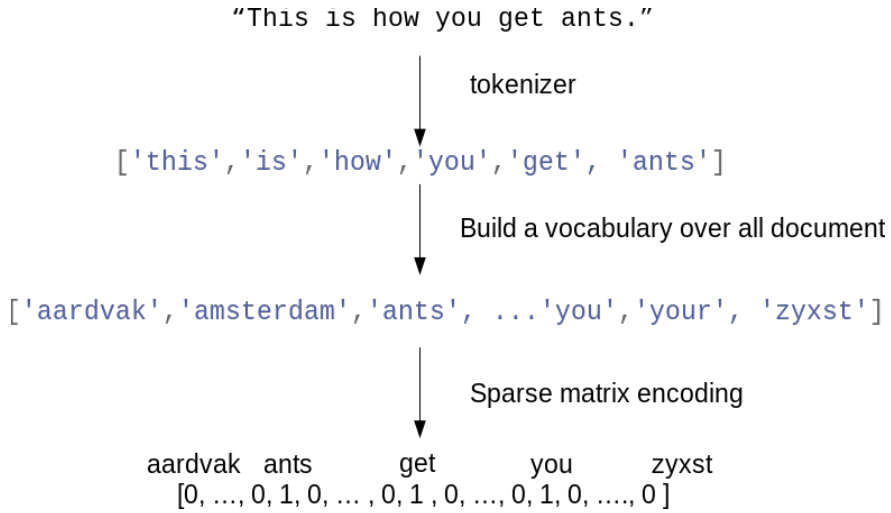

Figure 2.11: Bag-of-words for a sample data set [MG17]

First the text of the documents is split into tokens using a tokenizer. Afterwards a vocabulary is created from all occurring words in all documents. The result is

encoded into a sparse matrix. In the end every document is represented by a feature vector including the occurrences of the words in the vocabulary. Now one can assume that words with higher occurrences will describe documents better. This is probably not the case. Longer document will also have a higher average count of word occurrences compared to shorter documents [MG17]. A solution is the *TF-IDF (term frequency/inverse document frequency)* measure. First the relevance of a keyword in the document is determined by using the *term frequency (TF)*, which is stated in formula 2.7.

$$TF_{i,j} = \frac{f_{i,j}}{max_z f_{z,j}} \quad \text{[AT05]} \tag{2.7}$$

The frequency of keyword i in document j is denoted with $f_{i,j}$. This is divided with the maximum frequency of other keywords z in document j referred to as $max_z f_{z,j}$. In order to reduce the weights for keywords that appear more often, the *inverse document frequency (IDF)* shown in formula 2.8 is used.

$$IDF_i = log\frac{N}{n_i} \quad \text{[AT05]} \tag{2.8}$$

For every keyword IDF will divide the amount of documents N through the frequency of the keyword i in these documents denoted as n_i.

$$w_{i,j} = TF_{i,j} * IDF_i \quad \text{[AT05]} \tag{2.9}$$

Combining these two calculations results in the TF-IDF weight w for a keyword of a document as shown in formula 2.9 [AT05].

2.2.4 Evaluation and Model Improvement

This subsection deals with evaluating a machine learning approach and improving the used model by optimizing parameters.
All the collected feature values can be stored in a matrix X. The rows will contain the data points or observations, while the columns denote the used feature set. An additional vector y contains the labels for all observations in X. A simple way to evaluate a supervised model is splitting the data X, y into a training data set X_train, y_train and test data set X_test, y_test. The size of the training set is often set to 60%, whereas the test size is set to 40%. The training data set is

obviously used to train the model, whereby the testing data set is utilized for testing. Therefor only *X_test* is used to predict the labels. Afterwards *y_test* is utilized to compare the results and return the prediction accuracy for the test set. Beneficial would be shuffling the data beforehand and stratify the class distribution within training and test set. The test/train approach is simple, but provides only results for the specific test data set part. There are more superior approaches available.
Cross-validation for example will split the data set into several subsets, named folds. In a 3-fold cross-validation always one fold is used as test data set and the other two as training set. Such a standard cross-validation using three classes is depicted in figure 2.12. In summary there are three possible compositions and three results. They can be summarized for example by using the mean or median value. This approach is more robust and precise then a simple train-test split, because it takes more different training input into account as well as other different test data. It is possible that one class will be not present in the training folds, but appears in the test fold. In order to fix this a specific k-fold stratified variant of cross-validation can be used. It will preserve the ratio of the used classes within the training and test set. A 3-fold version using three classes is depicted in figure 2.12. Often five or ten folds are used in such an approach.

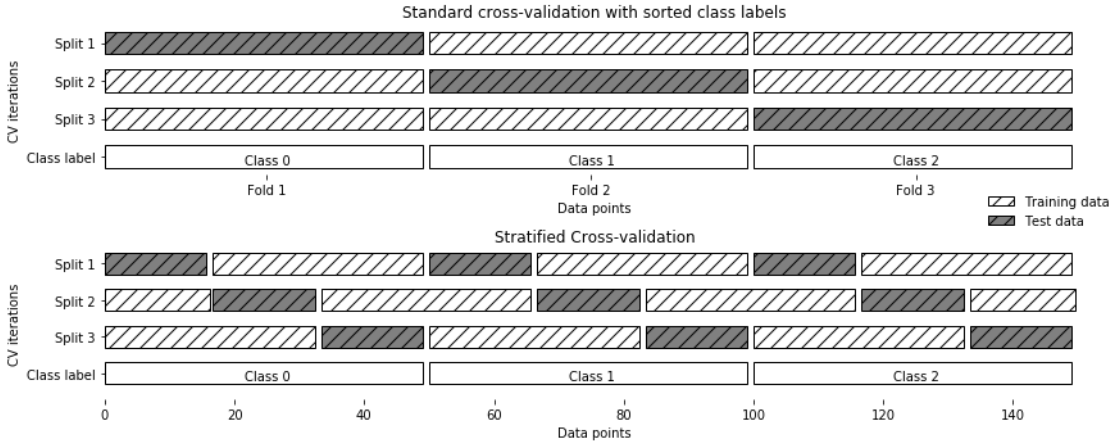

Figure 2.12: Scheme for a standard and stratified cross-validation with three classes [MG17]

In order to improve the model a possibility is the modification of the used parameters and observing the results. This parameter optimization can for example be realized by using *grid search*. Basically the first step is to choose the parameters, which should be used for optimization. Then a predefined range of values is used until the highest

score for a test data set is reached. In this case one big problem arises: overfitting. If the test data is used for setting up the model, it shouldn't be used for evaluating the model's quality. Otherwise knowledge about the test data leaks into the model. Thus an independent data set for the evaluation is needed [MG17]. A solution would be the usage of a threefold split into training, validation and test data as depicted in figure 2.13.

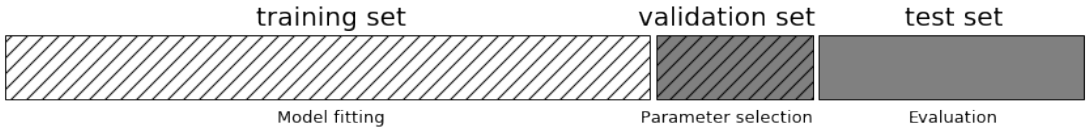

Figure 2.13: Scheme for train-validation-test split [MG17]

The validation set is used to optimize the model, whereas the test data is used as an independent test set.
Important to mention is the cost for performing a grid search. For every parameter combination the model needs to be built and tested. A lot of values result in high computational effort.
An optimal combination is using a nested cross-validation. In this case grid search is used with a cross-validation for parameter optimization together with k-fold stratified cross-validation to test the prediction results. This would result in even higher computational effort, but results in a more precise evaluation of the optimized model. This means that for every outer iteration training and test folds are constructed using cross-validation. Afterwards the data of the training fold is passed to the inner cross-validation. It applies a grid search with cross-validation on this data, in order to retrieve the optimized parameters. The outer iteration uses the best model on the test data set. This approach builds even more models, since the used parameter combinations are multiplied by the inner and the outer folds.
A classification with two classes is called a binary classification. In order to evaluate a classification, specific evaluation metrics are used. Therefor the previous example of family cars can be used. Imagine a binary classifier is trained for detecting family cars and should predict the labels of the test set X_test containing 100 cars. The label vector y_test contains a one for a family car and otherwise a zero. Altogether 25 items are labeled as family cars (abbreviated *fc*) and 75 are labeled as non-family cars (abbreviated *nfc*).

If a *fc* is correctly classified, it is called a `true positive (tp)` result. If a *nfc* is

correctly classified, it is called a *true negative (tn)*. If the model falsely predict *nfc* for a family car, its called *false negative (fn)* (or a miss) and vice versa *false positive (fp)*, if a family car is predicted, but actually labeled as non-family car (a false alarm). These values are the basic metrics for a binary classification and can be depicted in a so called confusion matrix:

$$\begin{bmatrix} True\ negatives\ (TN) & False\ positives\ (FP) \\ False\ negatives\ (FN) & True\ positives\ (TP) \end{bmatrix} \tag{2.10}$$

The actual accuracy measurement can be defined by using the following formula 2.11.

$$acc = \frac{tp + tn}{tp + tn + fp + fn} \tag{2.11}$$

Two extreme cases are assumed in the upcoming lines and shown in formula 2.12 and 2.13. If 100 family cars are predicted the accuracy is 25%. If 100 non-family cars are predicted the accuracy is 75%.

$$acc_1 = \frac{25}{100} = 0{,}25 \tag{2.12}$$

$$acc_2 = \frac{75}{100} = 0{,}75 \tag{2.13}$$

Image this set is even more imbalanced. 100 cars and just one is a family car. By choosing always the class *nfc* an accuracy of 99% is achieved. Therefor it is important to consider effects of missing or lacking balance of the categories within a data set. Other measurements are important as well. Precision or *positive predictive value (PPV)* will measure how much positive predictions are actually positive [MG17]. Recall or *true positive rate (TPR)* however is used as measurement if the avoidance of false negatives is required. The definitions for both measurements are depicted in formula 2.14 and 2.15.

$$ppv = \frac{tp}{tp + fp} \tag{2.14}$$

$$tpr = \frac{tp}{tp + fn} \tag{2.15}$$

A harmonic mean of the precision and recall is called f_1 *score*. It is defined as

follows:

$$f_1 = 2 * \frac{PPV * TPR}{PPV + TPR} = \frac{2 * tp}{2 * tp + fp + fn} \tag{2.16}$$

In order to optimize the parameter of a cluster algorithm like k-means specific algorithms are available. One is the so-called *silhouette coefficient*. It will provide a measurement for the quality of the clustering that is independent of k [FS08]. A silhouette coefficient s_i of a point p_i within the cluster is defined as:

$$s_i = \frac{b_i - a_i}{max(a_i, b_i)} \tag{2.17}$$

The variable a_i represents the average distance of p_i to all other points in the same cluster. Let b_i denote the minimum average distance from p_i to all points in the other clusters. The target set or co-domain of s_i will be in the range of $[-1, 1]$. If the average intra cluster distance is low and the distance to other clusters high, the silhouette coefficient will be close to 1. This indicates that the point is well-clustered. A value of -1 indicates that the point is clustered poorly. In order to evaluate the quality of clustering one can compute the average silhouette coefficient of all points [FS08]. This should be performed for several $k \in K$ to be able to compare the results. The silhouette coefficient is not a replacement for the human examination and interpretation of the results.

2.2.5 Decision Trees and Random Forests

This section deals with a supervised machine learning algorithm named *decision trees* and elaborate on the ensemble approach *random forests*.
Decision trees are used for supervised learning. They can be applied to regression and classification problems. A decision tree is composed of decision nodes and terminal leaves. The are connected through edges. The amount of outgoing edges to other child nodes can be in binary or non-binary form. In general the model is build by using a training set S. Let F^i denotes possible values for a feature f_i with $i = 1, ..., D$ and $L = 1, ..., K$ for the possible class labels. D stands for the feature amount and K for the amount of classes. The set S is defined as the proper subset of the following Cartesian product: $S \subset F^1 \times ... \times F^D \times L$ [RJPD14].
The process starts at the root node and for each node, which is not a leaf, one or several appropriate splitting features $f_i, ...$ are assigned. If one splitting feature is

used the tree is called univariate. If several splitting features are used the tree is called multivariate. The assignment of splitting features $f_i, ..$ is crucial for the decision tree construction algorithm. The splitting features will split the node into child nodes. Usually this is done by an *impurity measure* calculated for the corresponding subset S_q of the training data set S [RJPD14]. At the top of the tree the most significant features for decision nodes are used. The child nodes at the bottom will assign the data points to their categories in a more finer way. A split is pure if after the split, for all branches, all the instances choosing a branch belong to the same class [Alp14]. This results in a leaf node, which terminates the branch of the tree.
The used impurity measure and the child node structure are the main characteristics of decision trees. They differ for the used tree algorithm. Two popular examples for decision trees are ID3 or CART. CART can be used for classification and regression. It is a univariate binary decision tree and uses the Gini index as the impurity measure. ID3 however produces a non-binary tree and uses entropy to measure impurity [RJPD14].
Depiction 2.14 contains a simple data set and its corresponding decision tree. The data set consists of two classes C_1 and C_2 depicted as circles and rectangles in a two-dimensional space. The x- and y-axis are used to represent the two features x_1, x_2 for the data points. At the right a possible decision tree model for this data set is shown.

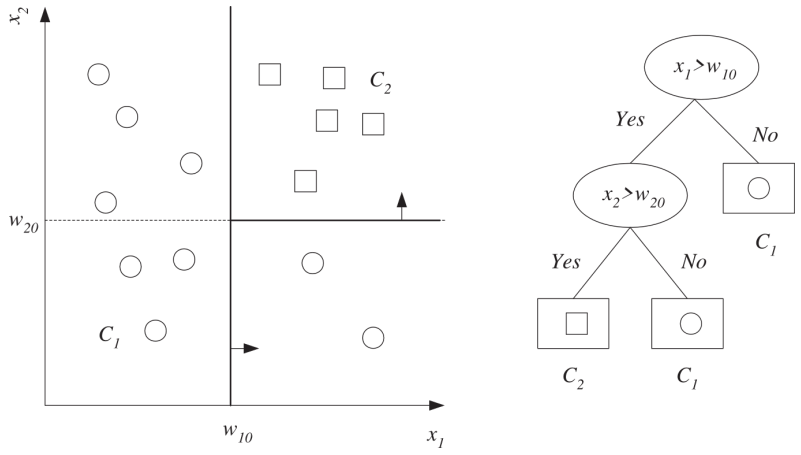

Figure 2.14: Example of a data set and the corresponding decision tree [Alp14]

In example 2.14 the first decision node divides data along the x-axis using a value w_{10}. If the feature x_1 has a value below w_{10}, then category C_1 is chosen. For this category the right node is a pure split and therefor defined as a leaf node. The left node in the same hierarchy is still impure and therefor a new decision node with the

test function $x_2 > w_{20}$ is applied. This will split the data according to the feature x_2 on the y-axis. The result are two pure splits, which cause the creation of two leaf nodes.
Advantages of decision trees are its simplicity, little data preparation, included feature extraction and interpretability of the model. This results in the ability to visualize the model [Alp14]. Decision trees can handle numerical as well as categorical data. Disadvantages are overfitting and instability. If a small amount of data changes, it can result in a complete new tree structure. If classes dominate the training set, it results possibly in a biased tree.
One solution to mitigate overfitting, is limiting the maximum depth of the tree. Another would be to set a threshold for the minimum number of samples to create a leaf node. A biased tree can be prevented by balancing the data set beforehand.
Decision trees can be used in an *ensemble approach*. This means several models using the same machine learning algorithm are build and used for prediction. It will affect the learning error in comparison to one single machine learning algorithm. Two popular examples are *bagging (bootstrap aggregating)* and *boosting*. Bagging will randomly choose subsets from the data set. This is done by bootstrapping, which means a random sampling with replacement. The resulting bootstrap subset it used to train the specific model. This is done in parallel and independently [SD02]. For decision trees, *random forests* is a popular bagging approach. The number of used trees in the forest is specified by the user. Bagging helps to reduce the variance and therefor prevent overfitting of the model. A final prediction is given by averaging the results of the individual trees in the forest. They will often provide better results than an individual model.
Boosting in comparison works in a sequential manner. A model choose a subset of training data and trains the model. The training data is then evaluated on the model. At each step of boosting the training data is reweighted, such that incorrectly classified data elements get a higher weight in the new training set [SD02]. Data elements with higher weight will be preferred in the subset for the next boosting step. Boosting will improve the bias.

2.3 Relevant Tools

This section will outline important tools and libraries used in this master's thesis. They are crucial for realizing proposed functionality and fulfilling requirements, described in chapter 3.

2.3.1 Elasticsearch

Elasticsearch is a software for building search-oriented applications. It was initially started by Shay Banon and published in 2010. Elasticsearch is able to store, search and replicate data in a cluster of computers [KR13]. Elasticsearch uses Apache Lucene an open-source full text search library. Apache Lucene is licensed under Apache License 2.0 [The18]. Elasticsearch is written in Java and therefor cross-platform compatible. It offers scalable indexing performance, powerful searching and sorting functionality.
Today the company *elastic* develops the open source software Elastic Stack, containing Elasticsearch and additional software products. For example *Kibana* is software to primarily visualize data from Elasticsearch and to navigate through the Elastic Stack. Another example is *X-Pack*, a plugin to enable graph visualization and machine learning in Kibana using data from Elasticsearch. X-Pack [14] offers additional functionality like monitoring, security, reporting and alerting.
Elasticsearch is currently released in version 6.2 and requires the Java Virtual Machine (JVM) to run on various operating systems. Currently it requires Java 8.
By default, Elasticsearch listens on `http://localhost:9200`. It provides a REST-API to configure features like cluster health, data storage, data query, searching data and more advanced requests like sorting or filtering data. The complete reference for version 6.2 can be found under the reference document [15].
The basic concept of Elasticsearch contains several important terms [KR13]:

- Data is stored in **indices**, which can be compared to databases in SQL. Apache Lucene is used to access an index.

[14] https://www.elastic.co/products/x-pack (Last access on 02.05.2018)
[15] https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html (Last access on 07.05.2018)

- An **index** contains documents, the main entities in Elasticsearch. A document can contain one or multiple fields and can be seen as key-value-storage. The JSON-Format is used to store data in a document.
- A **mapping** is used to describe the schema of a document within an index and the type of fields. Elasticsearch is able to automatically recognize a data type. Since version 6.0, only a single mapping can exist within an index.
- A **type** describes the data type of a value. This can be for example a text or an integer type.
- Elasticsearch runs as a server process and is optimized for working in **clusters**. A **node** is one Elasticsearch process. To handle bigger load or to enable replication several nodes run within an Elasticsearch cluster. This way more data can be stored than a single machine can handle.
- Elasticsearch splits data into several so-called **shards**. They can be assigned to several nodes. If a shard is copied to another node a replica is created.

To interact with Elasticsearch a powerful REST-API is provided. To demonstrate the API the storage, modification and retrieval of blog posts is utilized.
The Indices API is used to manage indices, index settings, aliases, mappings and index templates [Ela18].
To create a sample index *blog_posts* the `HTTP PUT` command is used. This is depicted in code snippet 1. The URL directly addresses the new index. The body can contain various settings and the mapping for the index. In this case the mapping is named *post* including three defined properties. They are named *id*, *content* and *author* with the appropriate data types: `integer` and `text`. It is not necessary to provide a mapping for an index. In Elasticsearch it is possible to provide no body alongside the PUT command.

```
Request method: PUT
URL: http://localhost:9200/blog_posts
Body:
{
    "mappings" : {
        "post" : {
            "properties" : {
                "id" : { "type" : "integer" },
                "content" : { "type" : "text" },
                "author" : { "type" : "text" }
            }
        }
    }
}
```

Code snippet 1: Create a new sample index *blog_posts* with mapping *post*

To create, update or delete documents within an index, the Document-API is used. This API contains sub-APIs like the Update-API or Get-API. To demonstrate the creation of a simple blog post within the previously generated index *blog_posts*, a simple example is shown in snippet 2. As shown in the URL the mapping *posts* is used in conjunction with the id 1. It is possible to create a document without having the index created in advance. An unknown index is automatically created by Elasticsearch.

```
Request method: PUT
URL: http://localhost:9200/blog_posts/post/1
Body:
{
    "id" : 1,
    "content" : "This is an example blog post",
    "author" : "Johannes"
}
```

Code snippet 2: Create a document with id *1* in index *blog_posts* using the mapping *post*

The developer can easily access the post by calling id 1. As shown in code snippet 2 to receive the content simply replace the PUT with the GET command. To operate on

the data set in a more advanced way the `_search` keyword can be used. As shown in code snippet 3 if the developer wants to receive the first ten ids in the index *blog_posts* additional information can be appended to the request.

```
Request method: GET
URL: http://localhost:9200/blog_posts/_search
Parameters:
{
    "from": 0, "size": 10,
    "_source": "id"
}
```

Code snippet 3: Use `_search` keyword for receiving the first ten ids from the index

The body contains a key `from` to set the start index of the documents and a key `size` to specify the amount of documents to receive. The field `_source` can contain specific fields of the document, which are returned on request. In this example only the `id` field will be returned.

```
Request method: GET
URL: http://localhost:9200/blog_posts/_search
Parameters:
{
    "query": { "match": { "content": "travel" } }
}
```

Code snippet 4: Looking for blog posts containing the term *travel*

Another useful API command is the keyword `_update`, to update existing documents with new information. It is appended to the id in the URL. The `POST` command is used and the target URL is `http://localhost:9200/blog_posts/post/1/_update`. The updated information is added to the body of the HTTP message.
The search API can also have a query string inserted, which supports regular expressions. This enables more powerful search requests. The example in snippet 4 returns all blog posts containing the term *travel* in the content field.

2.3.2 Scikit-learn and SciPy

Scikit-learn is a machine learning library for the Python programming language. It is open source software licensed under the BSD license [PVG+11]. Scikit-learn was initially started by David Cournapeau in 2007 as a Google Summer of Code project. In 2010 members of the French research institute INRIA [16] took leadership and made the first public release. Since then, more people joined and a new version is released every 3 month [Sci18c].
Scikit-learn is continuously developed, with a solid and fast implementation and an outstanding documentation of the implemented algorithms. It is used worldwide by hobbyists, researchers and corporations like the Otto group or Spotify. It provides classes for unsupervised and supervised machine learning approaches, like the described k-means, decision trees and random forests. It also provides efficient data structures for evaluation and model improvement.
Often used alongside scikit-learn is the *SciPy* ecosystem [17]. It provides several packages with different core features. For example *NumPy* [18] provides n-dimensional array structures with powerful linear algebra functionality. *Matplotlib* [19] is a 2D plotting library, which is able to provide fast and easy plotting results. It works seamlessly with the data structures of *SciPy*.

2.3.3 Android Debug Bridge and `fastboot`

This subsection deals with the interaction between the host developer machine and the Android device. Two popular command line tools of the official Android SDK will be described.
The *Android Debug Bridge (ADB)* [20] is a command-line tool as part of the Android SDK platform-tools and enables communication with an emulated or physical Android device. It is able perform various interactions with the device like installing apps, exchanging data, launching apps or opening the Linux shell. It uses a client-server infrastructure, which is divided into the following components [And18a]:

[16] https://www.inria.fr/en/ (Last access on 06.05.2018)
[17] https://www.scipy.org/ (Last access on 06.05.2018)
[18] http://www.numpy.org/ (Last access on 06.05.2018)
[19] https://matplotlib.org/ (Last access on 06.05.2018)
[20] https://developer.android.com/studio/command-line/adb (Last access on 06.05.2018)

- **Adb client:** This software runs on the development machine. The command `adb` is used to start commands on the device

- **Adb daemon:** The daemon is running on the device and executes commands sent from the client.

- **Adb server:** This software runs on the development machine and manages the client server-connections.

The adb server listens on TCP port 5037, so clients can connect. To connect a adb daemon through USB to the adb server, the adb debugging options on the device needs to be enabled. To do so, the hidden developer options on Android need to be activated. Since API level 17 it is required to accept the RSA key of the development machine, in order to use adb. This ensures that the user is aware of the adb debugging who is allowed to unlock the device [And18a]. To enable a wireless debug bridge it is possible to use adb through wireless network connection, after an initial setup over USB.
There are a lot of adb commands available, which empowers the user with full control over the device. The `adb shell` command lets the user interact with the remote shell of the device. Entering the command opens the shell and enables the user to type in commands like in a common *Secure Shell (SSH)* session. In combination with the powerful activity manager, which is executed by invoking `adb shell am`, a user is able to start components like activities and services of an Android app. It allows to stop processes and dump memory information. The *PackageManager* is executed by calling `adb shell pm`. This manager lets the user control app packages and permission information. It is even able to grant and revoke permissions, create or delete users. The *screencap* tool `adb shell screencap` is used to make screenshots or video recordings of the Android user interface.
of the device is rooted and adb is allowed to execute superuser commands, the `adb shell su -c <command>` lets the user execute commands with superuser privileges. The combination of `adb shell` and `input` is able to send nearly all input events to the device, like a touch, scroll or swipe events with specific position information. Also the `dumpsys` command, described in 2.1.3, can be used via adb.
Another useful tool often used in combination with `adb` is `fastboot`. To execute the command successfully, the device needs to be in fastboot mode. This can be achieved

by using the `adb reboot bootloader` command. The binary is a toolbox for flashing an Android device, unlocking the device or erasing data. To prevent modifications the bootloader for Android devices is in general locked and encrypted. Some vendors offer unlock tokens in order to let customers unlock their bootloaders. The unlock process is performed by executing the command `fastboot oem unlock`. The `fastboot flash` command then flashes the images to the device.

2.3.4 XPosed Framework

The *XPosed framework* can be installed on rooted Android devices and provides an extensible environment through modules found in the *XPosed Module Repository*. These modules have the ability to affect the system's behavior, for example by adding functionality and additional tweaks. This is realized without touching the installed APKs [Rov18]. The authors of the Xposed framework are *rovo89* and *Tungstwenty*. The source code is available on GitHub [21].
A popular module is *XBlast Tools* [22] , which is able to modify the appearance of the Android user interface. *Fake My GPS* [23] for example enables to inject forged GPS coordinates into most apps on the system. General information, downloads and module descriptions are available at the project's website [24]. Older versions of the framework are listed on the main page. Newer version of the framework can be found on the XDA website [25].
Obtaining root access on a device includes the following steps. First, the bootloader is unlocked by using the described `fastboot` command. Next, a custom recovery like TWRP [26] is flashed on the device. It modifies the system image so it is able to use the `su` package. Afterwards the XPosed framework can be installed and be activated using superuser privileges.
If an Android app is launched the activity manager calls the `app_process` command. The behavior is documented in the Android source code [27]. This command will handle the runtime and the startup of an application.
To modify the system's behavior XPosed is doing two things. First, the Xposed

[21] https://github.com/rovo89/XposedInstaller (Last access on 06.05.2018)
[22] http://repo.xposed.info/module/ind.fem.black.xposed.mods (Last access on 06.05.2018)
[23] http://repo.xposed.info/module/com.fakemygps.android (Last access on 06.05.2018)
[24] http://repo.xposed.info/ (Last access on 06.05.2018)
[25] http://dl-xda.xposed.info/framework/ (Last access on 06.05.2018)
[26] https://twrp.me/ (Last access on 06.05.2018)
[27] https://android.googlesource.com/ (Last access on 06.05.2018)

framework injects code into the `app_process` to load an Xposed library file. The main method is invoked which initiates data structures for the XPosed API and the loading of modules.
Afterwards a module app can access the XPosed API by integrating the appropriate dependency and start hooking methods of the Android operation system. Xposed methods like `findAndHookMethod` [28] are used to hook system methods and inject callbacks before or after the method invocation. This approach is able to modify the functionality of Android system calls. The feature replaces the genuine method of the Android API with a native implementation of XPosed. The native code then takes care of the implemented callbacks and of invoking the genuine Android method.
A useful module for dynamic analysis purposes is the Droidmon XPosed module. This is a dalvik monitoring framework for CuckooDroid, an Android malware analysis toolkit. It is developed by Idan Revivo and Ofer Caspi. CuckooDroid is contributed by Check Point Software Technologies [RC18]. This small application is able to log method calls of running applications to a log file in a unified format.

```
{
    "hookConfigs": [
        {
            "class_name": "android.telephony.TelephonyManager",
            "method": "getDeviceId",
            "thisObject": false,
            "type": "fingerprint"
        }
    ],
    "trace": false
}
```

Code snippet 5: Example snippet from `hooks.json`

The type of logged method calls is defined in the `hooks.json` file. It adds a log function while hooking several relevant Android system methods. The example in code snippet 5 shows the results in logging the call of method `getDeviceID()` of the Android's `TelephonyManager`.

[28] http://api.xposed.info/reference/de/robv/android/xposed/XposedHelpers.html (Last access on 06.05.2018)

2.4 Relevant Online Services

This section lists relevant online services used in this thesis. The service *Virustotal* is used for additional analysis purposes of Android apps and is outlined in section 2.4.1. In order to collect Android apps for the analysis, Android app repositories are necessary. Android malware can be accessed by the services described in chapter 2.4.2. The subsection 2.4.3 outlines repositories containing regular benign Android apps.

2.4.1 Virustotal

Virustotal is an online service for inspecting files by aggregating results of over 70 antivirus scanners and URL blacklisting services. Virustotal provides free and restricted usage for non-commercial use. Files can be submitted through web interface, desktop apps, browser extensions or a programmatic API [Vir18b].
Relevant for this thesis is the usage of the programmatic API. With the registration the user receives a personal API key to gain access. The public access is restricted to 4 requests per minute. A paid premium API access is offered as private API. It provides full access with full request rate. The private API also provides more functionality. For this thesis the public access is sufficient. All API communication uses HTTP and REST.
To receive a scan result for a file, a POST request including API key and file hash is sent to the API, as shown in code snippet 6.

```
1 Request method: GET
2 URL: https://www.virustotal.com/vtapi/v2/file/report
3 Parameters:
4 {
5     'apikey': '<API-KEY>',
6     'resource': 'bc4ecf6f5998eaaa57751cc7fc70d75c'
7 }
```

Code snippet 6: Virustotal request for a resource identified by MD5 hash

The response for the request is listed in code snippet 7. The response is represented as JSON and contains various key-value pairs. If the `response_code` is equal to one,

the file was successfully found in the Virustotal database. For an unknown file a zero is returned, whereas a -2 is returned for a queued scan. The field `verbose_msg` returns a description of the status. Attached are several hash values of the requested file, a field *permalink* to the Virustotal report and the scan date. The key `scans` contains all results of the used scan engines as separated JSON objects. The `total` field returns the total amount of used scan engines, whereas the `positives` field return the positive hits by these scan engines. The example in code snippet 7 contains the results of the *AVG* scan engine for brevity.

```
{
    'response_code': 1,
    'verbose_msg': 'Scan finished, information embedded',
    'resource': 'bc4ecf6f5998eaaa57751cc7fc70d75c',
    'scan_id': '10b4447e480ad391489de98be74a06445081948c78f3b9c0b88a15a61fa257ca
        -1516489560',
    'md5': 'bc4ecf6f5998eaaa57751cc7fc70d75c',
    'sha1': 'ddac83ee7158bc6a78573413d730d261642c7c28',
    'sha256': '10b4447e480ad391489de98be74a06445081948c78f3b9c0b88a15a61fa257ca',
    'scan_date': '2018-01-20 23:06:00"',
    'permalink': 'https://www.virustotal.com/file/
        10b4447e480ad391489de98be74a06445081948c78f3b9c0b88a15a61fa257ca/
        analysis/1516489560',
    'positives': 0,
    'total': 60,
    'scans': {
        "AVG": {
            "detected": false,
            "version": "17.9.3761.0",
            "result": null,
            "update": "20180120"
        },
        ....
    }
}
```

Code snippet 7: Virustotal response for the request listed in code snippet 6

2.4.2 Android Malware Repositories

This subsection introduces online repositories for Android malware. First the service *Virusshare* is described, followed by the *Android Malware Genome Project* and the *Drebin Dataset*.

Virusshare is a repository of malware samples to provide security researchers, forensic analysts and incident responders access to the samples. They can inspect and analyze the behavior and specifics of a certain malware themselves [Vir18a]. The repository is created and maintained by *VXShare* [29]. Access is invitation only and must be granted by the administrator. Information about the user and the reason for requiring access must be stated in the access request. Several malware samples are typically zipped into packages. They range between 10 to 100 gigabytes and can be downloaded directly from the website. A Zip archive contains several samples from different architectures and usually covers a defined time range. Special requests are zipped into additional packages, which is available via torrent download.

The *Android Malware Genome Project* [30] is an Android malware collection by Zhou and Jiang of the Department of Computer Science of the North Carolina State University. It is described in the scientific publication *Dissecting Android Malware: Characterization and Evolution* [ZJ12]. It contains approximately 1200 malware apps. The collection is utilized in several scientific publications. The composition of several malware families is valuable, but unfortunately the malware is outdated. It is stated that the apps were retrieved from beginning August 2010 to October 2011. Moreover the project was stopped in end of 2015.

Another Android malware collection is the *Drebin Dataset* [31] resulting from a scientific publication *DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket* published by researchers from the University of Göttingen and Siemens CERT [ASH+14]. The data set contains around 5,560 application from 179 malware families. The samples were collected from August 2010 to October 2012. The data set is still available if an access to their repository is granted. The collection is comparable to the malware genome project.

There a several smaller online malware collections available for example on GitHub [32] [33].

[29] https://twitter.com/VXShare (Last access on 07.05.2018)

[30] http://www.malgenomeproject.org/ (Last access on 07.05.2018)

[31] https://www.sec.cs.tu-bs.de/ danarp/drebin/ (Last access on 07.05.2018)

[32] https://github.com/ashishb/android-malware (Last access on 07.05.2018)

[33] https://github.com/fouroctets/Android-Malware-Samples (Last access on 07.05.2018)

2.4.3 Android App Repositories

This subsection introduces app repositories for benign Android applications. A large collection named *Androzoo* is described, followed by mentioning additional repositories.
Androzoo is a collection of Android apps maintained by the Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg and is described in the Paper *AndroZoo: Collecting Millions of Android Apps for the Research Community* [ABKLT16]. It is online accessible after requesting access on the Androzoo website [34]. The collection currently contains around 5,8 million different APKs from several sources like the official Google Play store [35], Anzhi [36], AppChina [37] and F-Droid [38], the popular open source repository of Android apps. The project provides additional meta data such as APK, manifest and permission information.
Google Play Store [39] is the most popular app store for Android applications. It offers a variety of apps organized in categories. It is not possible to directly download the apps in APK format, but the third party service *APK Downloader* [40] provides this functionality.
Other Android app repositories also allow a direct download of APKs for example *APKMirror* [41] or Apkpure [42]. Both websites offer downloads of benign apps from different categories and latest versions including hash sums. Another example is *Appsapk* [43] an online platform for downloading popular, recent and free Android applications.

2.5 Related Work

This section introduces related and mostly recent scientific research with focus on Android malware detection. The papers use different analysis approaches and machine

[34]https://androzoo.uni.lu/ (Last access on 07.05.2018)
[35]https://play.google.com/store (Last access on 07.05.2018)
[36]http://www.anzhi.com/ (Last access on 07.05.18)
[37]http://www.appchina.com/ (Last access on 07.05.18)
[38]https://f-droid.org/ (Last access on 08.05.2018)
[39]https://play.google.com/store?hl=en (Last access on 08.05.2018)
[40]https://apps.evozi.com/apk-downloader (Last access on 08.05.18)
[41]https://www.apkmirror.com/ (Last access on 08.05.2018)
[42]https://apkpure.com/app (Last access on 08.05.2018)
[43]https://www.appsapk.com (Last access on 08.05.2018)

learning techniques.
The publication *A review on feature selection in mobile malware detection* from 2015 by Feizollah, Anuar, Salleh and Wahab from the University of Malaya, Kuala Lumpur analyzed 100 scientific papers with the perspective of feature selection in Android malware detection [FASW15]. The researchers divide features into four categories: static, dynamic, hybrid and metadata. They discussed the details of these features and their propagation within the research community. The paper outlines an overview of used datasets and the most common evaluation techniques. It provides a detailed insight into challenges and open research areas in the mobile malware detection research field.
The publication *Accurate mobile malware detection and classification in the cloud* from 2015 by Wang, Yang and Zeng from the National University of Defense Technology in Changsha, China, introduces a hybrid mobile malware detection and classification system [WYZ15]. It uses an anomaly-based and a signature-based detection engine. The anomaly-based step checks for benign or abnormal apps by using a one-class *Support Vector Machine (SVM)* supervised algorithm. For this approach only dynamic features are used. The next step checks if the abnormal app is an already known malware application. If not, the app is handed over to the signature-based engine. This uses static and dynamic features in a *Linear Support Vector Classification (SVC)* approach. Static features involve the manifest file and disassembled code. For dynamic analysis Cuckoodroid is used (see chapter 2.1.3). User interactions are simulated with Robotium [44], which is not maintained any longer. Details about the performed interactions are not published. Feature selection is done with univariant statistics. The dataset comprises 6000 benign and 5560 malware apps. Benign apps are crawled from three Chinese app repositories. Malware apps are completely taken from the outdated Drebin dataset. No details about the importance of the used features are given. Resulting experiments show high true positives rates in detecting abnormal malware from benign apps [WYZ15].
The paper *PIndroid: A novel Android malware detection system using ensemble learning methods* from 2017 by Idrees, Rajarajan et. al. introduce PIndroid a system for classifying Android malware. A static analysis is implemented, which processes permissions and intents of Android apps by reading the manifest file. The approach compares six different classifiers. Resulting f_1 scores range between 95,6% and 99,6%. Three ensembles of machine learning techniques are used and evaluated. The best

[44]https://github.com/robotiumtech/robotium (Last access on 08.05.2018)

f_1 score lies around 99,7% [IRC+17]. The prototype uses an overall amount of 1745 Android apps, with 1300 malicious and 445 benign apps. 1000 of these malware samples come from the Genome and 100 from the Drebin dataset. Nearly 85% of the malware was collected from outdated repositories.

The publication *A pragmatic android malware detection procedure* from 2017 by Palumbo, Sayfullina et. al. explains import requirements for deploying Android malware detection systems in real world [PSK+17]. They designed and implemented an approach for automatic Android malware detection systems using a machine learning ensemble approach with a static analysis to extract features. The first decision layer uses Naive Bayes classifiers. The resulting confidence values are processed in the second decision layer with a Support Vector Machine. The total amount of selected features is 40 million. They are reduced to a threshold of 100 features by using dedicated experiments and Naive Bayes classifier [PSK+17]. The total amount of samples used in this scientific work is stated with an outstanding amount of one million samples, whereas 120 thousand are used for the training model. The latest apps in this sample are retrieved in end of 2015. The evaluation used a continuous stream of real-life Android applications over a period of three month after the initial training of the model. The evaluation shows a false positive rate of 4.07% and a true positive rate of 90.82% on the evolving stream [PSK+17].

The publication *Android Malware Classification by Applying Online Machine Learning* from 2016 by Pektaş, Çavdar and Acarman from the Galatasaray University Istanbul uses CuckooDroid (see section 2.1.3) to extract runtime features of Android malware. An online machine learning approach is used to classify malware into their appropriate families. Around 2000 malware samples are used, gathered from the Virusshare repository. Their malware overview indicates a partial use of dated malware. The classification accuracy is evaluated and several regularization weight parameters are used. The accuracy lies between 76% and 89% [PÇA16].

The paper *A Machine Learning Approach to Android Malware Detection* from 2012 by Sahs and Khan from the University of Texas presents detection of Android malware by using an One-Class Support Vector Machine. The open source project AndroGuard [45] was used to extract static features from Android applications. This work uses the permissions and control flow graphs from the raw bytecode of methods [SK12]. For evaluation 2081 benign and 91 malicious apps were used. Only benign apps were used to train the machine learning model.

[45] https://github.com/androguard/androguard (Last access on 08.05.2018)

The publication *A hybrid approach of mobile malware detection in Android* from 2017 by Tong and Yan from the Xidian University in China and Aalto University in Espoo, Finland, primarily uses a dynamic approach for detecting Android malware [TY17]. Malicious and benign Android apps are executed and individual and sequential systems calls are collected. A static procedure extracts patterns to identify benign and malicious apps. These sets are continuously updated. A common static analysis of the Android app, e.g. by using the Android manifest, is not part of the publication. Around 2000 apps are utilized for this work. All malware comes from the Android Malware Genome Project. The overall detection rate lies around 90% [TY17].

3 Problem Description

This chapter will introduce the central problem description of this master's thesis, define a goal and the requirements. It will also refine the main research question.
In general software products or libraries continuously evolve and so does malware and also Android malware. New features and patches for bugs or security vulnerabilities are frequently applied to software products. Therefor malware which exploits security vulnerabilities needs to continuously adjust its code base to obtain new attack vectors. This results in a race between maintaining security and successfully distributing malware. Especially code changes within the operating system or libraries potentially introduce new vulnerabilities, which can be exploited by more recent versions of malware. In Android various API levels exist with varying functionality. Often older versions are no longer supported and will not receive any patches for critical vulnerabilities. Malware continuously improve itself, but the countermeasures do as well.
Training a machine learning model using outdated malware can affect the prediction accuracy for more recent malware. New malware approaches and their specific characteristics are never trained, so the specific features remain unknown to the machine learning algorithm.
The central problem is that many scientific publications, as well as more recent ones, utilize outdated malware in supervised machine learning approaches, in order to predict the detection of untrained Android malware. Often a dedicated static or dynamic analysis approach is used, which doesn't combine the results in order to mitigate drawbacks of each analysis. Dynamic analysis is often used together with click tools to increase the code coverage of the approach. The increased time and resource consumption put the static analysis in a preferred position.

3.1 Goals

To overcome problems with outdated Android malware, this thesis will focus on processing recent Android malware. One goal is to build up a collection of recent malicious and benign apps of 2017. If older malware is available at the same time it will be downloaded as well, since it could be useful for future work. In order to mitigate the drawbacks of the static and dynamic analysis approach, this master's thesis will implement a hybrid analysis prototype. Furthermore to investigate the value of automated click tools during a dynamic analysis for machine learning, this thesis will realize two different dynamic analysis approaches. Since they require static analysis results they are defined as two different hybrid analyses. A supervised machine learning approach will be implemented in order to perform classification of malware. Therefor the results of the analyses will be used. The central research question for this thesis reads as follows: *How precise is the predictive identification of recent Android malware by comparing two different hybrid analysis approaches?* The prediction performance of both will be compared in the evaluation. The thesis will be finished by answering the central research question along with a conclusion and recommended future work.

3.2 Requirements

In order to accomplish the goals and to find a solution for the problem description several requirements will be defined. The following list will outline these requirements in detail.

1. Collect Android malware and benign apps of the year 2017
2. Collect a sufficient amount of apps suitable for machine learning
3. Implement a hybrid analysis prototype with static and dynamic analysis capabilities
4. Refine the dynamic analysis into two approaches one using autonomous clicks and an alternative
5. The complete analysis must be able to handle large amounts of apps
6. Store analysis results in a suitable database

7. Implement a suited machine learning approach
8. Handle feature engineering and model optimization
9. Evaluate the prediction performance of both approaches

Both prototype implementations for analysis and machine learning should be computationally efficient and not rely on one big analysis tool [PSK+17]. The analysis prototype must be able to perform both hybrid approaches fully automated. Moreover comprehensive tests especially for the dynamic parts are required. Included libraries should have a detailed documentation and use open source licenses.

4 Concept

In order to reach the defined goals and answer the research question, a structured approach is required. This concept chapter addresses this issue by modularizing the tasks and explaining core strategies and decisions. An overview about the concept and further modules is given in the upcoming section 4.1.

4.1 Overview

The concept builds upon the goals and the defined requirements of the problem description. The tasks are split into sequential modules and build a foundation for the upcoming implementation phase. Every module serves as input for the following module until the chain is completed with the final evaluation. Decisions concerning functionality, processes and used tools are content of the concept. The chain of modules is depicted in figure 4.1.

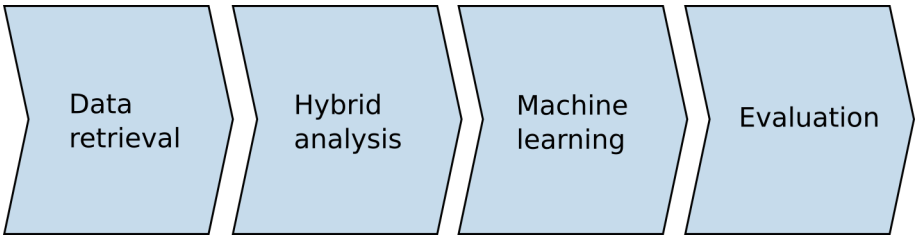

Figure 4.1: Module overview

The module *data retrieval* is outlined in section 4.2, while *hybrid analysis* can be found in section 4.3 and the module *machine learning* is described in section 4.4. The evaluation is a special case, which is not divided into concept and implementation. The entire evaluation approach is described and performed in chapter 8.

In order to illustrate the overall picture of the concept and to reveal the relationships between modules and additional components, a concept overview is shown in figure 4.2.

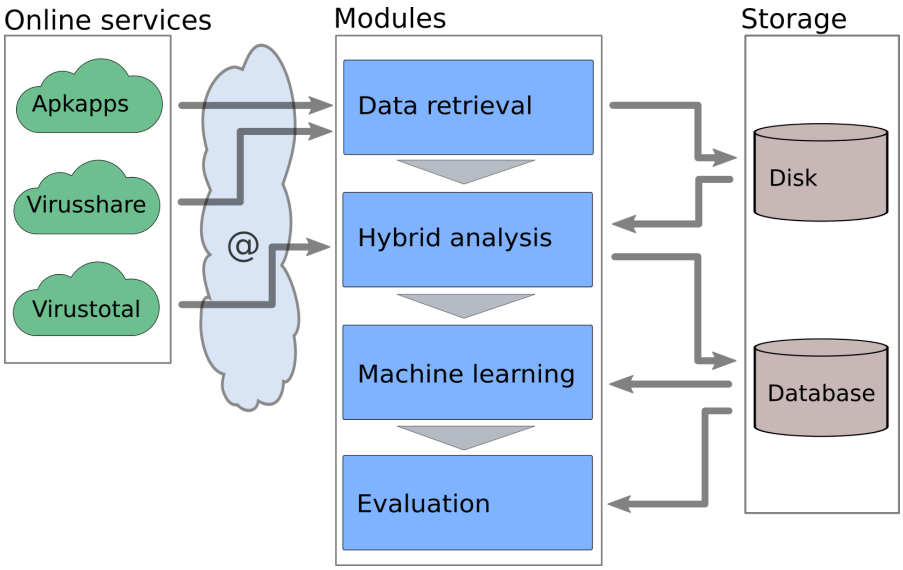

Figure 4.2: Concept overview

The box in the center of figure 4.2 contains the modules. On the left side there are the used online services and on the right side the utilized storage capabilities are shown. The module *data retrieval* will use the online services *Apksapps* and *Virusshare* to receive and store the initial data. The services are described in chapter 2.3. The module *hybrid analysis* is going to utilize the raw data collection. In order to perform the hybrid analyses the *Virustotal* online service will be used. The results will be stored in the database. The collected analysis data will be used for the module *machine learning*. The *evaluation* is going to complete the work by measuring and comparing the prediction results.

4.2 Data Retrieval

This section deals with the concept of retrieving the required apps as input data. One core requirement is to collect Android malware and benign apps of the year 2017. It is planned to store the apps on disk and perform regular backups to prevent data loss. First the decision for the Android malware repository is described, followed

by the benign app repository.
Because the Android malware repositories *Android Malware Genome Project* and *Drebin Dataset* (see section 2.4.2) contain outdated malware, they are not used within this thesis. Instead the *Virusshare* repository will be requested. The more malware is available, the better the malware ecosystem will be captured. Therefor a lot of malware is required. An invitation is needed in order to request the Virusshare service. It was granted with the help of personal email communication including a short description of the use case and the research institute.
First the special request section on the platform will be searched for recent Android malware collections. In the case of failure a request will be sent, in order to create a recent malware collection for the year 2017. If this is without results, the last possibility is to filter the zipped direct downloads with mixed malware samples for Android malware. This results in additional effort. If the repository contains older Android malware it will be downloaded as well. It might be useful for future work in this field. Details are stated in the implementation chapter 5.
In order to detect malware in the machine learning module, also benign applications from 2017 will be collected. This is going to help the learning approach to detect characteristics and differences between the two classes. It is also recommended to collect benign apps from several categories in order to model the majority of the app ecosystem. This is important, since it is easier to distinguish malware from a single app category. Therefor the machine learning approach could simply identify characteristics of this app category instead of malware.
The online service *APK downloader* (see subsection 2.4.3) is working for single APKs, but is unsuitable for massive usage. Automating the procedure using scripts seems to be tricky due to obfuscation techniques. Furthermore the service can take around three minutes to receive an application from the play store. A recent version of an app is not guaranteed since the apps are cached by the *APK downloader* service. There is also a browser plugin available.
Github also provides some projects for downloading APKs from the play store using the command line. One project is *googleplay-api* [1] which unfortunately is not working anymore. The project *gplaycli* from github [2] was also tested, but either the downloads didn't start or it took too long to finish them.
The services *APKMiror* and *Apkpure* prevent downloading the APKs programmatically by obfuscating the links. To gather benign application in the first place *Appsapk*

[1] https://github.com/egirault/googleplay-api (Last access on 09.05.2018)
[2] https://github.com/matlink/gplaycli (Last access on 09.05.2018)

will be utilized. This online service made the links clearly visible. Moreover a request to join the closed repository *Androzoo* was made. It was approved by the administrator.
In summary the sources for malware and benign Android apps were elicited. In order to receive malicious Android apps the Virusshare online service is going to be utilized and Appsapk will be used for benign apps.

4.3 Hybrid Analysis

This section introduces the concept of both hybrid analysis approaches. It is divided into a static and a dynamic analysis part. Details can be found in the subsections 4.3.1 and 4.3.2. The upcoming lines will outline the general motivation of a hybrid analysis. Furthermore it explains the depicted interconnections in the concept overview 4.2. This includes details about the online service Virustotal and the database for storing the analysis results.

The main reasons for a hybrid approach are to utilize advantages of the static and dynamic approach and simultaneously neutralize disadvantages. Advantages of a static analysis are performance, resource efficiency and the high code coverage of an application. Main disadvantages are problems in detecting dynamic code loading and handling obfuscation. The main advantages of a dynamic analysis on the other hand are the ability to handle dynamic code loading, obfuscation and revealing actual behavior of an application. Disadvantageous are the code coverage and the required resources to perform an analysis. It also needs special isolation to treat the danger of exploiting vulnerabilities, because of actually executing the code. Moreover in scientific research automated click tools (see section 2.5) are often used.

The dynamic analysis concept will deal with the two analysis approaches (see subsection 4.3.2). The hybrid approach guarantees analysis results of both worlds. It is relatively rarely used [FASW15], but enables the machine learning approach to utilize the best fitting features. This is a major advantage compared to stick with one approach beforehand.
In general the analysis phase should focus on the later prediction using machine learning. It should encompass which data can be collected from an Android app and might be useful for subsequent processing. This data is the foundation for later

feature selection.
The dynamic analysis requires results of the static analysis such as package name and components, so the static approach must be executed first. One implementation is to run the static analysis and the dynamic analysis per application. The alternative approach performs the static analysis for all apps first, which is followed by the dynamic procedure. The first approach would be more suitable if the system is running in production mode. The second procedure is a reasonable approach if a large collection of apps is present. This thesis is going to use the second approach and will run the static and dynamic analysis independently.
It is planned to implement both analyses in Java. Main reasons are its extensive usage within the research institute, platform independence and mature concurrency capabilities. The prototype focuses on automation of the analysis and will utilize tests for a successful realization.

Besides the core analysis the online service Virustotal will be used. A hash sum for every analyzed app will be created, which will be used to request aggregated virus scanner results. The result is going to be used to identify and label the collected apps as malware or benign apps. This is useful, since their is no guarantee that Appsapk only contains benign apps and Virusshare only contains malware. The usage of several anti virus scanners reduces the teacher noise for the upcoming machine learning process.

A database will be used to store static, dynamic and Virustotal results permanently. Because of the effort of the analysis and the value of the generated data, the database should support replication capabilities. Due to the free user limitations the Virustotal requests are particularly time-consuming. The Virustotal service and the dynamic analysis are going to utilize the JSON format for results. Therefor it makes sense to use a database, which supports this format natively. Thus a NoSQL database with JSON support is the most suitable choice. After comparing alternatives the Elasticsearch application is the preferred solution. Elasticsearhc is a suitable data storage because of the ability to simply store JSON documents, to use a comprehensive cluster modus including replication and to perform easy horizontal scaling of the cluster. Furthermore Elasticsearch is platform independent, well documented and search-oriented. The additional Kibana application enables the presentation of results in a web UI as well as various chart views. Executing static and dynamic approaches will result in a very large amount reports. Therefor it is planned to use individual indices with specific mapping for every analysis approach. This is going to

improve the structure and loading time of the indices. X-Pack will not be used due to license costs. To uniquely distinguish analysis results of apps, a SHA256 hash sum for an APK will be generated and used as a unique identifier within the database. Details about the static analysis will be outlined in the upcoming subsection.

4.3.1 Static Analysis

This part will introduce decisions and details about the static analysis to provide the basis for the implementation. It will first elaborate on static data, which can be extracted from an Android application. Afterwards suitable tools and libraries for a realization will be determined.
The essential input for the static analysis is the *Android Package (APK)*. It contains several valuable information described in chapter 2.1. Due to planned machine learning it is important to focus on data that is usable as a feature. An orientation on published scientific work in this specialized field is helpful. The publication *A review on feature selection in mobile malware detection* [FASW15], which summarizes over 100 scientific papers, states that scientific research in the last years mainly utilizes the program code and the manifest file as static features [FASW15]. In more detail it describes that permissions, bytecode, intent filters, network addresses, strings and hardware components are commonly used. Therefor it is planned to extract at least all of these features.
For a well structured approach, it makes sense to divide the required data into smaller components. The APK is divided into three components. Each contains detailed information what is going to be extracted and is going to be used as features in a later stage. The components and data are depicted in figure 4.3.

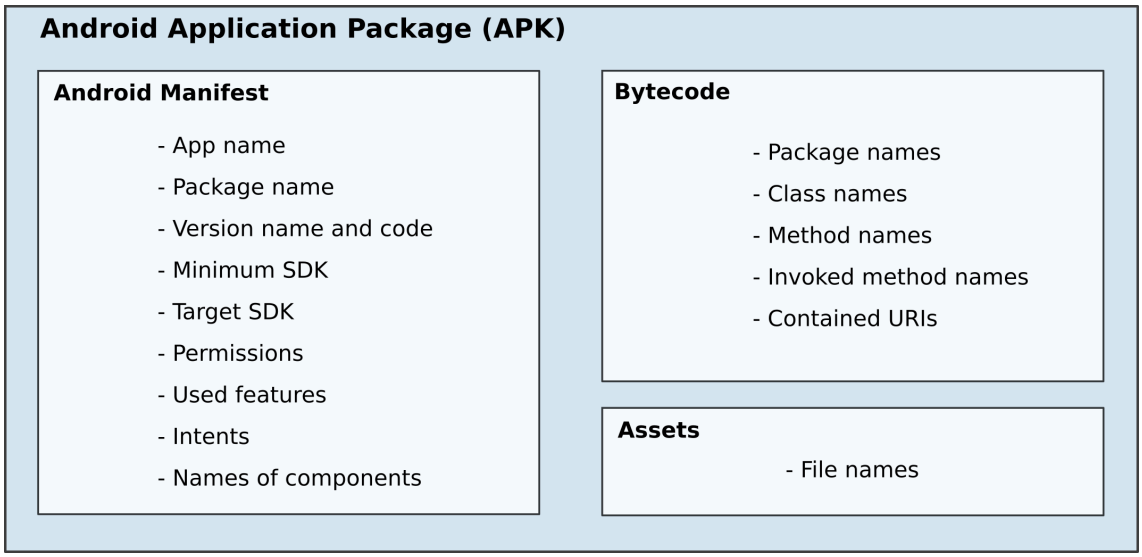

Figure 4.3: Planned data extraction through static analysis

The Android manifest contains core information about the application such as names of the components and permissions. Bytecode represents the `classes.dex` files. Information about class names, method names, invocations and URIs can be extracted. Assets contains names of the included assets. In this work files from the `assets` and the `lib` folder will be processed. There is improvement for the analyzed data, which can be implemented in the future. An improved approach would be the analysis of the content type (Media type or MIME [3]) of the assets. Found native code files could be analyzed using a native analysis. This would be an entire individual topic. The analysis of resource files could also be a useful feature. Though this is dropped at the moment due to feature size and the lack of expected benefits. Currently the multidex feature is not handled by the static analysis. However the missing code will be covered by the dynamic analysis. Additional information can be found in the section 9.1. The next lines will discuss the used tools in order to realize the static approach.
As described in chapter 2.1.3 tools like *Androlyzer* and *Flowdroid* are able to perform detailed flow analysis and leak recognition using control and data flow graph modeling. Handling this kind of data in machine learning approaches is more complex and has been realized in scientific projects within a single static analysis approach. One example is the publication *A Machine Learning Approach to Android Malware Detection* from Sahs and Kahn, University of Texas at Dallas in the year 2012 [SK12].

[3] https://www.iana.org/assignments/media-types/media-types.xhtml (Last access on 03.05.2018)

However implementing a hybrid analysis is more challenging and complex than a single static approach. Due to this fact using graph results from flow analysis is beyond the scope of this thesis. As described in chapter 2.1.3 *Apktool* can decode an Android application into Smali code for further processing. This might be useful as a fundamental tool for extracting further information. However this tool lacks in automatic app analysis and suitable storage of results. It also would be a significant dependency if integrated in the prototype. The static analysis of the *MobSF* framework delivers comprehensive results and is able to analyze massive amounts of apps. Nevertheless the resulting report will not be provided in an applicable format to support further processing.
All these tools offer a quite useful functionality. Mandatory features are missing sometimes or the results are inconvenient for further processing. Preventing a major dependency of one analysis tool maintains independence and protects against trouble with the lack of functionality due to bugs. It also prevents stressful debugging, helplessness due to upcoming issues and the need to replace the tools later. An improved approach should rely on compact libraries, which provide solid documentary and are well understood, flexible, replaceable and could be reimplemented if necessary. According to these guidelines a prototype using libraries is going to be implemented. It is planned to cover the program code and the manifest file with the help of two libraries. To cover the program code the *DexLib2* library will be used (see chapter 2.1.3), which provides a lot of functionality for the Dalvik bytecode. It supersedes the predecessor *DexLib*. The *APK-Parser* will help to decode the binary-encoded manifest file and will also support processing the assets files.
The final concept for the static analysis is depicted in figure 4.4.

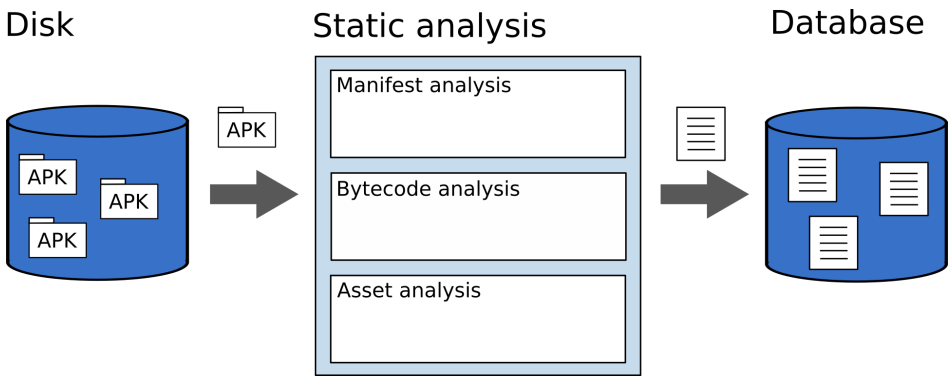

Figure 4.4: Conceptual overview of the static analysis

It is planned to read in the APKs from disk and hand them over to the static analysis

shown in the central box of figure 4.4. The analysis is composed of three components. The *Manifest Analysis* will collect the data shown in the appropriate box in figure 4.3. It will use the library *APK-Parser*. The *Bytecode analysis* will realize the data collection, mentioned in the *Bytecode* box in depiction 4.3. It will utilize the *DexLib2* library. The last component *Assets analysis* will collect asset information. The results of all components are going to be merged into a single report and will be stored into the Elasticsearch database.

4.3.2 Dynamic Analysis

This subsection deals with the concept of the dynamic analysis. According to the goals and the central research question, it is crucial to design an automated and robust analysis, which is able to handle a large amount of data. Moreover it is mandatory to realize two different approaches of a dynamic analysis. This next lines discuss the dependencies and tools for a realization. Is is also described what kind of information is feasible to be extracted from the dynamic analysis. Afterwards the two approaches are outlined in more detail.
The publication *A review on feature selection in mobile malware detection* [FASW15] states two main types of dynamic features: system calls and network traffic. The majority of dynamic analysis approaches use one of both feature types. Some publications also use system components like CPU utilization, memory usage and running processes as dynamic features. Another paper focuses on user interactions during the usage of Android apps. For the dynamic analysis it is planned to monitor Android applications during run time and track specific system calls and collect network traffic.
The dynamic analysis tools *MobSF* and *CuckooDroid* have already been described in chapter 2.1.3. They both provide a comprehensive dynamic analysis. The setup is not trivial, but a documentation is provided. The emulator images for dynamic analysis are quite old and have performance issues. The usage of a hardware device is supported, but only under strict requirements like the usage of an outdated Android API. The results of *MobSF* are reliable, but no API is provided for further processing. The results are only visible in the web app or can be exported as PDF. It would be possible to modify the source code in order to provide an API for dynamic analysis as a feature. Though this could result in unpredictable effort and additional restrictions. *MobSF* also doesn't provide the possibility to perform massive automatization of the dynamic analysis process.

CuckooDroid on the other hand provides an API to retrieve reports and is more suited for the automated analysis. Unfortunately it was not possible to get *CuckooDroid* to work despite complying with the documentation. The project has bugs, which are not solved in autumn 2017. Both tools don't provide an automatic click runner as a feature.
Due to these reasons an own implementation of a dynamic analysis will be realized. Specific requirements can always be directly integrated into the implementation. It is also possible to minimize dependencies of third party code. This analysis can be integrated into the static analysis. This would result in a merged project. Instead of emulators, real devices will be used due to performance and fidelity of running applications. More details regarding advantages and disadvantages can be found in section 2.1.3. Since several devices will be used for the dynamic analysis, it is crucial to use a concurrent design for parallel execution on the devices. It is planned to use a newer Android API Level, which will match the recent distribution of Android. The currently most used Android Version is *Marshmallow* (API 23) with about 28% of the worldwide distribution (see chapter 2.1).
To improve the structure and keep track of the dynamic analysis, the approach is separated into tree main steps. They are depicted in figure 4.5.

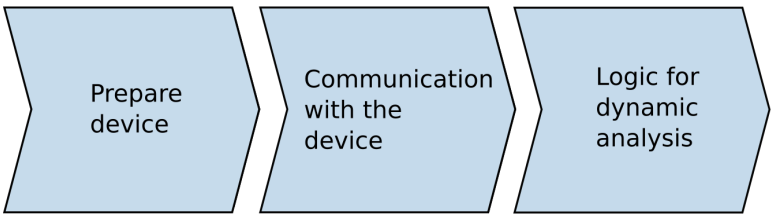

Figure 4.5: Conceptual steps of the dynamic analysis

The first step will be the preparation the device for the dynamic analysis. The *LG Nexus 5 (D821)* is going to be used as the analysis device. It is an official discontinued Google device, which is easy to unlock and to root. It is widely supported by the community and is relatively cheap to purchase. One device is provided by the DAI laboratory. Two more devices will be purchased in order to reduce the analysis duration using a parallel approach.
It is planned to prepare the device with the *XPosed* framework and monitor the API calls by using the *Droidmon* module, described in chapter 2.3.4. The defined

API hooks of the default `hooks.json` [4] will be used. It contains ten categories of API classes including different assigned methods. They are illustrated in figure 4.6. Category *globals* contains methods that store data in global storage capabilities like the Android shared preferences. Besides *dex* covers functionality for dynamically loading Dex files. The group *file* handles file operations, whereas *binder* contains receiver handling and Activity launching abilities. The category *fingerprint* covers methods in order to identify a device. The group *crypto* includes crypto operations and *reflection* a method to perform reflection operations. The category *generic* contains miscellaneous operations, *network* involves methods for establishing a network connection. The group *content* covers access to content resolvers, media and location data. In total sixty four methods will be monitored by the dynamic analysis.

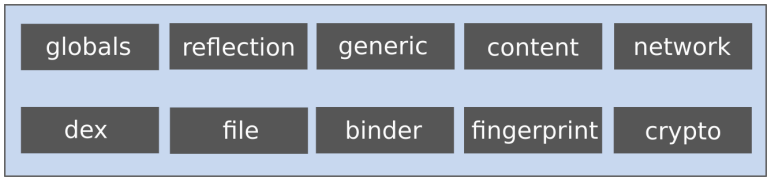

Figure 4.6: Categories for the API calls

The *Android Debug Bride (ADB)* will be used to interact with the device (see chapter 2.3.3). It is used for installing the app, launching components and clicking UI elements and dialogs. Moreover it also handles granting the permissions, rebooting the device and transferring data.
In order to reach the goals of this thesis and to answer the research question two dynamic analysis approaches will be implemented. They are required to determine the necessity of automated click tools for a machine learning approach. A click tool benefits from using real devices, due to more fluent interactions. The first one will not use any simulated click interactions. Instead it will rely on the defined activities and services of an app and is going to start them iteratively. Therefor the determined component names of the static analysis will be used. Thus this approach is called the *component approach*. Starting secondary components without the context of their main entry point could result in thrown exceptions. This potentially prevents starting the components and lose valuable content for the dynamic analysis.
However the interactive strategy is going to use an automated click engine. It will launch the main component of the application and click through further linked

[4] https://github.com/idanr1986/droidmon/blob/master/hooks.json (Last access on 15.05.2018)

components to increase code coverage. It is planned to automatically store displayed elements of every displayed UI. The elements will be further processed and clicked. In this way the components of the UI will be invoked in a breadth-first search approach. The *dump* tool is going to be utilized to determine the displayed UI elements of the app components (see chapter 2.1.3). The approach will minimize the probability of raising exceptions due to context lack. This *interactive approach* is going to use a breadth-first search to span a tree. This will initially start as much linked components as possible. Further the algorithm will follow the next branch and will repeat the procedure for the containing components and its UI elements. The approach will take care that UI components are not processed twice. The analysis will detect the switch to other apps and return to the prior UI. However a guaranteed termination of this interactive approach remains problematic. A possible solution is the usage of an time threshold, which ensure the termination after expiration. The overall performance is reduced due to built-in timeouts between click events. However the timeouts are required to act on UI interactions. The key steps of the dynamic analysis are depicted in figure 4.7.

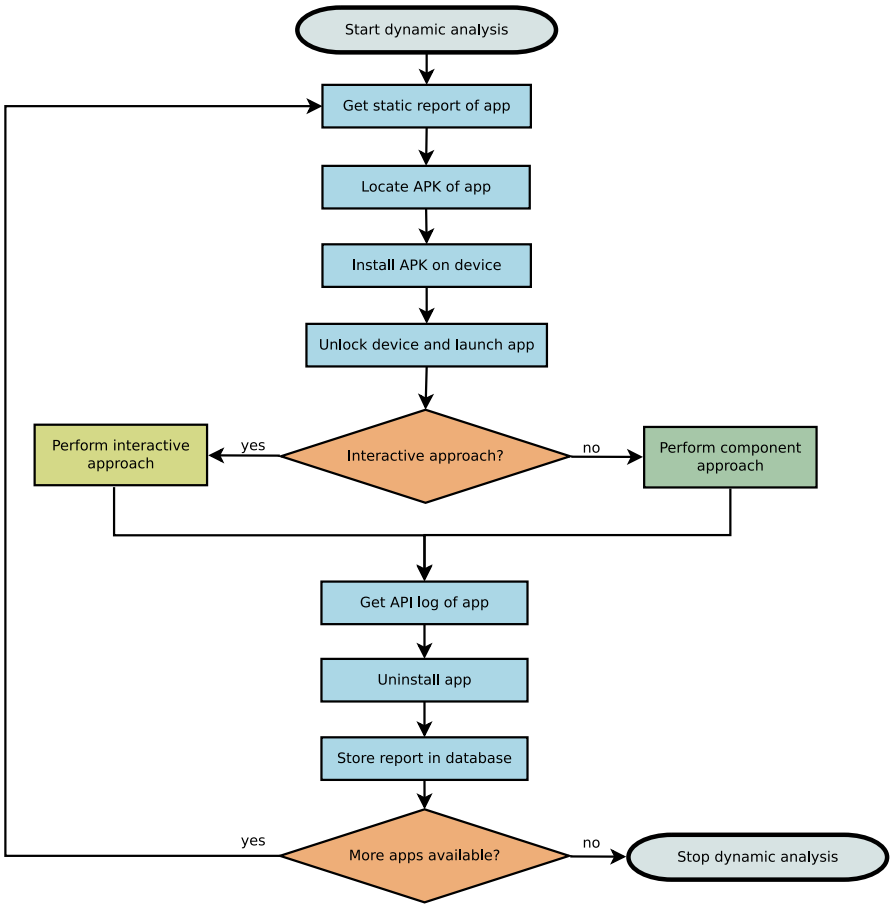

Figure 4.7: General logic of the dynamic analysis

The depicted logic will be part of a concurrent design, which is going to be executed in parallel. One dedicated thread will be used for each connected device. The dynamic analysis depends on the static analysis results. It will initially process the static report and locate the appropriate APK file using the SHA256 hash. Afterwards the app will be installed on one of the devices. In the next step it will be unlocked and the main component of the app will be launched. Either the *interactive* or the *component* approach is going to be executed. After executing the retrieval of the API log is performed. The log will be processed and the entries for the desired app will be filtered out. Afterwards the API log will be checked for successful entries, otherwise the process will be restarted. After API call extraction the app will be uninstalled and the results are going to be stored in the Elasticsearch database.
This subsection is finished with a short summary of the approach. The hybrid analysis will encompass a static and dynamic analysis implementation. The prototype is going to use compact libraries due to dependency reduction. The module will be

implemented in Java and will use Elasticsearch to store the analysis reports. The static approach is going to use program code, the manifest file and the assets of the APK. Summarized the dynamic analysis will provide a component and an interactive approach using real Android devices.

4.4 Machine Learning

This section describes details regarding the machine learning concept. The main approach should be able to predict malicious and benign apps. This can be formulated as a binary classification problem and will be realized while using benign and malware as labels. A training data set involves benign and malicious apps and will be utilized to train the model. Figure 4.8 illustrates the classification process.

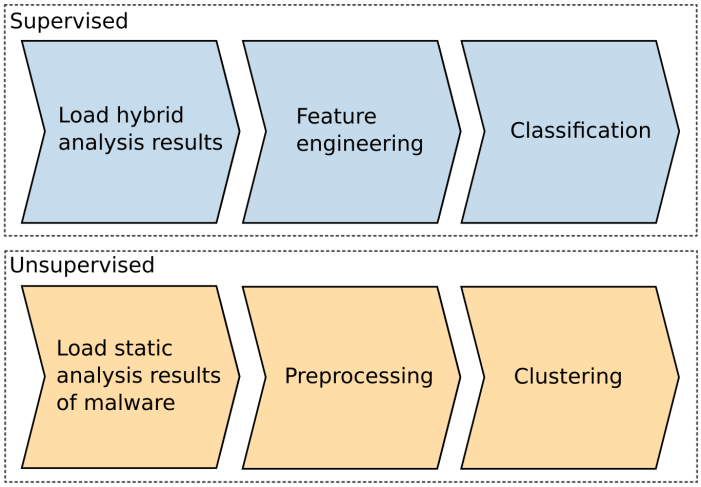

Figure 4.8: Conceptual steps of the supervised approach (in blue) and the unsupervised approach (in yellow)

In the supervised approach the first step is to load the results of the hybrid analysis from the database. Afterwards the feature engineering will be performed, which will be followed by the classification. The unsupervised approach will be executed alongside the main approach in order to detect the specific categorical belonging of the malware. In order to cluster malware into specific groups, the results of the static analysis of the malware will be used. Afterwards the preprocessing will be executed and will be followed by the clustering procedure. The next lines will outline the classification approach and proceed with the clustering procedure.
Both hybrid analyses use static results of benign and malicious apps. The class

assignment of an app is determined by the Virustotal report. Therefor the number of detected positives of the aggregation service will be used. The teacher noise of the model is reduced by using results of several virus scanner. Collected static (see figure 4.3) and dynamic information (see figure 4.6) will be transformed into categorical features for feature engineering. Additionally the amount of items within a category will be stored as a continuous feature. An example would be the permissions, where every permission is assigned to its category. The overall amount of used permissions will be stored as feature as well. Continuous data values will be used as continuous features, whereby categorical data will be transformed into categorical features using one-hot-encoding (see subsection 2.2.3).
Random forests, a bagging ensemble approach of decision trees, will be used as the classification algorithm. More details can be found in chapter 2.2.5. Random forests are used due to their simplicity and effectiveness. They also don't require feature preprocessing and prevent overfitting, which usually occurs when decision trees without further configuration are used. CART will be used as the underlying decision tree algorithm.
Besides an unsupervised approach will be used to gain insight into the types of the analyzed malware. The Virustotal service will provide the utilized malware terms. It is revealed by the virus scanners if a malware is successfully detected. The names will be split into strings and will be transformed into a matrix of tokens. TF-IDF is going to be used (see chapter 2.2.2) as a weighting approach for the used terms. K-means will be applied as a cluster algorithm afterwards. In order to receive the appropriate cluster size K the silhouette coefficient will be utilized. Both approaches are described in subsection 2.2.2. Further decisions due to evaluation of the classification and of the clustering approach can be found in detail in chapter 8.
For implementation the scikit-learn library will be used due to its rapid prototyping capabilities. It provides a wide choice of efficient machine learning algorithms and comes with an outstanding documentation. For realization the Python programming language in version 3 will be applied. For efficient data structures tools from the SciPy library will be utilized. Both libraries are described in subsection 2.3.2.

5 Data Retrieval

This chapter deals with the implementation of the first module *data retrieval*. Details and scripts for retrieving the required data collection are described in section 5.1, followed by the results in section 5.2.

5.1 Implementation

The platform Appsapks [1] is used to download benign Android applications. It contains several thousand Android applications. The website separates apps into different categories like games, messaging, productivity or weather. The browser is utilized to download the apps. The popular video player application VLC for example can be downloaded using the address `http://www.appsapk.com/downloading/latest/VLC%20for%20Android.apk`.
The `%20` is used as URL encoding for the *space* character. A bash script is utilized to perform APK downloads automatically. Its content is shown in code snippet 8. The most used commands are build-in *Command-line interface (CLI)* tools of the most Linux distributions. Their functionality is documented within the Linux man-pages project [2]. However the tools `curl` [3] and `unzip` [4] are not included. The category *all apps* of the repository is used. It provides access to apps from various categories. This script is designed in a single thread manner, to prevent blocking due to overmuch requests. At the beginning the method `crawl_repository` is called (line 39). It enumerates through the pages 1 to 200, which denote the possible page indices within the category *all apps*. The content of every page is received by calling `receive_content` (line 30). Primarily this function uses the CLI command `curl`

[1] http://www.appsapk.com/ (Last access on 06.05.2018)
[2] https://www.kernel.org/doc/man-pages/ (Last access on 05.05.2018)
[3] https://curl.haxx.se/ (Last access on 05.05.2018)
[4] https://www.info-zip.org/pub/infozip/ (Last access on 05.05.2018)

with the silent option `-s`. The response is matched against the `$app_pattern` using `egrep`. The command `cut` cuts out the app name from the fourth field. Option `-d` sets the delimiter. The access field is specified with `-f`. The resulting list is filtered with `uniq` in order to retrieve an unique set of app names (line 12/13). This set is stored using the variable `content`. Every app entry from `content` is concatenated with the `BASIC_URL` and used as an argument of function `receive_apk_url`. The commands `curl`, `egrep` and `cut` are utilized to find the APK URL. `Sed` replaces white spaces with `%20` using the common code point U+0020. Finally the function `download_apk` (line 21) checks if `apk_url` has a tailing APK file ending. If yes, `wget` downloads the Android application into the current directory.

```
#!/bin/bash

readonly BASIC_URL='http://www.appsapk.com/'
readonly ROOT_URL=$BASIC_URL'android/all-apps/page/'
readonly APP_PATTERN='<a href="http://www.appsapk.com/.*/" title='
readonly DOWNLOAD_PATTERN='<p><a class="download" rel="nofollow"
                           href=".*">Download APK</a></p>'
readonly WHITESPACE_REPLACE='s/ /%20/g'
readonly SLASH='/'

receive_content() {
    content=$(curl -s $ROOT_URL$1$SLASH | egrep -i "$APP_PATTERN"
              | cut -d/ -f4 | uniq)
}

receive_apk_url() {
    apk_url=$(curl -s $1 | egrep -i "$DOWNLOAD_PATTERN"
              | cut -d\" -f6 | sed -e "$WHITESPACE_REPLACE")
}

download_apk() {
    if [[ $1 == *apk ]]; then
        wget $1
    fi
}

crawl_repository() {
    for site in {1..200}
    do
        receive_content $site
        for app_entry in $content
        do
            receive_apk_url $BASIC_URL$app_entry$SLASH
            download_apk $apk_url
        done
    done
}

crawl_repository
```

Code snippet 8: Download script for the Appsapk repository

The *Virusshare* repository is used to download Android malware. Since the special request section does't contain malware of 2017, a friendly request for such a collection was sent to the administrator.

```
#!/bin/bash

readonly ZIP_PATTERN="(Zip)|(Java) archive data"
readonly MANIFEST="AndroidManifest.xml"
readonly PWD=$(pwd)"/"
readonly PWD_CONTENT=$PWD"*"
readonly OUTPUT=$PWD"filtered_malware/"

check_for_archive_file() {
    if [[ $1 =~ $ZIP_PATTERN ]]; then
        malware_file=$(echo $1 | cut -d: -f1)
        move_if_android_malware $malware_file
    fi
}

move_if_android_malware() {
    is_android_app=$(unzip -l $1 | grep $MANIFEST | wc -l)
    if [[ $is_android_app -eq 1 ]]; then
         mv $1 $OUTPUT
    fi
}

filter_android_apps() {
    mkdir $OUTPUT
    for malware in $PWD_CONTENT
    do
      malware_type=$(file $malware)
      check_for_archive_file "$malware_type"
    done
}

filter_android_apps
```

Code snippet 9: Script for filtering Virusshare downloads

Since today it was neither fulfilled nor answered. Using the direct downloads of the repository was the only left alternative. In irregular intervals new discovered malware

samples are archived and released as zip files. They contain different types of malware and often several gigabytes of data. Within this thesis the following zip files of 2017 are used: `Virusshare_00276.zip` (24.02.2017) until `Virusshare_00096.zip` (26.08.2017). This particular choice roughly covers six months of 2017.
The Android malware of a *Virusshare* zip file is filtered out with the help of a script shown in code snippet 9. At the beginning the method `filter_android_apps` is invoked (line 32). The method loops over the current directory content and determines the file type. The type is passed over to `check_for_archive_file` (see line 28), which checks if it matches the `$ZIP_PATTERN`. If true, the file name is cut using `cut` and the delimiter `":"`. The result is stored in the variable `$malware_file` and passed to the method `move_if_android_malware` (see line 12). The command `unzip -l` is used to check for an Android application. In detail `grep` and `wc -l` are used to count the resulting lines for `AndroidManifest.xml`. The amount is stored in `is_android_app`. Afterwards the file is moved to the path `$OUTPUT` (line 19), if the amount equals one.
The following outdated malware collections were downloaded as well. They could be useful for the future work. These collections are provided in a torrent format:

1. Request for all 'Android' matches until 09.05.2013.
2. Request for all 'Android APK' files 2013-05-06 to 2014-03-24.

5.2 Results

This section presents the results of collected Android apps. First the Appsapk repository was requested and 5243 recent benign apps were downloaded and stored. They utilize a disk size of 33.7 GiB in total.
The direct downloads were filtered in order to build a collection of malware of 2017. In total 4160 malware apps were extracted. The data collection of the 2017 malware is about 44.7 GiB large.
Besides the core collection older malware were downloaded from *Virusshare*. The first torrent download covers Android malware from the beginning of 2012 until mid of 2013. In total 11080 apps with an overall size of 7.2 GiB were downloaded. The second torrent download covers malware from mid of 2013 until mid of 2014. This collection contains about 24317 applications with an overall size of 50.7 GiB.

Both collections are not used within this thesis, but suitable for possible future applications. The gap between malware for the years 2015 and 2016 can be closed in the future.

6 Hybrid Analysis

Within this chapter the implementation of the prototype for the hybrid app analysis will be elaborated. Therefor the implementation is divided into several sections. First general information is depicted in section 6.1, followed by details of the static and dynamic approach (see section 6.2 and 6.3). The implementation of the database connection is described in section 6.4. The final section 6.5 will encompass the results of the hybrid analysis. For the purpose of imparting the used structure, class diagrams in the fashion of UML are utilized. To keep track of the structure some attributes and operations were omitted.

6.1 General

The prototype is implemented in Java using Oracle JDK 9 and Jetbrains Intellij IDE on an Arch Linux environment. Both, the static and dynamic implementation, are integrated into the same Java project. The prototype is called *AndroML* and is created as Gradle project. The implementation mostly follows *Test-driven development (TDD)*. The predefined package structure `src` is used for productivity and `test` is utilized for test code. Dependencies of the application are defined within the `build.gradle` file. The general dependencies can be found in the following list:

- JUnit 4.12 test framework [1]
- Mockito Core 2.8.47 as mocking library for unit tests [2]
- SLF4J logging library [3]
- Apache Commons Configuration 2.1.1 [4]

[1] https://junit.org/junit4/ (Online on 01.05.2018)
[2] http://site.mockito.org/ (Online on 01.05.2018)
[3] https://www.slf4j.org/ (Online on 01.05.2018)
[4] https://www.slf4j.org/ (Online on 01.05.2018)

AndroML uses the following package structure for the production code:

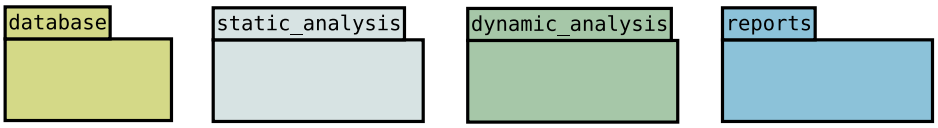

Figure 6.1: Package overview

A more detailed explanation of these modules can be found within the next sections. The test code has the identical structure, but includes the additional package *learning_tests*. This package contains tests to investigate the functionality of new included libraries within the project [Mar08]. They are not described within this thesis. Within this prototype several classes are included into a package. It will be referred to it as a component. These components try to follow close *component cohesion*. Therefor three principles stay in focus. The *Reuse/Release Principle (REP)* states that classes in a component must belong to a cohesive group, should make sense and should be releasable together. The *Common Closure Principle (CCP)* states that classes in a component should change for same reasons and at the same time. The *Common Reuse Principle (CRP)* states in a simplified way, that components should not depend on things they don‘t need [Mar17].
To follow the *Dependency Inversion Principle (DIP)* simple constructor injection as a dependency injection technique is used. This will also ensure convenient testing using the Mockito library. Therefor the use of a Singleton pattern is consciously skipped. It is commonly known as hard to test and introduces problems within a concurrent architecture. However it is not required to consider all these principles and best practices permanently. Since the prototype will not contain an official API, it is not recommended to use Javadocs due to distraction [Mar08]. It is more recommended to apply clean coding principles instead [Mar08]. Javadocs are often bloated, prone to duplicate information and support inconsistencies during code refactoring. This recommendation will be followed. The prototype is designed to run the static and dynamic analysis independently from each other.

The class `AndroML` contains the main method of the application. This class initially creates a configuration instance from class `AndroMLConfig` and a database instance from class `ElasticAdapter` (see section 6.4). These are injected into an instance of `StaticAnalysis` (see section 6.2) and `DynamicAnalysis` (see section 6.3). The `AndroMLConfig` class uses Apache Commons Configurations to read in a

configuration file and provides specified settings through appropriate getter methods.

6.2 Static Analysis

The static analysis is located within the module `static_analysis`. The class diagram in figure 6.2 gives an overview about used classes, packages and dependencies. Details regarding the implemented analyzers are omitted, but depicted and described in the subsections 6.2.1, 6.2.2 and 6.2.3.

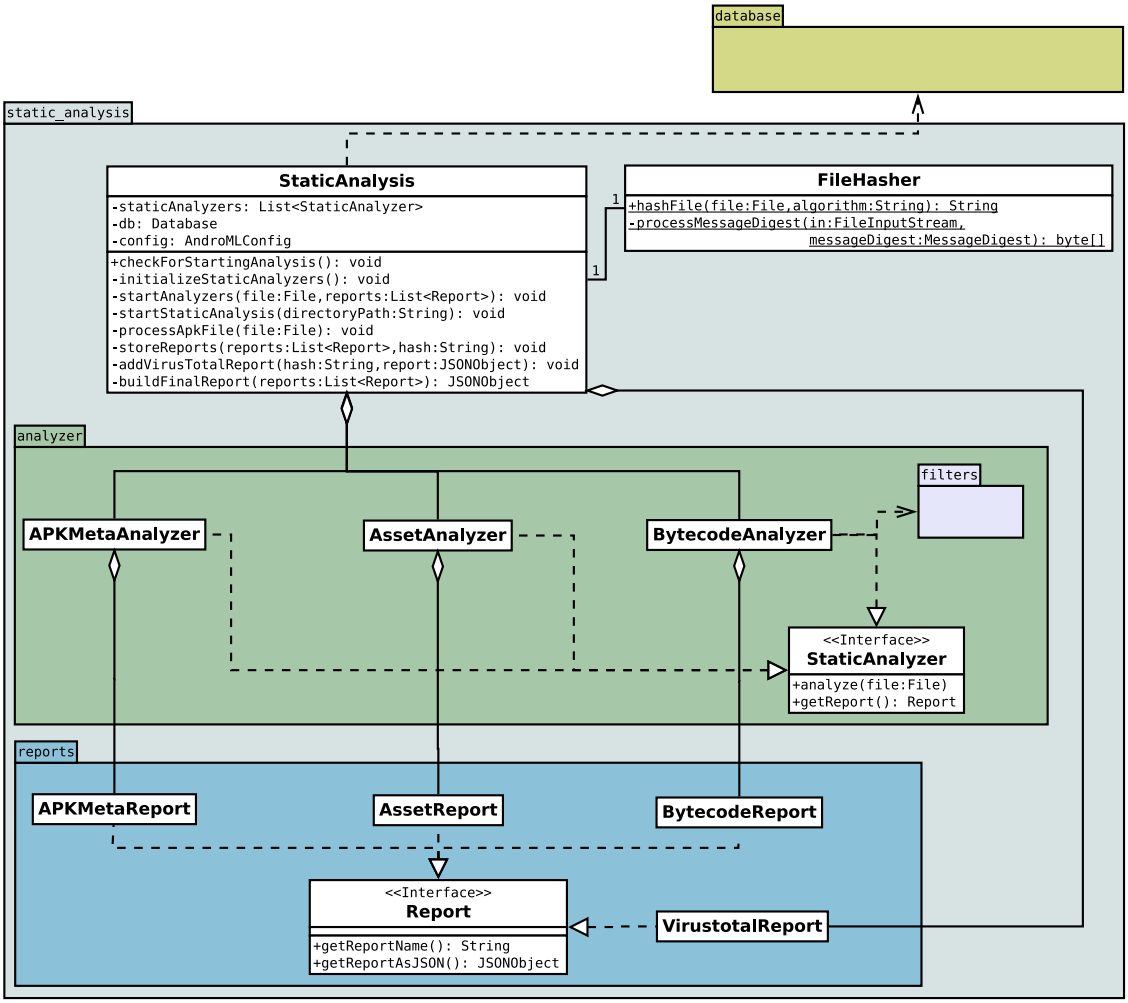

Figure 6.2: Class diagram of the static analysis

The main entry point is the class `StaticAnalysis`. It contains dependencies to the external module `database`. Within the package `static_analysis` the class

`FileHasher` is utilized and the inner packages `analyzer` and `reports` are accessed. They contain logic for the analysis processes and the collected analysis results. The interface `StaticAnalyzer` specifies general functionality for analyzers. Therefor the method `analyze()` and `getReport()` are defined. Every analyzer takes a file and analyze it. The results are stored in the appropriate report. `StaticAnalyzer` is implemented by three analyzers: `ApkMetaAnalyzer`, `AssetAnalyzer` and `ByteCodeAnalyzer`. They are described in the upcoming subsections 6.2.1, 6.2.2 and 6.2.3. The `BytecodeAnalyzer` has a dependency to the `filters` package, which includes logic for filtering the bytecode. For every analyzer class an appropriate report class is implemented: `ApkMetaReport`, `AssetReport`, `ByteCodeReport`. They are located in the package `reports` and implement the `Report` interface. This interface provides two getter methods in order to retrieve the report name and the report in JSON format. In order to store the Virustotal responses the `VirusTotalReport` is used. In summary both interfaces fulfill the *Open/Closed Principle (OCP)* [Mar17].
To run a static analysis the `StaticAnalysis` instance checks the global configuration by invoking `checkForStartingAnalysis()`. If activated, it calls `initializeStaticAnalyzers()` to add the three described analyzers to the list `staticAnalyzers`. Afterwards `startStaticAnalysis()` is called. This procedure recursively iterates through a given directory and their subdirectories. If an APK is found the method `processApkFile()` is invoked. A file object is generated and passed as a parameter. This method generates a SHA256 hash of the APK file, using `FileHasher`. It checks if a report already exists. If not, the method `startAnalyzers()` with the APK file and `storeReports()` are invoked. The method `startAnalyzers()` works on the `StaticAnalyzer` interface and calls `analyze()` and `getReport()` in order to store the resulting report. This enables convenient extensibility which realizes OCP. Finally `storeReports()` collects reports of the analyzers. Invoking `addVirusTotalReport()` results in building a `VirusTotalReport` object. The method `buildFinalReport()` merges the reports into a final report, which is stored into the database with the help of the `database` package. More information about data storage can be found in section 6.4. The upcoming subsection describes the implementation of the analyzers in more detail.

6.2.1 Class `APKMetaAnalyzer`

This subsection outlines details about the `APKMetaAnalyzer` class with the help of a class diagram pictured in figure 6.3.

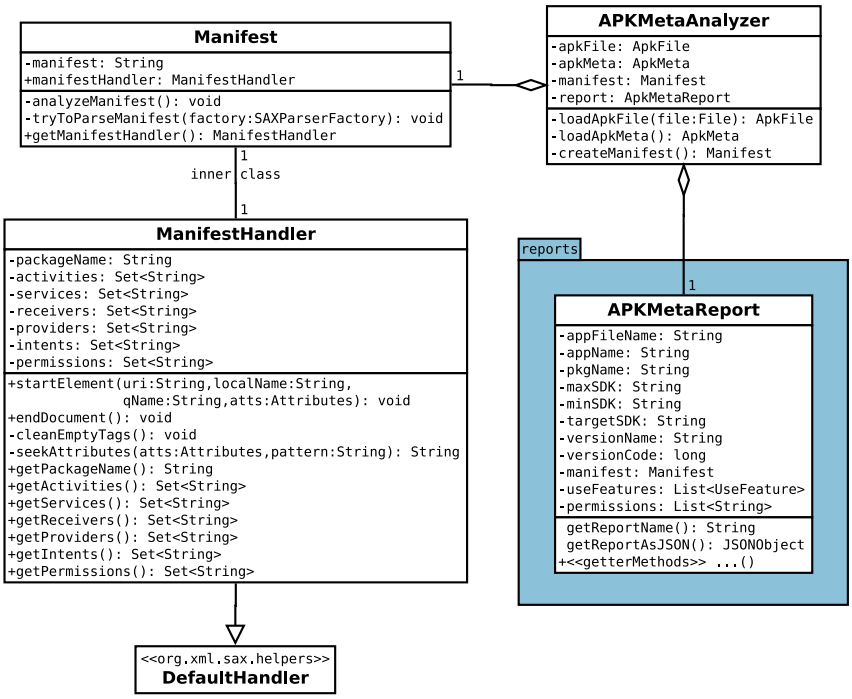

Figure 6.3: Class diagram for `ApkMetaAnalyzer` and periphery

This class extracts data from the `AndroidManifest.xml` using the *APKParser* library (see section 2.1.3). It makes parts of an APK file programmatically available as an `ApkFile` instance. This instance is created by calling `loadApkFile()` and stored in the class attribute `apkFile`. The method `loadApkMeta()` retrieves an instance of `ApkMeta` by invoking `apkFile.getApkMeta()`. This class provides general information about the app. It is stored in the class attribute `apkMeta`. In order to extract and binary-translate the `AndroidManifest.xml` the *APKParser* method `apkFile.getManifestXml()` is invoked. The resulting string is handed over to an instance of `Manifest`, which is created by the method `createManifest()`. This class is depicted in figure 6.3 and contains an instance of `ManifestHandler`. This handler class extends the `DefaultHandler` class and is handed over to a `SAXParser` instance alongside with the manifest string. This is realized by calling `analyzeManifest()`, which contains the exception handling. Afterwards `tryToParseManifest()` is invoked, which creates the `SAXParser` using a `SAXParserFactory` and call `parse()`.

Within the handler the method `startElement()` is called for every XML element to filter for specific tags. It detects the tags: `manifest`, `activity`, `service`, `receiver`, `provider`, `action` and `permission`. The `action` tag occurs within the `intent-filter` tag and is used to determine the intent names. More information can be found within the official documentation [5]. The mandatory attributes of the tags are extracted by invoking `seekAttributes()` and are stored within the appropriate `Set<String>` class attributes. A `Set` is used to prevent redundant entries. The procedure `cleanEmptyTag()` removes unwanted empty entries within the data structures.
Finally an instance of `ApkMetaReport` is created. It takes the `Manifest` and `ApkMeta` instances in order to extract the required data. This data is stored in the fields of the report as shown in figure 6.3.

6.2.2 Class `ByteCodeAnalyzer`

This subsection describes the analysis of Android bytecode using the `ByteCodeAnalyzer` class. Details about the implementation and interaction with corresponding classes and data structures are depicted in figure 6.4.

[5] https://developer.android.com/guide/topics/manifest/manifest-intro (Last access on 10.05.2018)

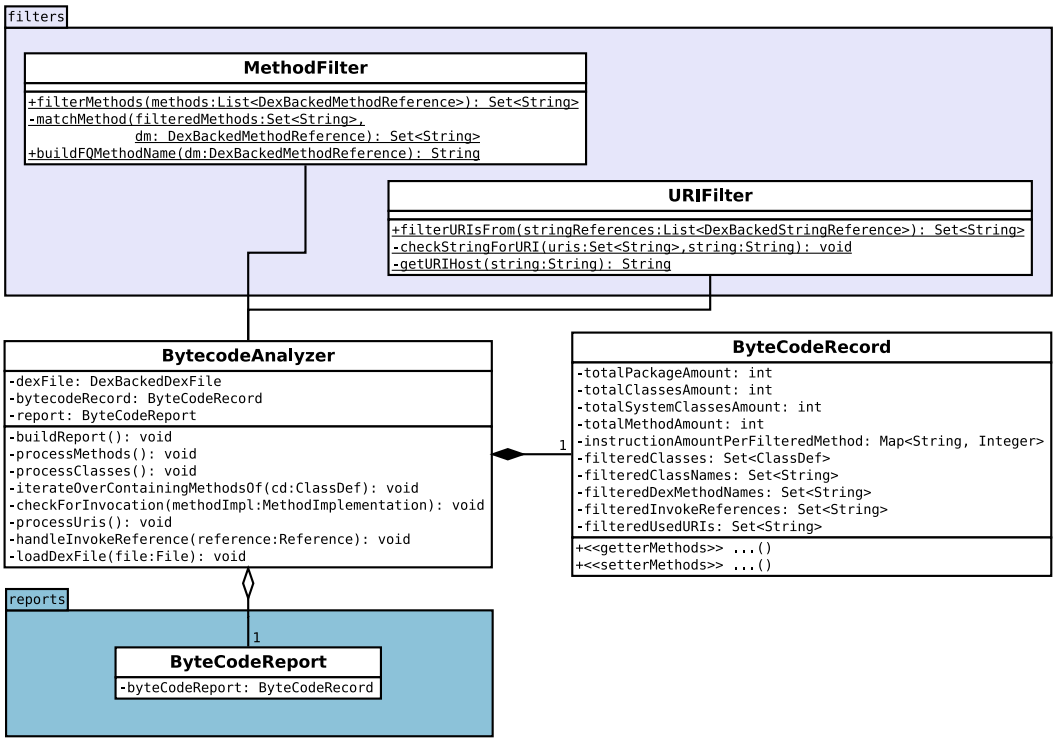

Figure 6.4: Class diagram for `ByteCodeAnalyzer` and periphery

This class utilizes the `DexLib2` library (see subsection 2.1.3) in order to access the Android bytecode. The analyzer calls the overwritten method `analyze()` at the beginning. An instance of `ByteCodeRecord` is used to store the data. It is accessible as a class attribute within `ByteCodeAnalyzer`. In order to access bytecode the `classes.dex` is processed. Some files use the multidex approach (see subsection 2.1.2), which isn't implemented currently. Around 9% of the collected APKs use more than a single Dex file. The majority outsources only a small amount of methods to a secondary Dex file. However the dynamic analysis is able to cover the missing code. The feature is covered in section 9.1.
The method `loadDexFile()` within the analyzer uses the `DexFileFactory` to load a `DexBackedDexFile` into the `dexFile` class attribute. Next the bytecode classes and methods are ready for processing.
The method `processClasses()` collects class data and iterates over all classes by using `dexFile.getClasses()`. Within this loop the method information is collected as well, since it would be inefficient to process the classes several times. Thus for every processed class `iterateOverContainingMethodsOf()` is called. With `checkForInvocation()` every instruction within the methods is checked for an invocation. If an invocation is detected, `handleInvokeReference()` is called.

It stores the detected name into `filteredInvokeReferences` of the attribute `byteCodeRecord`. During the iteration Android API classes are removed. The following variables are attributes of the `ByteCodeRecord` class. Class names are added to the class attribute `filteredClassNames`. The actual `ClassDefItems` of *DexLib2* are stored within `filteredClasses`. The amount of total classes and total system classes are saved in appropriate fields.
Invoking `processMethods()` collects the `totalMethodAmount` and `filteredDexMethodNames`. To do so the static method `filterMethods` of `MethodFilter` is invoked. This method calls `matchMethod` to filter the full-qualified method name against a pattern. For building the fully-qualified name the method `buildFQMethodName()` is used. Currently no filtering is performed.
The *Uniform Resource Locators (URIs)* are collected by calling `processURIs()`. This causes the usage of `URIFilter`. The static method `filterURIsFrom()` is called. It iterates over the string references, which are handed over as a parameter. For every String the method `checkStringForURI()` is called. The method body contains a `Matcher` to match URI patterns using regular expressions. Different patterns are used for recognizing URLs or e-mail addresses.
In the end the `byteCodeRecord` is handed over to an instance of `ByteCodeReport` by calling `buildReport()`. The `ByteCodeReport` uses the `ByteCodeRecord` instance to build a JSON object from the collected data.

6.2.3 Class `AssetAnalyzer`

This subsection deals with the analysis of the asset files within an APK. The class `AssetAnalyzer` is implemented and details are depicted in figure 6.5.

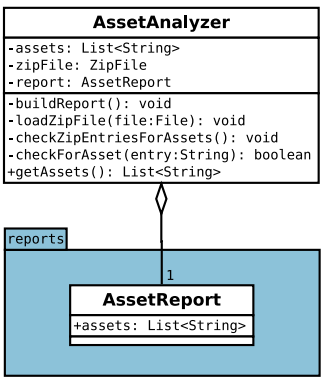

Figure 6.5: Class diagram for `AssetAnalyzer` and periphery

This class calls `loadZipFile()` with the `File` instance of the APK file. In order to iterate over all entries in the Zip file the method `checkZipEntriesForAssets()` is invoked. For each entry `checkForAsset()` is called, which matches against files in the `assets` and `lib` folder. If a match is found the name of the entry is added to the `assets` list. `buildReport()` is called and the `assets` list is passed as a parameter. The `AssetReport` takes the list of assets and transforms it to a JSON object using the term *assets* as key.

6.3 Dynamic Analysis

This section deals with the implementation of the dynamic analysis approach. In order to visually support the realization a class and package overview is shown in figure 6.6.

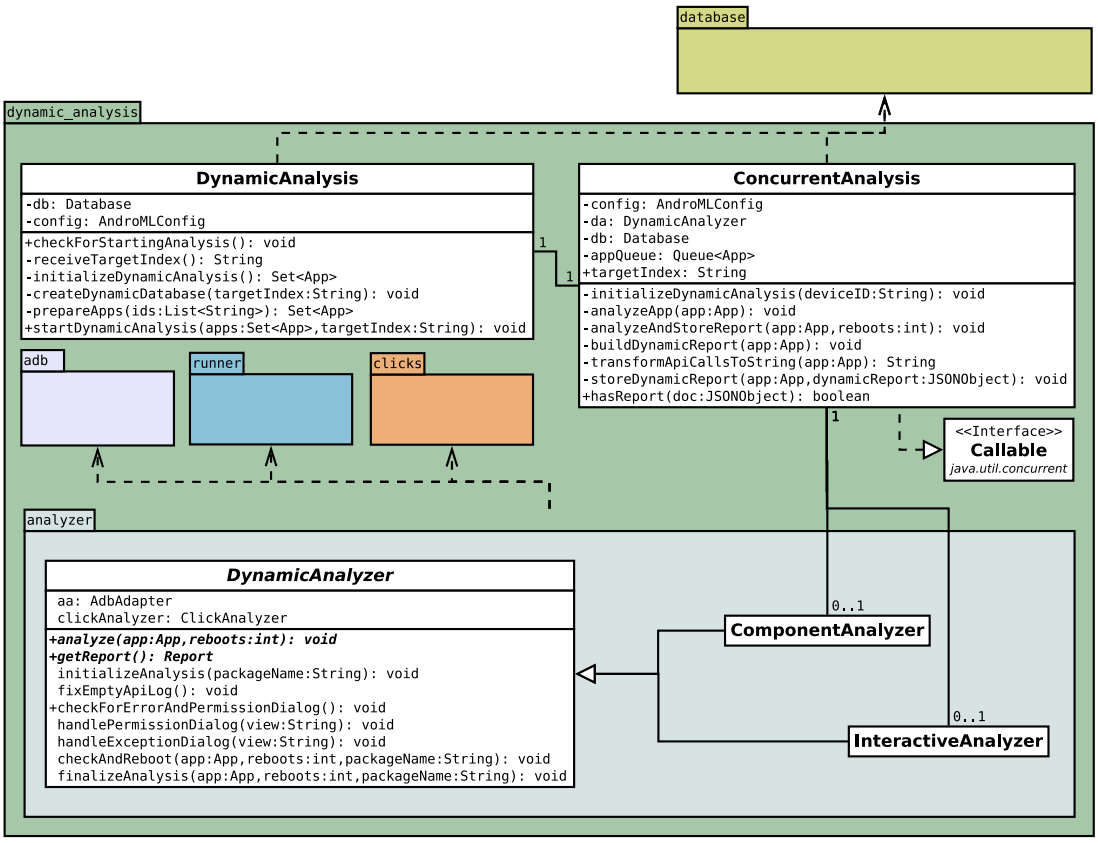

Figure 6.6: Class diagram for the dynamic analysis

The main entry point for the dynamic analysis is the class `DynamicAnalysis`. The class `ConcurrentAnalysis` implements the `Callable` interface and an instance

performs the dynamic analysis within a separate thread. Details can be found in subsection 6.3.2. The module has a dependency towards the `database` package. The main package `dynamic_analysis` contains four modules: `adb`, `runner`, `clicks` and `analyzer`.
The module `adb` is outlined in subsection 6.3.1. The dynamic analysis provides the component analysis using the class `ComponentAnalyzer` (see subsection 6.3.4) and the interactive analysis using the class `InteractiveAnalyzer` (see subsection 6.3.5). Common functionality of both is implemented in an abstract class `DynamicAnalyzer` described in subsection 6.3.3. The device preparation and the setup of the environment is outlined in subsection 6.3.6.
The dynamic analysis is started by invoking the method `checkForStartingAnalysis()`. Afterwards `initializeDynamicAnalysis()` is called. This method receives the database ids and invokes the procedure `prepareApps()`. It receives stored static analysis information of an app. Both is realized by using the `ElasticAdapter` within the database module. The target index for storing the dynamic analysis results is determined by using the configuration file. The index is stored in the class attribute `targetIndex` within `ConcurrentAnalysis`.
An app is represented by an instance of `App` using the collected data. They are appended to a `List<App>` and returned to `checkForStartingAnalysis()`. The target index is retrieved by invoking `receiveTargetIndex()`. If necessary, the appropriate database is created by calling `createDynamicDatabase()`.
Afterwards the method `startDynamicAnalysis()` is called. It launches the analysis. Therefor a new `ExecutorService` with a fixed thread pool is created. The size of the pool depends on the amount of connected devices. Within this thesis three devices are used. The collected apps in the `List<App>` are added to a `ConcurrentLinkedQueue` named `appQueue`. For every device an instance of `ConcurrentAnalysis` is created. As parameters the `appQueue`, device identifier and `targetIndex` are used. Finally all worker threads are started by invoking `invokeAll()` on the `ExecutorService`.

6.3.1 Module `adb`

The module uses the Android Debug Bridge to control the devices with specific terminal commands. The `adb` package is composed of three classes which are depicted in the class diagram in figure 6.7.

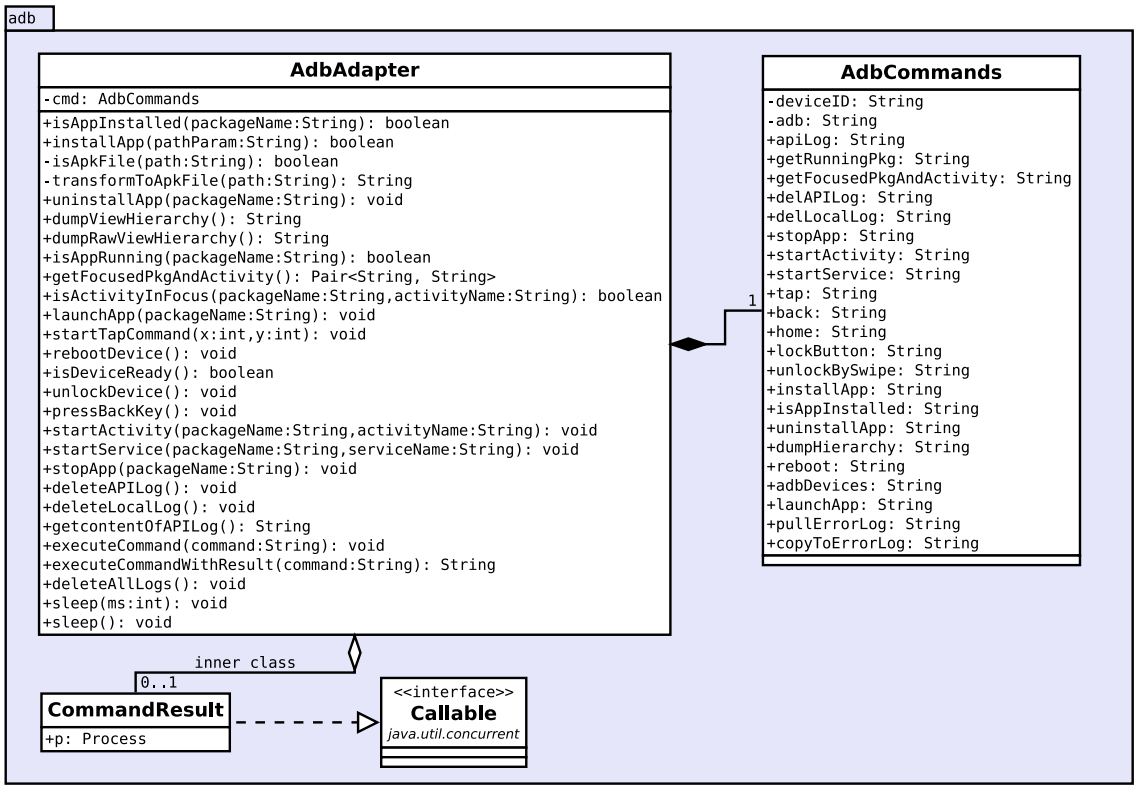

Figure 6.7: Content of the `adb` package

The class `AdbAdapter` is the main entry point for every interaction command. Commands that are used in the analysis are enumerated as public methods in the class diagram 6.7. This class is able to install and uninstall apps on the device. It can also launch components like activities or services and can check if a specific app is in the main focus of the device. It is also capable of rebooting the device and checking dynamic analysis logs. Moreover `AdbAdapter` is able to dump the recent view and components and to perform various interactions such as clicking, pressing the back key or unlocking the device. The required methods are listed in figure 6.7. To illustrate an example of the usual functionality the method `unlockDevice()` is shown in code snippet 10.

```
1  public void unlockDevice() {
2      String view = dumpRawViewHierarchy();
3      if (view.contains(LOCK_SCREEN_IDENTIFIER)) {
4          try {
5              executeCommand(cmd.powerButton);
6              executeCommand(cmd.unlockBySwipe);
7          } catch (IOException e) {
8              LOG.error("Unlocking of device failed.", e);
9          }
10         sleep(3000);
11     }
12 }
```

Code snippet 10: Program code of `unlockDevice()` within the class `AdbAdapter`

First `unlockDevice()` invokes `dumpRawViewHierarchy()` which uses the `dump` command to receive the recent hierarchy view. The device id is used to access the device. If the view contains a specific lockscreen identifier two `executeCommand()` methods will be invoked. The `executeCommand()` uses the Java runtime and calls its `exec()` method to execute the command. The first invocation of `executeCommand` receives the parameter `cmd.powerButton` and the second invocation `cmd.unlockBySwipe`. The variable `cmd` is an instance of the class `AdbCommands`. This class contains every used command and is created by `AdbAdapter`. Since the commands need to be specific for each device the device id is used to construct the commands within the `AdbCommands`. The command `cmd.powerButton` holds the command for pressing the power button of the device, whereas `cmd.unlockBySwipe` stores the following command: `adb -s <device_id> shell input touchscreen swipe 930 880 930 380`. This runs `adb` for a specified device and uses a shell command named `input touchscreen` to perform a swipe movement. The first two coordinates define the start point $(x1, y1)$, followed by the target point $(x2, y2)$. Since the y-coordinate is reduced to 380 pixel, an upside-down movement is performed which unlocks the device. The subsequent command `sleep()` lets the application sleep for 3000 milliseconds. The duration of the sleep varies for each type of command. An app installation needs to determine the success or failure of the process dynamically. This requires a dynamic sleep time. Other commands work in a similar way. They use a command from `AdbCommands` and execute it. Sometimes instead of `executeCommand()` the method `executeCommandWithResult()` is invoked. The method actually returns

the result of a command. This is required for receiving the view hierarchy. For `executeCommandWithResult()` an additional `Callable` named `CommandResult` is used, which executes the commands in a separate thread. After finishing it outputs the results. It is valuable to have structured and clean tests for the `AdbAdapter`. They help to ensure that the interaction with the device is fully functional.

6.3.2 Class `ConcurrentAnalysis`

The class `ConcurrentAnalysis` controls the dynamic analysis for a single device and runs as an individual thread. For that the interface `Callable` from `java.util.concurrent` is implemented. Three instances of this class are running in parallel, due to three connected devices. If an instance is created it uses the `appQueue` and `targetIndex`. In order to prevent race conditions `ConcurrentAnalysis` creates own instances of `AndroML` and `ElasticAdapter`. Afterwards the type of dynamic analysis is determined by calling `initializeDynamicAnalysis()`. Within this method an instance of `InteractiveAnalyzer` or `ComponentAnalyzer` is created and then stored as `DynamicAnalyzer`.
The main logic is triggered by running `call()`, which is handled by the `ExecutorService`. The main task checks `appQueue` for remaining apps. If there are remaining apps, a single app is polled out of the queue and `analyzeApp()` is called. Since `appQueue` is a `ConcurrentLinkedQueue` it provides thread safety. If the queue empties out, the thread will finish. In this way the capacity of the connected devices are utilized.

6.3.3 Abstract Class `DynamicAnalyzer`

This abstract class is extended by the classes `ComponentAnalyzer` and `InteractiveAnalyzer` and contains common functionality for both implementations. It defines two abstract methods `analyze()` and `getReport()`, which are implemented by the two classes. The class `DynamicAnalyzer` is depicted in figure 6.6.
The method `initializeAnalysis()` receives a package name and performs the initialization process described in figure 6.8. An instance of the `AdbAdapter` is created and stored in the variable `aa`.

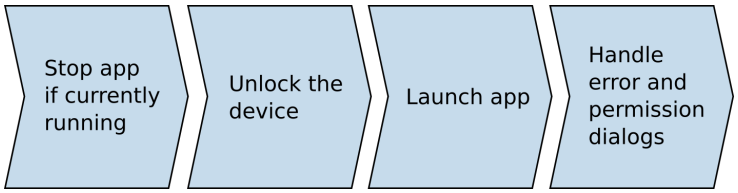

Figure 6.8: Common initialization process of a dynamic analysis

First `aa` is used to stop the app and if necessary to also unlock the device afterwards. The app is started and possible error or permissions dialogs are handled. Therefor the method `checkForErrorAndPermissionDialog` invokes three methods: first `aa.dumpRawViewHierarchy()` to receive the actual view, then `handlePermissionDialog()` and `handleExceptionDialog()`. The method `handlePermissionDialog()` takes the view as parameter and checks via `UIPatterns.isPermissionDialog()` if a permission dialog is present. This is performed within a `while` loop. Inside this loop `Interactions.allowPermission()` is invoked in order to click the allow button (see more details in section 6.3.5). Afterwards a new view hierarchy is dumped and checked within the `while` loop. If every single permission is allowed, the method terminates. The method `handleExceptionDialog` follows the identical logic, but checking for exceptions and clicking the OK button.
In order to let the device react to the start of components the sleep method is invoked. The procedure `finalizeAnalysis()` performs the finalization steps for the dynamic analysis as illustrated in figure 6.9.

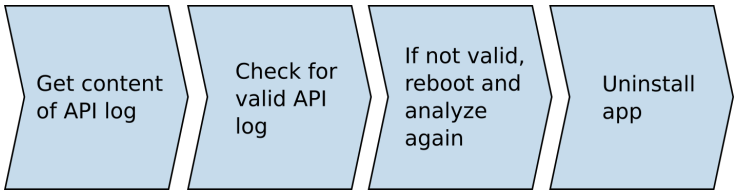

Figure 6.9: Common finalization process of a dynamic analysis

The method `finalizeAnalysis()` pulls the dynamic analysis log from the device by invoking `aa.getContentOfAPILog()`. Afterwards `checkAndReboot()` is called to check if the API log is empty. In this case `fixEmptyAPILog()` is called, which reboots the device and restart the analysis. To prevent endless rebooting, only a single reboot is performed before continuing with the next application. Afterwards the app is uninstalled. Finally `checkForErrorAndPermissionDialog` is called to clean up the dialogs.

6.3.4 Class `ComponentAnalyzer`

This class controls the dynamic analysis by starting activities using an iterative approach. An overview is depicted in diagram 6.10.

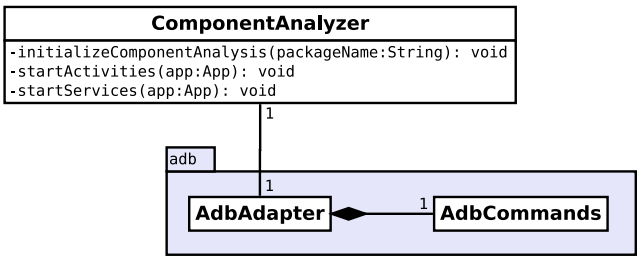

Figure 6.10: Class diagram for `ComponentAnalyzer` and periphery

An instance of this class first calls `initializeComponentAnalysis()` which executes `initializeDynamicAnalysis` of the class `DynamicAnalysis`. Afterwards `startActivities()` is invoked. It uses `app.getActivities()` to iterate through all collected activities. An instance of `AdbAdapter` named `aa` is also used. The methods `aa.startActivity()`, `aa.sleep()` and `checkForErrorAndPermissionDialog()` are called for every activity. The method `startServices()` iterates through the services of an `App` instance. For each service `aa.startService()` is called, followed by `checkForErrorAndPermissionDialog`.

6.3.5 Class `InteractiveAnalyzer`

A detailed class diagram for the interactive approach can be found in figure 6.11. The class `InteractiveAnalyzer` is the main entry point, which uses the packages `clicks` and `runner` alongside the package `adb`. Package `clicks` is responsible for extracting information from the hierarchy view. The package `runner` interacts with the device through clicks. In order to understand the analysis approach both packages will be explained in more detail.

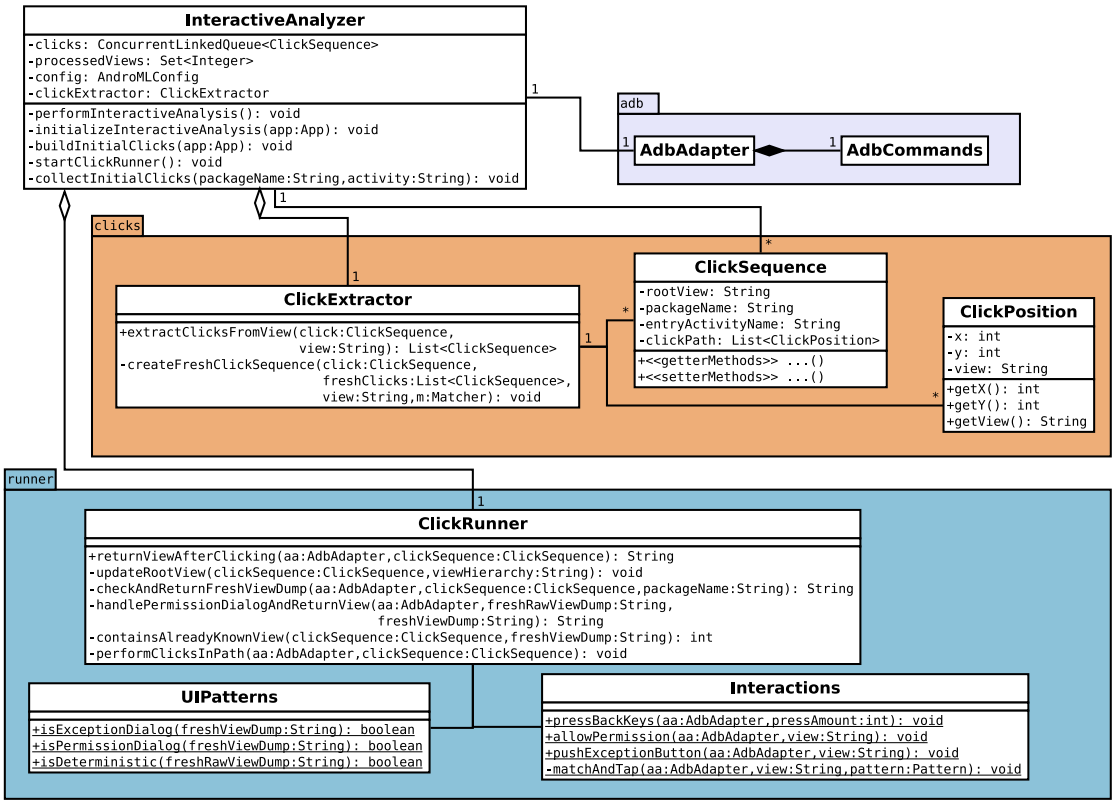

Figure 6.11: Class diagram for `InteractiveAnalyzer` and periphery

The main entry point of the `clicks` package is the class `ClickExtractor` containing the method `extractClicksFromView()`. The method receives a view and a `ClickSequence` named `clicks`. A `ClickSequence` is a data structure containing a chain of performed clicks. It is constructed by passing the following information: the root view, the name of the entry activity and the package name of the app. It is important to identify the click routines among themselves and to prevent circles in the sequence. Also a list of `ClickPosition` instances is handed over, which is named `clickPath`. If a click at the end of a click sequence opens an unknown UI, the root view stores the resulting hierarchy. The class `ClickPosition` encompasses (x, y) coordinates of a click and the view as `String`. As a result a `ClickSequence` represents several clicks on the UI until a specific depth, given by the size of the `clickPath`.

The function `extractClicksFromView` processes every line of the given view and extracts the position. This is handed over to `createFreshClickSequence()`, which creates new instances of `ClickPosition`. These are added to the provided `ClickSequence clicks`. As a result `List<ClickSequence> freshClicks`

is returned, including a `ClickSequence` for every found click in the view hierarchy.

The main entry class of the package `runner` is the class `ClickRunner` containing the method `returnViewAfterClicking()`. This method takes a `ClickSequence`, clicks the stored positions and returns the resulting view hierarchy. Therefor it first checks if the app is still running, performs the clicks and checks again. If a unknown user interface is detected, its view hierarchy is dumped and stored as the new root view. The interactive analysis follows the depicted program flow as illustrated in figure 6.12.

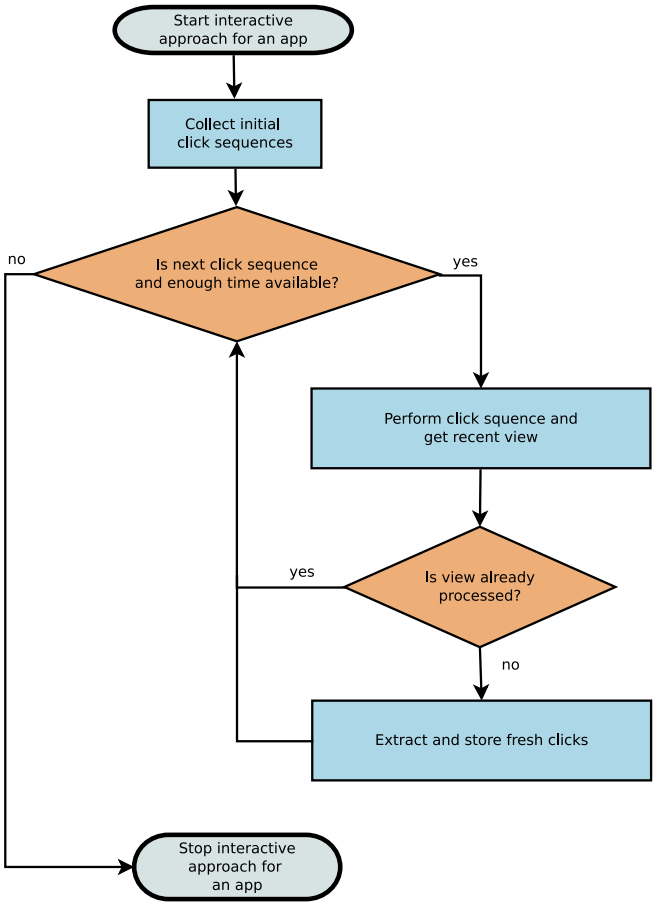

Figure 6.12: The interactive analysis program flow for an app

The `InteractiveAnalyzer` primarily operates on `clicks` a `ConcurrentLinkedQueue<ClickSequence>`. It contains the click sequences, which are sequentially processed as depicted in diagram 6.12. Also the following instances `processedViews` a `Set<Integer>`, a `ClickExtractor` and a `ClickRunner` are used. The data structure `clicks` records every discovered click within an app. It is able

to contain several `ClickPosition` instances. The data structure `processedViews` contains hash sums of already processed views. It is used in order to process new UI components within the app only.
The main starting point of `InteractiveAnalyzer` is the overwritten `analyze()` method. At the beginning the desired app is installed. Afterwards `initializeInteractiveAnalysis()` is invoked. This procedure calls `initializeAnalysis()` from the abstract class `DynamicAnalysis` followed by `buildInitialClicks()`. The logic uses the main entry activity component and checks if it matches the stated package and activity name. If so, `collectInitialClicks()` is called. The method `clickRunner.returnViewAfterClicking()` is invoked to receive the first initial view. The hash code of this view is stored in `processedViews`. It builds an initial list of `ClickSequence` by invoking `clickExtractor.extractClicksFromView()`. This list is added to the data structure `clicks`.
The next called method within `analyze()` is `performInteractiveAnalysis()`. As long as `clicks` is not empty and the time threshold is not reached, it calls `startClickRunner()`. This procedure is the main logic within this class. It polls a click sequence from `clicks`, performs the clicks by using `clickRunner.returnViewAfterClicking()` and checks if the reached UI is a new one. If so, it extracts the unknown clicks of the UI and adds them to the queue `clicks`. The process follows the content of figure 6.12. After every click is processed or if the time threshold is reached, the analysis of the app terminates.

6.3.6 Environment- and Device Preparation

This section describes how the analysis environment and the devices were prepared for the dynamic analysis. The first part starts by describing the setup, followed by the preparation of the devices. An overview of the setup is depicted in figure 6.13.

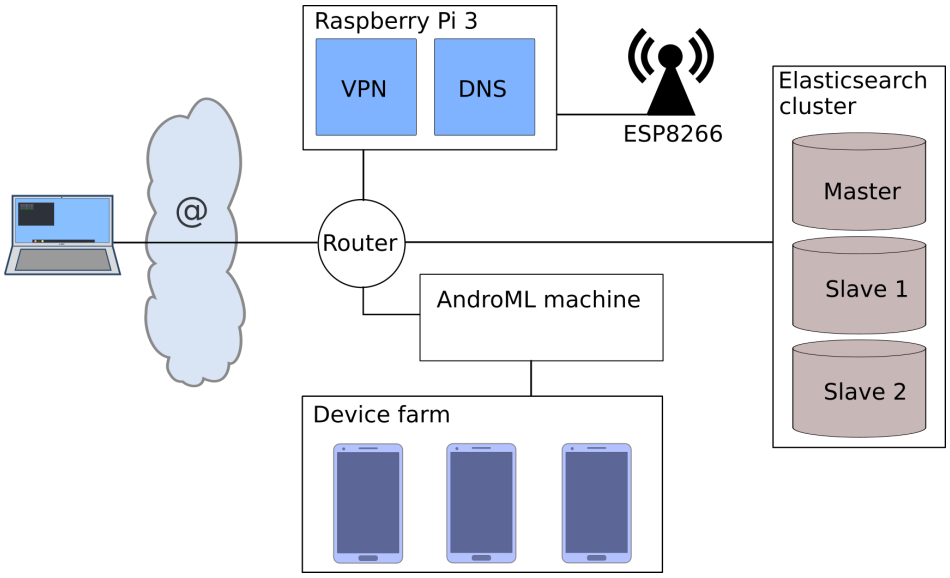

Figure 6.13: Environment overview

The analysis took place in a home network on the *AndroML machine*. This is a Debian notebook containing the APK data collection (see section 5.2) and the program code of the analysis as well as the machine learning prototype. The Elasticsearch cluster is composed of one master node and two slave nodes. They guarantee reliability due to replication of the data. Not all slave nodes were running permanently, in consequence of high energy consumption. Details can be found in section 6.4. The home network used a central router with integrated switch to connect the devices. A DNS server run on a Raspberry Pi 3 [6] in order to work with names instead of IP addresses within the network. This tiny single board computer is very suitable for being online permanently, due to the low power consumption. The DNS server functionality was provided by *dnsmasq* [7]. The raspberry was also utilized as a VPN gateway using *PiVPN* [8]. This made it possible to communicate with the Elasticsearch cluster and to check the analysis status from the internet. In order to record network traffic during the dynamic analysis a WiFi network was established using a *ESP8266* WiFi *System on a Chip (SoC)* [9]. A picture of the setup is shown in figure 6.15.

Since only one Nexus 5 was available within the DAI laboratory, two other devices were purchased. Due to relatively high prices of devices in sound condition, it was cheaper to buy devices with broken displays. Unfortunately the touch interaction

[6]https://www.raspberrypi.org/ (Last access on 10.05.2018)
[7]http://www.thekelleys.org.uk/dnsmasq/doc.html (Last access on 10.05.2018)
[8]http://www.pivpn.io/ (Last access on 10.05.2018)
[9]http://esp8266.net/ (Last access on 10.05.2018)

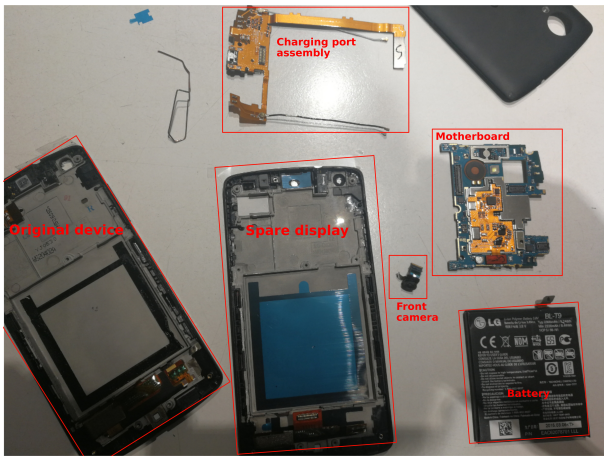

Figure 6.14: Changing the display of a Nexus 5

Figure 6.15: Setup of the environment

was affected due to broken Digitizers. Therefor both broken displays and Digitizers were replaced with working spare parts as depicted in figure 6.14. In summary this assembly process saved money and gave interesting insights into mobile device construction.

The interaction with the Android devices and the installation of the XPosed framework needed some preparation. On all devices a custom recovery TWRP should be installed in order to be able to root the devices and enable backup and restore capabilities (also called NAND backup). The best way to achieve this was to configure one device in the first place. In the next step the system was backed up to an image, which was then copied onto the other devices. This way every device used the same image, which enabled identical analysis conditions.

In order to install TWRP, the devices were unlocked by using `fastboot oem unlock` within the fast boot mode. See section 2.3.4 for more details. Subsequently the TWRP image was flashed onto the device by using `fastboot flash recovery twrp-<version>-hammerhead.img`. After booting into recovery the device was rooted. The app *SuperSU* [10] was used to control the root access. This was done for all three devices.

In the next step one device was chosen to build the common configuration. The

[10] http://www.supersu.com/ (Last access on 10.05.2018)

original Android 6.0.1 (API 23) was used. It was updated with security patches. In the next step the XPosed Framework was installed via `adb install`. Afterwards the Droidmon module was installed and was activated in the XPosed app. It was required to enable USB debugging to communicate through adb. Therefor an activation of the developer settings was required. A WiFi connection was set up and acted as a black hole. It dropped internet network requests, thus an app was not able to establish a connection. The device screen was set to *stay active*. Thus the display stays permanently active when power is connected. This configuration is located in the developer options. The SuperSU app was configured so only Xposed and adb were able to request root permissions. The devices were connected to the *AndroML machine* and were tested using the analysis tests. After a successful testing procedure a NAND backup of one device was created using TWRP. It was stored on the machine and deployed on the other two devices. All three devices were using the same operating system and the same configuration in order to provide common analysis conditions. The devices were put on a vibration absorbing pad and were connected to a Debian machine, which executed the analysis application. Furthermore the speaker were taped to reduce the noise during the analysis.

6.4 Elasticsearch

This section deals with the database for the analysis results. The setup of Elasticsearch is described in the following lines. The used module `database` in the analysis prototype is outlined in subsection 6.4.1.
In order to use the Elasticsearch cluster the required package must be downloaded from the elastic website [11]. Elasticsearch uses Java which requires an appropriate JVM. In version 6.2.2 as least Java 8 is required [12]. Elasticsearch is installed on three machines to run a cluster. The machines are depicted in figure 6.13. Elasticsearch needs a lot of file descriptors or file handles. To prevent data loss the amount of maximum file descriptors is set to a minimum of 65.536. The virtual memory amount is also increased to 262144 memory map areas [13]. The `elasticsearch.yml` is the configuration file of an Elasticsearch instance. Within this file the cluster and node name is set. Configurable are also network settings and the node master. The JVM

[11] https://www.elastic.co/downloads/elasticsearch (Last access on 10.05.2018)
[12] https://www.elastic.co/guide/en/elasticsearch/reference/current/setup.html
[13] https://www.kernel.org/doc/Documentation/sysctl/vm.txt (Last access on 10.05.2018)

configuration file `jvm.options` is used to configure the minimum and maximum head size of the Java Virtual Machine. The log settings are modified within the `log4j2.properties` file. Kibana is installed alongside Elasticsearch to provide a web UI for the resulting data of the analysis.

6.4.1 Module `database`

In order to interact with Elasticsearch the `database` package is used. The main entry point for the analysis logic is the `ElasticAdapter` class. It implements the interface `Database`. This realizes the Dependency Inversion Principle (DIP) which simplifies a potentially exchange of the database. The module provides communication with the database by directly invoking endpoints and is also able to build appropriate mappings. An overview about the package is provided in figure 6.16.

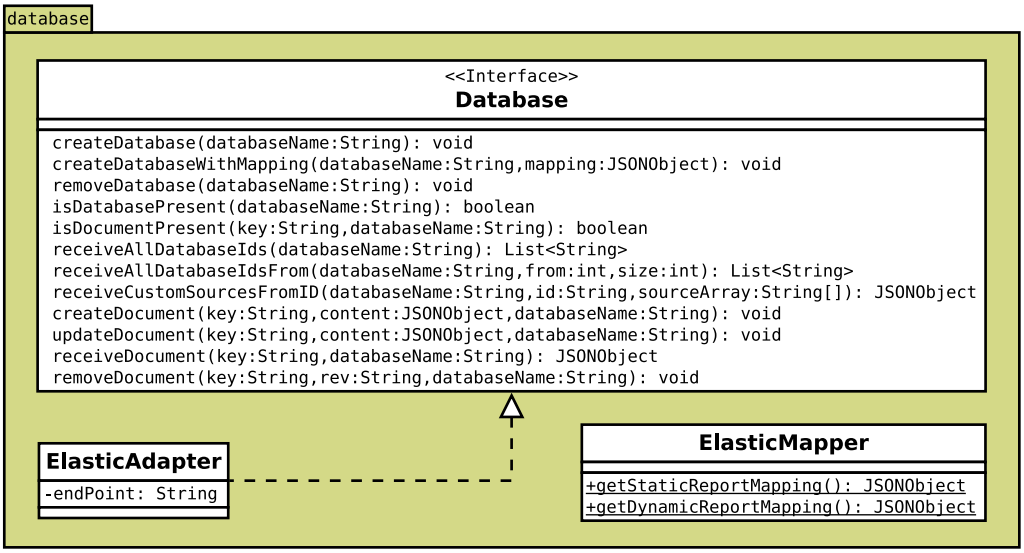

Figure 6.16: Class diagram for `database` module

The `ElasticAdapter` uses the `AndroMLConfig` in order to build the database endpoint of the database. The endpoint is stored within the class attribute `endPoint`. All required functionality is provided by the `Database` interface and implemented within the class `ElasticAdapter`. Due to brevity the methods are omitted in the class diagram for `ElasticAdapter` but shown in the interface. In order to communicate with the Elasticsearch cluster HTTP and REST is used. The basics are described in section 2.3.1. For sending and receiving HTTP requests and responses the library *Unirest*

[14] is utilized. It provides convenient access to HTTP methods and has an excellent documentation. The class `ElasticAdapter` provides creation and deletion of indexes and documents. It is also able check if an index or document is present. Moreover `ElasticAdapter` can receive a specific range of ids from a given index. It is also possible to create, update, receive and to remove documents within specific indices. The class `ElasticMapper` provides two static methods `getStaticReportMapping()` and `getDynamicReportMapping()` in order to provide a specific mapping for the Elasticsearch indices.

6.5 Results

The analysis took place from mid of December 2017 to end of March 2018. The next lines will outline the duration of the analysis followed by an overview about the results. Some collected apps could not be analyzed either in a static or dynamic way. This results in a reduced amount of analysis results.
In order to answer the research question and reach the goals of this thesis, two hybrid analyses were performed. In total a set of 4054 malicious and 5151 benign apps were analyzed. The foundation of the hybrid analysis is the static analysis, which was performed once for both app categories. The static reports are used for the component and interactive approach. The static analysis would take a few hours for all apps without requesting Virustotal. However utilizing the Virustotal API with a limit of 4 request per minute increased the analysis duration. The static analysis for 9205 apps took 40 hours in total.
For the dynamic analysis the component and interactive approach were executed. The component procedure installed the apps, launched the defined components and uninstalled the app afterwards. Sometimes API logs remained empty and the device needs to reboot. The analysis was performed on three devices in parallel, until one device broke. In total the component analysis took around two weeks.
The two devices left were used for the interactive analysis. The threshold for the click runner was set to ten minutes. Including overhead the analysis for 9205 apps took in total around 40 days.
It was required to patch bugs and fix crashes during the analysis phase. This took several days for both analysis approaches in total. Malware sometimes reconfigured the device, which requires to restore the original image via TWRP.

[14] http://unirest.io/java (Last access on 10.05.2018)

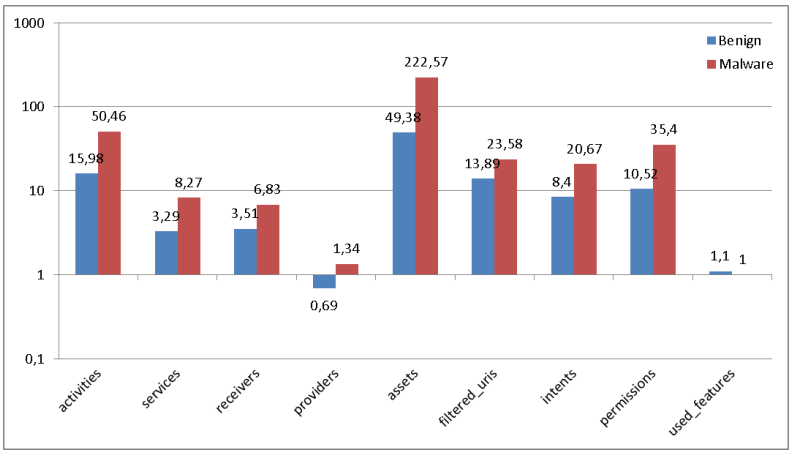

Figure 6.17: Average amounts of collected static data

To summarize the results of the static analysis average amounts of several data types are depicted in figure 6.17. The chart uses a logarithm scale base 10 for the average amount on the y-axis. The x-axis contains the categories of collected data. The red bars illustrate the malicious apps and the blue bars represent the benign apps. The first four categories contain the defined Android components of the Android manifest. Next the average amount of assets and found URIs are depicted. The intent, permission and used feature amount are illustrated on the right side. In general malicious apps contain a higher average amount of the described categories than the benign apps. Especially the difference of 222,57 average asset files for malware and 49,38 for benign apps is remarkable. Also the average amount of activities of malware is around 34 higher compared to benign apps. Malware also uses around 25 permissions more on average.

An overview about the results of the dynamic analysis are depicted in figure 6.18. On the x-axis the chart shows the average amount of API calls grouped into ten categories. For every category two stacked bar charts are shown one for the component and one the interactive approach. Every stacked bar is divided into the average amount of API calls for malware and benign apps. As illustrated, the majority of detected API calls are located in the categories *globals*, *reflection*, *content* and *file*. Less are assigned to *network* and *dex*. The total average amount of API calls for malicious and benign apps together is always higher in the component approach as in the interactive procedure. However these results show the plain API calls from the log. This also includes identical invocations with different timestamps. It can be assumed that the higher API call amount in the component approach could be caused by a redundant execution of identical code. This is eventually the case, when

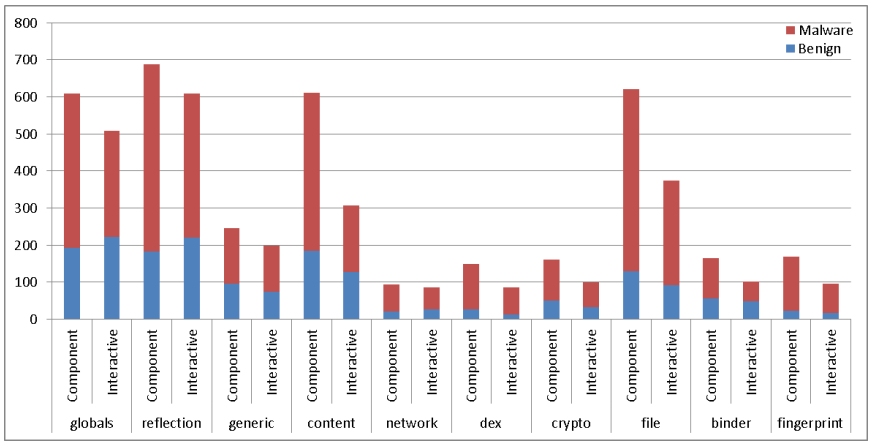

Figure 6.18: Average API calls per category

components are started manually without context. The component analysis could have a higher code coverage than the interactive analysis. Currently login dialogs of apps are not covered by the interactive analysis, which could cause this behavior. Another explanation could be the used time threshold. If increased, the interactive approach could achieve improved code coverage. Assignments of methods to the shown categories can be obtained from the `hooks.json` [15].

The chapter A contains six further charts. In figure A.1 the average positive detection results for the malicious and benign data set is shown. The apps from the malware data set are detected with 17.79 positives and benign app with 0.478 in average. That means that the repository Appsapk contains malware. The machine learning approach will assign the correct categories, so the teacher noise will be reduced. Figure A.2 shows the average amounts of classes, methods and system classes of malicious and benign apps. The two figures A.3 and A.4 compare the top five target SDK levels of benign and malicious apps. The most used target API level for benign apps is API 22 whereas malware uses an API level of 19. Google introduces the runtime permissions in API level 23. It is assumed that Android malware avoids runtime permissions since less interaction with the user is required. Runtime permissions much more involve the user in granting permissions. The two figures A.5 and A.6 compare the top five permissions used in the benign and the malware data set.

[15]https://github.com/idanr1986/droidmon/blob/master/hooks.json (Last access 10.05.2018)

7 Machine Learning

This chapter describes the implementation of the machine learning prototype. Section 7.1 will introduce general implementation details, followed by the classification approach in section 7.2 and the clustering in section 7.3.

7.1 General

In order to implement the prototype the Python language in version 3.6.4. is utilized. The prototype is created with Vim as IDE on an Arch Linux machine. The *Flake8* plugin [1] served as a Linter. For debugging `pdb` [2] is used. The general dependencies for the prototype are the following:

- `argparse` [3]
- `logging` [4]
- `pytest` [5]

The package `argparse` enables parsing of arguments when starting the Python application. The `logging` package enables the logging feature. The log level is set to *INFO* by default. The package `pytest` is used as a testing framework for the application. Due to brevity tests are not content of this chapter.
The Python prototype uses the package structure depicted in figure 7.1:

[1] http://flake8.pycqa.org/en/latest/ (Last access on 10.05.18)
[2] https://docs.python.org/3/library/pdb.html (Last access on 01.05.2018)
[3] https://docs.python.org/3/library/argparse.html (Last access on 01.05.2018)
[4] https://docs.python.org/3/library/logging.html (Last access on 01.05.2018)
[5] https://docs.pytest.org/en/latest/ (Last access on 01.05.2018)

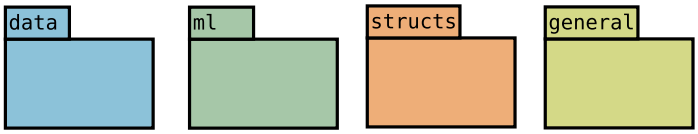

Figure 7.1: Overview of the package structure

The `data` module contains classes responsible for loading data. Package `ml` contains functionality for machine learning. `structs` holds data structures for representing apps and their analysis data. The package `general` contains generic data structures necessary for implementation. A detailed class diagram is depicted in figure B.2.

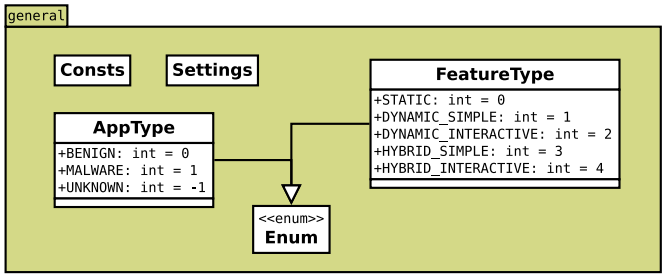

Figure 7.2: Class diagram for the `general` package

The package `general` holds four classes in total. The class `Const` contains important constants like the group of used index names for Elasticsearch and various strings in order to access the reports of the documents. The class `Settings` holds settings like the Elasticsearch configuration including the endpoint IP and several default configurations, which can be changed by using arguments. As arguments the amount of processed apps or the used feature class like static, dynamic or hybrid can be used. Also the amount of malicious and benign apps can be set by using an argument. Both classes are depicted in figure B.2 without the constants for retaining a clear overview. The classes `FeatureType` and `AppType` inherit the `Enum` class. The first one provides specific feature configurations and the second one is used for the determination of the class label.

This prototype performs classification and additional clustering. Both modules are started using separate python files. Their functionality is described in the following sections 7.2 and 7.3.
Classification is ran by using `classify.py` and clustering by `cluster.py`. Both implementations accept the described arguments.

7.2 Classification

An overview about classes and data structures involved in the classification can be found in figure 7.3. The classification is performed in the `ml` package and requests required data from the `data` package. The `data` package uses the `structs` package to structure the apps and their analysis results. The classes in the packages be connected in the depicted way.

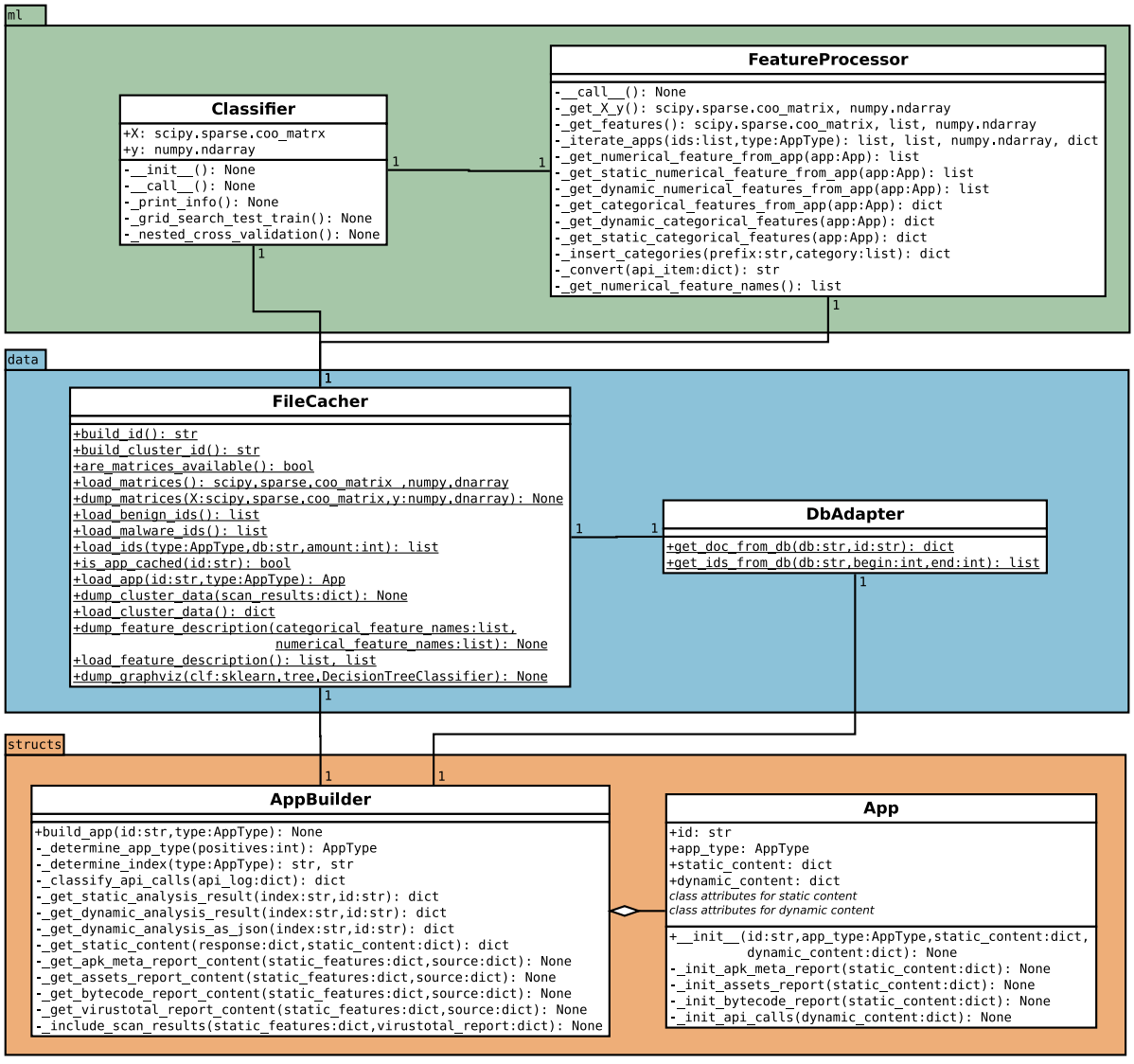

Figure 7.3: Class diagram for classification

The class `Classifier` is initialized by calling `__init__()`. This method creates an instance of `FeatureProcessor`. By calling the class, the `X` and `y` matrices for the

classification process are returned. Variable `X` contains the data points including the feature vectors and `y` contains the appropriate label assignments. Afterwards `print_info()` is called to output information about the size of the matrices. Calling the `Classifier` instance invokes the method `__call__`. This method performs a grid search with a train/test data split or a nested cross-validation using grid search. What is used is configured within the settings.
For the first approach the method `_grid_search_test_train()` is called. It splits the data using the scikit-learn method `train_test_split()` from the package `sklearn.model_selection`. The split size is divided into 60% train and 40% test data. The stratify option is used to retain the class distribution within the sets. A random seed value is set to be able to verify the results after multiple calculations. Afterwards the `RandomForestClassifier()` estimator and a parameter grid are created. They are used as arguments for `GridSearchCV()` object, also available in the model selection package of scikit-learn. Next the model is fitted by invoking `fit()` and using the train data `X_train` and `y_train`. The best estimator is chosen and applied to the test set by calling `predict()`. The results are plotted using a confusion matrix. The `ConfusionMatrix` object is provided by the SciPy package.

For nested cross-validation the method `_nested_cross_validation()` is invoked. Instead of manually split the data and call `fit()` and `predict()` with train and test sets, the estimator `GridSearchCV` is used together with `X` and `y` in the method `cross_val_score()` of the package `sklearn.model_selection`. It performs the cross-validation with a grid search and returns a score array, which is outputted together with the mean of all scores.
The described approaches will be applied in the evaluation in chapter 8.

7.2.1 The `FeatureProcessor` Class

This subsection describes the implementation of the feature engineering part. The `FeatureProcessor` is responsible for providing the `X` and `y` matrices. Therefor the method `_get_X_y()` is utilized. It uses a `FileCacher` instance, which is stored in the variable `fc`. First the `FeatureProcessor` checks if the matrices are already available by calling `fc.are_matrices_available()`. If yes, `fc.load_matrices()` is invoked and `X` and `y` are returned. If no, the `FeatureProcessor` instance generates them. This is done the following way.

First indices for benign and malware apps are received by invoking `load_benign_ids()` and `load_malware_ids()` of the `FileCacher`. The `fc` looks for cached data first and otherwise requests the data using the `DbAdapter` and `get_ids_from_db()`. Afterwards `_get_features()` is called, which returns a list of numerical and categorical features together with the label matrix `y`. In order to receive these features the method contains the invocation of `_iterate_apps()`. It processes the apps for benign and malware category.
The method `_iterate_apps()` iterates over all ids of an app category and build instances of the class `App` by invoking `fc.load_app(id, type)`. The method either returns the app by loading the cached app or using `AppBuilder` to build a fresh instance of `App`. This functionality is described in more detail in subsection 7.2.2. To extract the collected information of the app the methods `_get_numerical_features_from_app()` and `_get_categorical_features_from_app()` are invoked. They use sub-methods to separate static and dynamic features. A list with the features for every method is returned. Numerical features are returned as list and categorical features are returned as dictionary per app.
A list of numerical features is stored into the variable `numericals` and a list of categorical features into the variable `categories`. Categorical content is transformed using *one-hot-encoding*. During each iteration the app type is determined and stored into a list. The scan results of the Virustotal report are stored for every app and returned as list. In the method `_get_features()` the following data for benign and malware apps is available: label information, categorical- and numerical features and the scan results of Virustotal. Numerical and categorical features are merged into a common data structure. Scan results are cached using `fc.dump_cluster_data()`. They will be used during the clustering approach described in section 7.3. The label information is merged into the matrix `y`.
The method `_get_X_y()` joins the categories using the `hstack` method, which is shown in code snippet 11. This method is part of the SciPy library. Beforehand the categorical features are processed using a `DictVectorizer`. This is required for *one-hot-encoding* to create a single vector for the complete dictionary.

```
numericals, categories, y = self._get_features()
vectorizer = vec.fit_transform(categories)
X = hstack([vectorizer, numericals])
```

Code snippet 11: Merge features and create the matrix `X`

The following subsection will introduce the creation of `App` instances using the `AppBuilder` class.

7.2.2 The `structs` Package

This package contains the classes `AppBuilder` and `App`. It is used if no cached app is available. An `AppBuilder` instance is created in the `FileCacher` and invoked by calling `build_app(id, type)`. The code for this function is listed in figure 12.

```
1  def build_app(self, id, type):
2      LOGGER.info(f'Build app with id: {id}')
3      static_index, dynamic_index = self._determine_index(type)
4      static_result = self._get_static_analysis_result(static_index, id)
5      dynamic_result = self._get_dynamic_analysis_result(dynamic_index, id)
6      if (Consts.POSITIVES in static_result):
7          positives = static_result[Consts.POSITIVES]
8          app_type = self._determine_app_type(positives)
9          return App(id, app_type, static_result, dynamic_result)
10     else:
11         return App(id, AppType.UNKNOWN, static_result, dynamic_result)
```

Code snippet 12: The function `build_app` in `AppBuilder`

First the used dynamic and static indices are determined by calling `_determine_index()` in line 3. Next the methods `_get_static_analysis_results()` and `_get_dynamic_analysis_result()` are invoked using the index and the app id (see line 4 and 5).
The method `_get_static_analysis_results()` requests the database using `get_doc_from_db()` and receives the static report document. An empty dictionary `static_content` is created to store the static information as key-values. The

dictionary is passed as reference to `_get_static_content()` and is filled with the analysis results. The method calls four methods to extract data from the APK-meta, asset, bytecode and Virustotal report. An example is the method `get_apk_meta_content()` depicted in figure 7.3.
The method `_get_dynamic_analysis_results()` invokes `_get_dynamic_analysis_as_json()`, which retrieves the document for the appropriate dynamic index. The result is stored and returned in JSON format. Afterwards `_classify_api_calls()` is called. This method takes the JSON and transform it into a dictionary, which classifies the API calls.

```
1 classified_api_calls = {'globals':[], 'reflection':[], 'generic':[], 'content':[],
2 'network':[], 'dex':[], 'crypto':[], 'file':[], 'binder':[], 'fingerprint':[]}
```

Code snippet 13: The empty data structure for classified API calls

As keys the API category and as values the list of entries are used. The empty data structure is depicted in code snippet 13.
Finally the method `build_app()` builds an instance of `App`. The Virustotal report is used to determine the app type by calling `_determine_app_type()`. The method decides if the app is classified as malicious or benign. Afterwards the app is created and returned (see line 9). The `else` case is used if it is not possible to determine the app type due to problems with the Virustotal report (see line 11). In this case the app type is set to `Type.UNKNOWN`.
The constructor of the class `App` receives the `static_results` and `dynamic_results`. They are stored in the variables `static_content` and `dynamic_content`. Afterwards several methods are used to extract features from the content variables and store them in a direct `App` class attribute. This makes the feature directly accessible in an instance of `App` and improves the structure.

7.3 Clustering

The sections deals with the clustering process, which is executed after the classification. An overview of the module and data structures is depicted in figure 7.4.

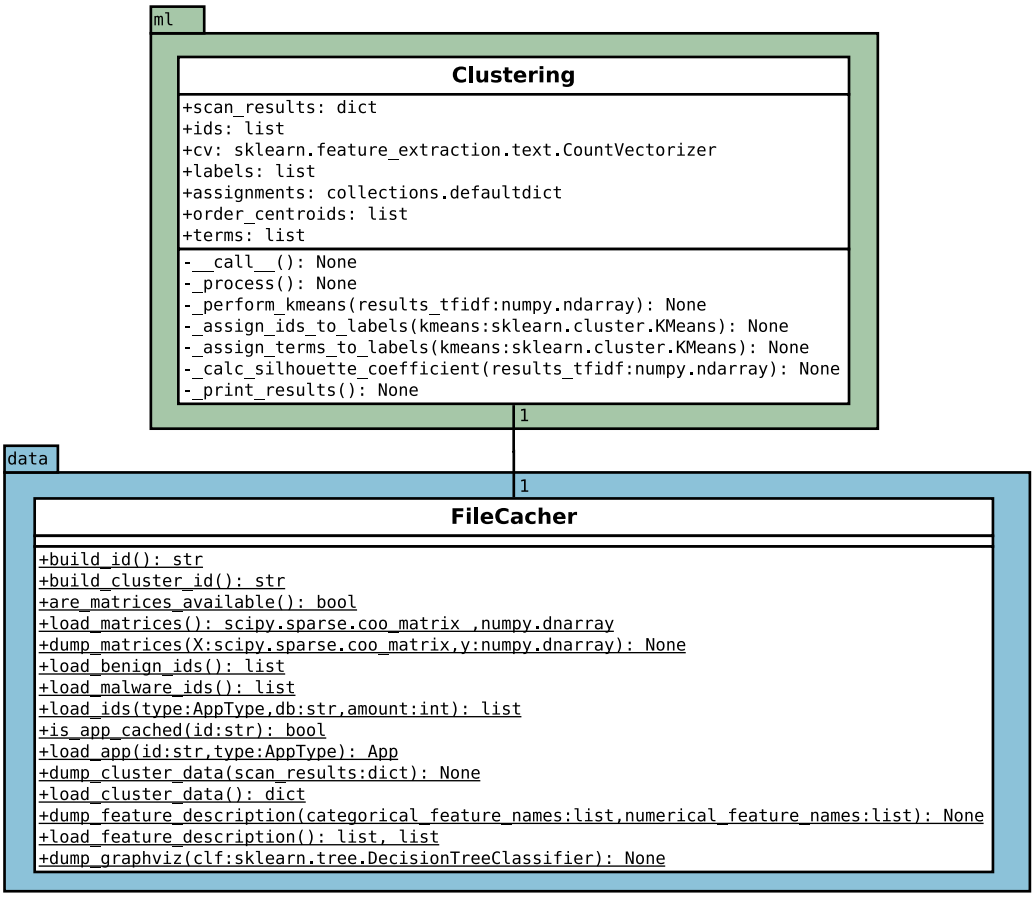

Figure 7.4: Class diagram for clustering

An instance of `Clustering` is created and called in `cluster.py`. This invokes the method `load_cluster_data()` from the `FileCacher` instance. This methods returns a dictionary with the app ids as keys and the results of the virus scanner as values. For every result of a virus scanner a string is stored. This dictionary is stored as `scan_results_raw`. The data was previously collected by the classification module. The raw results are converted into a new dictionary where a single concatenated string for the value is used. The strings are now delimited with spaces and converted to lower case. Any occurrences of *android* and *androidos* were removed, since the terms don't add value to the clustering approach. The code is listed in code snippet 14.

```
self.scan_results =
{id:' '.join(results).lower().replace("androidos", "").replace("android","")
for id, results in scan_results_raw.items()}
```

Code snippet 14: Filter scan results

The result is stored in `scan_results`. In the end the `__call__()` method invokes `_process()`. The method initializes a `CountVectorizer` object stored in the variable `cv` from scikit-learn. The invocation of the method `cv.fit_transform(scan_results.values())` learns the dictionary using the concatenated strings. A term/document matrix is returned and stored in `results`. Documents represent the malware apps and terms the used Virustotal tags. Every app in this data structure is a vector with the length of the dictionary.
In order to deal with the frequency of words TF-IDF is used (see chapter 2.2.2). Therefor the method `results_tfidf = TfidfTransformer.fit_transform(results)` is invoked. The results are handed over to `_perform_kmeans(results_tfidf)`. The method performs k-means by invoking `kmeans = KMeans(n_clusters = Settings.CLUSTER_SIZE).fit(results_tfidf)`. The result is passed to the methods `_assign_ids_to_labels(kmeans)` and `_assign_feature_names_to_labels(kmeans)`. The first method creates the dictionary `assignments` and includes the label as key and the list of connected ids as values. This is realized by using `kmeans.labels_`, which stores the cluster label for every app in a sorted way. With the help of `ids` it is possible to assign all apps to their appropriate cluster.
The second method `_assign_feature_names_to_labels(kmeans)` assigns the used top terms to the cluster labels. The cluster centroids are sorted and stored into `order_centroids` using `kmeans.cluster_centers_`. The terms are stored using `terms = cv.get_feature_names()`. Both together fill a `defaultdict` named `category_terms` with the category as key and appropriate terms as value. The method `_print_results()` prints out the results

8 Evaluation

In order to answer the research question on *"how precise is the predictive identification of recent Android malware by comparing two different hybrid analysis approaches?"*, an evaluation has to be performed. Therefor the implemented classification approach is used. To provide further insights into the analyzed malware a clustering is applied. Moreover it is necessary to define the evaluation data and procedures.
The approach for the classification can be found in section 8.1, while details regarding clustering can be found in section 8.2. This evaluation involves a total amount of 9205 apps from which 4054 are malware and 5151 are benign apps.

8.1 Classification

In order to evaluate the classification two hybrid approaches will be compared: the component- and the interactive analysis. Two different procedures for the evaluation will be applied as well. Both will utilize the two analysis approaches. The first one compares the analysis prediction using a train/test split with parameter optimization. This approach provides insights into the mechanism and enables subsequent use of the model. The second procedure is a nested cross-validation approach. This is executed for both hybrid approaches, both dynamic analyses and the static analysis. For the prediction the `RandomForestClassifier` object is used (see section 7.2). The first evaluation approach returns confusion matrices and the second one accuracy values. Conclusions about the results will be covered in chapter 9. Details about the approaches are outlined in the upcoming segment.
The train/test split is used to shuffle the data and to split them into 60% training data and 40% test data. The training data is divided into a training set `X_train` and `y_train` for labels. The test data is composed of a test set `X_test` and `y_test`. Moreover a random seed is used in order to create similar and comparable data sets when executing multiple evaluations. For the execution the scikit-learn method

`test_train_split()` from `sklearn.model_selection` is used. The ratio of malware and benign apps is preserved by setting the `stratify=self.y` option. The algorithm uses the distribution of classes in the label matrix `y`.
The training set is used for parameter optimization. This prevents leakage of knowledge from the test set into the model, which would result in an overfitted model. For the parameter optimization a cross-validation is used. This functionality is provided by the scikit-learn meta estimator `GridSearchCV`. Grid search uses a 3-fold stratified cross-validation by default.

Two parameters are considered to optimize the performance of the random forest. The first parameter is called `n_estimators` and defines the amount of used trees in the forest. The second parameter is named `max_features` and defines a specific amount of features when looking for the best split [Sci18b]. For `n_estimators` a range of $[20, 200]$ is used. Values lower than 20 result in poor prediction performance. Higher values require considerably more computational effort without increasing performance. For `max_features` the two most commonly applied parameters `sqrt` and `log2` are used. Using a range of fixed numbers is omitted, due to increased computational effort.

The `GridSearchCV` needs to operate on the random forest estimator, the parameter dictionary and the number of folds. For this approach ten folds are used. This is a balanced trade-off between evaluation precision and performance. Due to the bagging approach of a random forest, it is recommended to let the trees grow full size [BC04]. The random forest automatically prevents overfitting [Bre01]. If a single decision tree is used, parameters like the maximum tree depth or minimum number of samples for a leaf should be configured.
The grid search is executed using the training data. Parameters are finely adjusted and 9197 apps are processed in total. Eight apps have no valid Virustotal report and are therefor excluded. The results for the optimal parameters are depicted in table 8.1. The feature values differ in exactly one feature, which is caused by the dynamic analysis as shown in table 8.2.

Table 8.1: Results of the parameter optimization using train/test split

	Used feature amount and optimized parameters		
Analysis	**Feature amount**	**n_estimators**	**max_features**
hybrid component	163558	143	sqrt
hybrid interactive	163559	140	sqrt

The feature is located in the API category *generic* and calls the `abortBroadcast()` method from the class `BroadcastReceiver`. The dynamic feature amount is composed of ten numerical and 53 categorical features for the component approach and respectively 54 categorical features for the interactive procedure.

```
['android.permission.READ_PHONE_STATE',
'api_fingerprint::android.telephony.TelephonyManager->getSubscriberId',
'api_fingerprint::android.telephony.TelephonyManager->getDeviceId',
'android.permission.ACCESS_WIFI_STATE', 'assets_amount',
'com.android.launcher.permission.INSTALL_SHORTCUT',
'total_method_amount', 'permission_amount',
'android.permission.SEND_SMS', 'total_classes_amount']
```

Code snippet 15: Top ten features of the hybrid component approach

```
['api_fingerprint::android.telephony.TelephonyManager->getSubscriberId',
'permission_amount', 'android.permission.GET_TASKS',
'android.permission.MOUNT_UNMOUNT_FILESYSTEMS',
'api_fingerprint::android.telephony.TelephonyManager->getDeviceId',
'assets_amount', 'com.android.launcher.permission.INSTALL_SHORTCUT',
'android.permission.READ_PHONE_STATE', 'filtered_class_names_amount',
'total_classes_amount']
```

Code snippet 16: Top ten features of the hybrid interactive approach

The top ten features of both approaches are shown in the code snippets 15 and 16. They differ at some points regarding their order and features. Both of them cover two dynamic API calls from the *fingerprint* category. The method `getSubscriberId()` returns the *International mobile subscriber identity (IMSI)* for a GSM device. Invoking `getDeviceId()` returns the device identifier. Common static numerical features are for example the amount of assets, permissions and classes. The categorical features used in both approaches are the permissions for reading the phone state and installing shortcuts. However, differences can be found in the permission for sending SMS and `android.permission.GET_TASKS`. This enables an app to get information about tasks started by the user.
The parameters from table 8.1 are used to predict the labels for the apps in the test set `X_test` and compare the results with the true labels `y_test`. Around 404

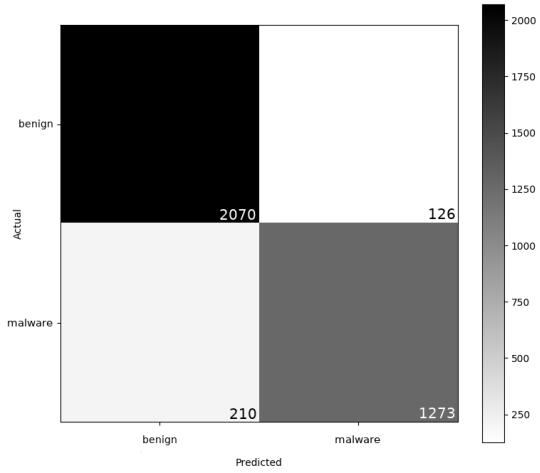

Figure 8.1: Confusion matrix of the component-based hybrid analysis

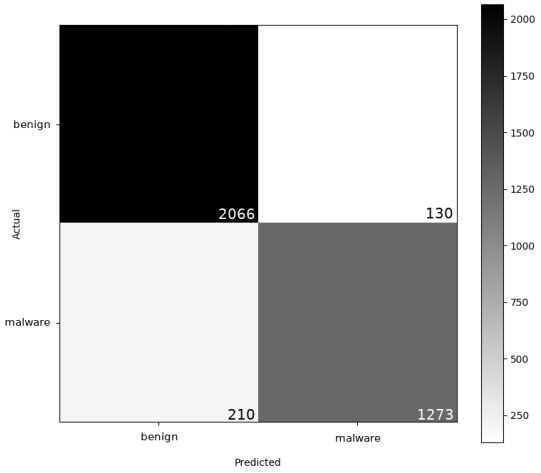

Figure 8.2: Confusion matrix of the interactive hybrid analysis

features in total are used in order to detect the best split. The results are presented as confusion matrices shown in depiction 8.1 and 8.2. The amount of apps per category are stated in the cells. These confusion matrices are used to calculate the final metrics for the results of both approaches:

$$acc_{component} = 0{,}9087 \tag{8.1}$$

$$acc_{interactive} = 0{,}9076 \tag{8.2}$$

$$f_1^{component} = 0{,}8834 \tag{8.3}$$

$$f_1^{interactive} = 0{,}8822 \tag{8.4}$$

The accuracy of both approaches is shown in formula 8.1 and 8.2. The appropriate f_1 measure can be found in formula 8.3 and 8.4. An improved approach is the usage of the nested cross-validation. It performs parameter optimization on different training-, validation- and test folds. This will result in improved accuracy compared to the previous evaluation approach. It is caused by the utilization of a cross-validation instead of a train/test split.

Table 8.2: Results for the nested cross-validation

	Used feature amount and mean accuracy values		
Analysis	**Feature amount**	**Mean accuracy**	**Median accuracy**
static	163495	0.894	0.941
dynamic component	63	0.868	0.912
dynamic interactive	64	0.869	0.915
hybrid component	163558	0.894	0.942
hybrid interactive	163559	0.894	0.94

A disadvantage is the missing possible to use the constructed model for later predictions. The results are plain measurements for the nested cross-validation with parameter optimization. The method `cross_val_score` of scikit-learn is used in combination with the meta estimator `GridSearchCV`. The variables `X` and `y` are handed over as parameters as well. The number of folds is set to 5 while the parameter `n_estimators` is set to a list of `[100, 120, 140, 160, 180, 200]` and `max_features` to the list `['sqrt', 'log2']`. The increased computational effort of this evaluation approach results in a reduced number of folds and a reduced range of values for `n_estimators`. The results of the nested cross-validation are shown in table 8.2. For each analysis approach the value of the feature amount, the mean accuracy as well as the value of the median accuracy are shown.

8.2 Clustering

This section deals with the clustering results for the 4054 malware apps. *K-means* is used for clustering (see section 2.2.2). As input data the ids of the apps and the Virustotal reports are utilized. The determination of the cluster size is a crucial step and supported by the silhouette coefficient (see chapter 2.2.4). In this evaluation an interval of [1,100] is applied. By default k-means runs ten times with different initial seed points [Sci18a]. The silhouette coefficients for $k = [2,100]$ are depicted in figure 8.3. For $k = 2$ the result is close to zero and can be interpreted as overlapping clusters. If k is increased until $k = 20$, the silhouette coefficient lies around 0.1. By increasing k the coefficient slowly increase as well. The peek for the cluster size in this interval lies around $k = 95$ clusters with the average value $s_i^{avg} = 0.17$ for all apps. This value is far away from 1, which denotes the perfect clustering result.

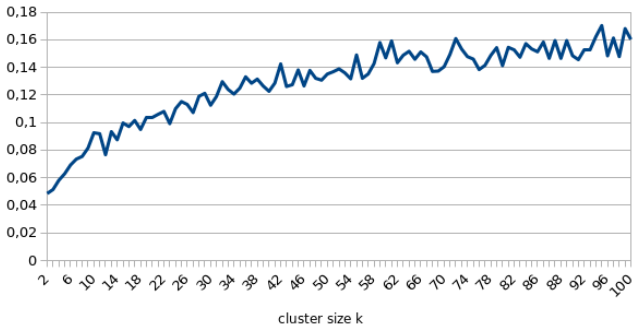

Figure 8.3: Silhouette coefficients for $k = [2,100]$

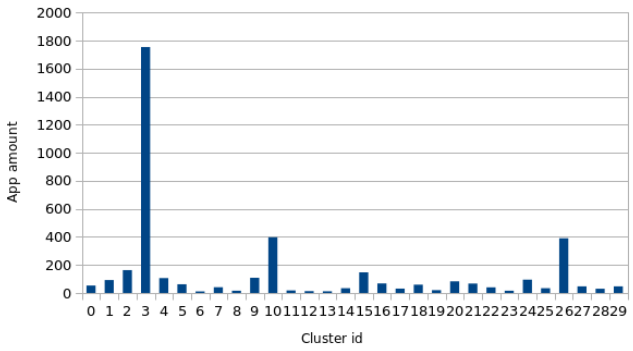

Figure 8.4: Amount of apps per cluster

Nevertheless, this coefficient should not replace the human review of cluster sizes. After inspecting the data for several k values, the details for $k = 30$ are shown in this work. This provides a sufficient representation for considerable tags and will not overload this section. The distribution of apps for cluster size $k = 30$ is depicted in figure 8.4. The x-axis shows the cluster id k, whereas the y-axis provides the amount of apps within the cluster. The most data points contains cluster 3 with 1754 apps, followed by cluster 10 with 397 apps and cluster 26 with 390 apps. All other clusters lie between 10 and 200 apps. To get more insight into the analyzed data the amount of clusters, the size of the particular cluster and the most significant keywords are listed in table 8.3.

Table 8.3: Overview about identified clusters

Id	Amount of apps	Keywords
0	53	droidrooter, exploit
1	92	smsreg, adware
2	163	skymobi, smspay
3	1754	trojan, pup, generic
4	106	smsreg, emagsoftware, dinehu
5	62	kingroot, droidrooter, rooter
6	11	cimsci, olympus, genpua
7	41	skymobi, smsreg, risktool
8	16	simpatchy, agent, cooee
9	108	kuguo, dowgin, addisplay
10	397	smsreg, riskware, risktool
11	18	tekwon, wondertek
12	13	plankton, apperhand, counterclank
13	12	bl, downloader, dropper
14	34	utchi, mtk, chitu
15	147	secapk, pup, pua
16	68	apptrack, spr, mytrackp
17	31	hiddad, fakeapp, mobidash
18	59	systemmonitor, moplus, backdoor
19	20	lockscreen, ransom
20	83	mobidash, yekrand, pua
21	67	dowgin, artemis, adware
22	40	g2p, suspicious, multi
23	16	boogr, lockscreen, ransom, itcpp
24	95	ewind, anydown, downloader
25	34	shedun, dropper, paccy, revo
26	390	dropper, ztorg, blouns
27	47	cimsci, smssend, misosms
28	30	aio, hypay, pua
29	47	subspod, sisnit, gdiwi

In the next lines some of the cluster groups will be described. Some malware types will be outlined to give the reader an overview of the analyzed malware. It was not

the intention to describe every single mentioned malware keyword. The goal is to extract meaningful keywords, in order to give the reader insights about identified malware. Moreover the user is able to perform further research regarding the malware types.
The biggest cluster with id 3 contains the significant keywords *trojan*, *pup* and *generic*. Pup stands for *Potential unwanted programm* [Mal18b]. This chapter contains applications which are detected as general malware or apps with Trojan functionality. Often these free apps include aggressive advertisement or privacy leaks. They don't contain a specific malware signature like for example *Plankton*, which collects sensitive data and forwards it to a remote location [F-S18b].
The cluster with id 10 contains keywords like *smsreg*, *riskware* and *risktool*. It contains more specific malware like riskware and a specific variant named smsreg. Some variants are marketed to improve battery life of the device, but collect various data from the device without the knowledge of the user [F-S18a].
The Cluster with id 26 contains the keywords *dropper*, *ztorg* and *blouns*. Dropper malware is sophisticated malware, which is able to act harmless in the first way. Installed, it decrypts assets, executes code and dynamically loads other APKs from untrusted sources. It requests installing these apps to the user during runtime. Within this thesis a multi-stage malware dropper was analyzed. It tries to trick the user to download a new Adobe Flash Player app. This cluster contains specific malware like ztorg. Variants of this malware are concealing its maliciousness by using string obfuscation for the location of remote content. They perform emulator detection and communicate encrypted with the remote server. They are also able to download Android applications from untrusted sources [Apv18].

Besides the larger cluster groups, others contain more specific malware types. *Shedun* (cluster 25) for example often repacks legitimate Android apps with adware or tries to root the device by using exploits. If installed they become very hard to remove [Ben18]. *Mobidash* for example also follows the trend of an adware. It is an ad SDK which can be added to nearly every Android app. It shows advertisements. If the user clicks on the ads, it tries to install additional applications [Mal18a].
Cluster 20 for example contains ransomware. A popular example are lockscreen apps, which were able to access the lockscreen function of Android, set it with a pseudorandom value and blackmail the user [Ven18]. The malware collection also contains clusters with rooting apps like cluster 0 or 5. In general they are often used by users with the intention to root their device. But sometimes these apps

are performing harmful activities as in the case of *Kingroot* [1]. The official website looks clean and professional, but anti malware vendors and discussions in the XDA forum try to warn the user about unwanted features of the app. Kingroot will install adware and try to gather information.

[1] https://kingroot.net/ (Last access on 10.05.2018)

9 Conclusion and Outlook

This chapter will review the results of the evaluation in chapter 8 divided into the main classification and the additional clustering approach. It will answer the central research question of this thesis. The section 9.1 will outline the recommended future work for this thesis.

The results of the first classification approach show a slightly better performance of the component hybrid analysis, when comparing accuracy and f_1 measurement (See formula 8.1 until 8.4 in chapter 8). However, if nested cross-validation is used the interactive hybrid approach performs slightly better. This can also be adapted when the component and interactive dynamic approaches are considered. Since the nested approach is more precise, its results should be given more weight. However, the results differ in the third decimal place. Therefor the final results for both approaches can be considered as comparable good. In the scenario without further modification the component hybrid analysis should be preferred, since it has a better performance and is less error prone compared to the interactive approach. The interactive approach might be more precise, if the analysis threshold would be increased to more than ten minutes. Increasing the feature amount of the dynamic analysis could change the impact and accuracy of the dynamic approach. Moreover a higher time threshold for the interactive approach could improve the code coverage. This in turn could enhance the prediction accuracy using this approach. Astonishing is the fact, that the static analysis results alone are capable of detecting the malware with the same accuracy than the hybrid approaches. It seems that the feature amount of the dynamic analysis is not able to provide added value to the static analysis. This might be true for this data set, but will not make the dynamic analysis useless. There are scenarios like dynamic code loading and emulator detection, where a on-device dynamic analysis is not replaceable.

In order to answer the central research question of this master's thesis: *How precise is the predictive identification of recent Android malware by comparing two different hybrid analysis approaches?*, one can denote that the predictive identification for

detecting recent Android malware lies at around 90% detection accuracy. Both analysis approaches do not differ noteworthy for the inspected scenario.
This result seems to keep up and sometimes outperform other machine learning approaches described in chapter 2.5. However various approaches achieve outstanding precision close to 100%. This might lie in the usage of outdated malware, which could be more easy to identify. Perhaps the large amount of PUP and generic malware in the recent malware collection could be harder to identify, than the utilized collections of the related work. A reason for that are improved APIs and a more hardened Linux kernel. The malware will focus more on tricking the user. Interesting is the increased usage of the target SDK version prior to the runtime permission feature. Due to backward compatibility these apps are still working using the prior installing permission dialog. This should be changed since the majority of Android devices is already using API level 23 including the runtime-permission feature. In order to prevent the installing of dynamically loaded untrusted applications, the function of installing 3rd party apps could be deactivated. This however damages the more open approach of the ecosystem in comparison to other mobile operating systems.
Within this work malware is labeled as malware if it has at least two detections of anti malware tools. The other apps are labeled as benign. This should minimize false positives and reduce training outliers. Trends deduced from malware samples are potentially unwanted programs, adware and multi-stage loading of untrusted applications and trying to trick the user into installing these applications. Also rooting using exploits is used.

9.1 Future Work

This chapter contains recommendations for additional and useful tasks in the future. It covers important thoughts and ideas, which were not included in this thesis, due to time and scope restrictions. This section can be separated into general proposals and improvements for static as well as dynamic analyses.
It is recommended to integrate the implemented prototypes into a productive solution. A recent machine learning model could be made available to classify unknown apps. The functionality could be provided via an API or a web application. Also imaginable is the deployment of the static analysis and a machine learning model on a mobile device, in order to enable malware recognition as an Android application. Moreover an integration into the analysis tool Androlyzer is recommended.

The collected older malware could be used to evaluate the prediction results in comparison to the recent ones. This could help to reconstruct the results from the related work chapter with nearly 100% accuracy and help to understand differences compared to recent malware. One can also investigate how the malware has changed and adjusted to the latest detection techniques. As a precaution the malware ids per cluster were stored for later usage. This allows an investigation of the detection accuracy for specific malware apps of a cluster against the benign apps. It is imaginable that for example malware with rooting capabilities could be detected more precisely than general PUPs. Also interesting is a modification of the labeling of malware. Therefor the positives threshold of the Virustotal aggregator is utilized. For example at least ten positive scan results of different scanners will label apps as malware.
For the dynamic analysis the amount of dynamic features could be increased, for example by utilizing arguments of the API calls. Additionally the logged internet traffic could be inspected and used as features. Meta-information about the network connection like source and target address or used protocols can be transformed into features. Also information about the payload can be included.
Another improvement would be handling multidex applications in the static analysis. For the interactive approach the click runner can be enhanced for example by generating input for specific input fields. Moreover additional swipe movements can be added. Also a good idea would be to make use of the `adb shell monkey` [1], which is able to generate different random input.
Also interesting would be a comparison between an emulator and real device dynamic analysis. Especially in the cases where emulator detection techniques of malware take effect. Therefor an emulator image with a more recent API level for dynamic analysis can be created. An advantage would be the possibility of massive deployment in the cloud.
For the static analysis an improvement would be the implementation of a multi-threaded static analyzer. Besides, the analysis of assets can be enhanced by analyzing the mime type and size of a specific asset. Maybe specific malware can be identified. An example would be dynamic code loading hided in a concealed picture file. Moreover an improvement would be adding a native code analysis.
For the clustering it is recommended to improve the keyword extraction separated for anti-malware tools. They differ in working and naming scheme. It is imaginable that they use the same keywords for different malware. Valuable would be the usage

[1] https://developer.android.com/studio/test/monkey.html (Last access on 10.05.18)

of *Latent Semantic Analysis (LSA)* in the keyword processing step, in order to reveal latent structures which could improve the clustering approach [DDF+90]. Besides the clustering of malware it is imaginable to cluster benign apps as well. Therefor descriptions for apps of one or several app stores can be used. This may be helpful to identify if specific benign categories are easier to separate from malware than others.

A Hybrid Analysis Results

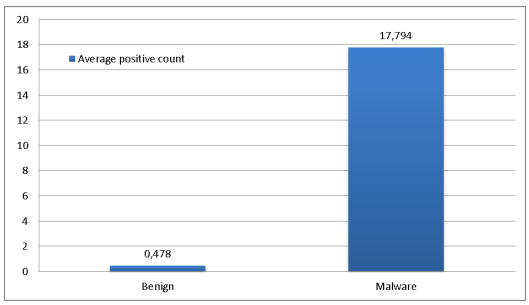

Figure A.1: Average positives of Virustotal service

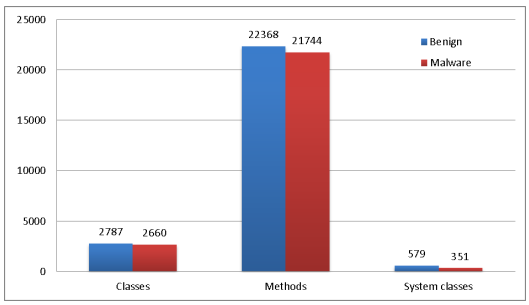

Figure A.2: Average amounts of bytecode statistics

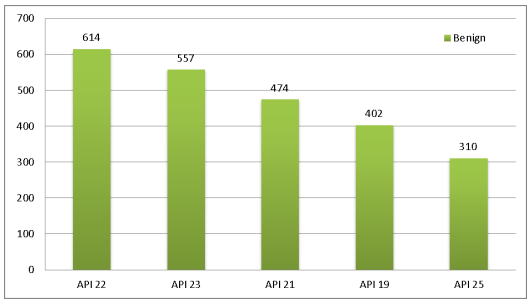

Figure A.3: Top five target SDKs for benign apps

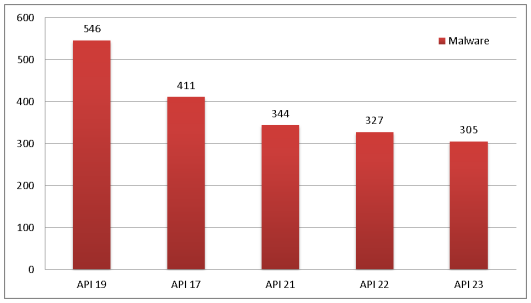

Figure A.4: Top five target SDKs for malware

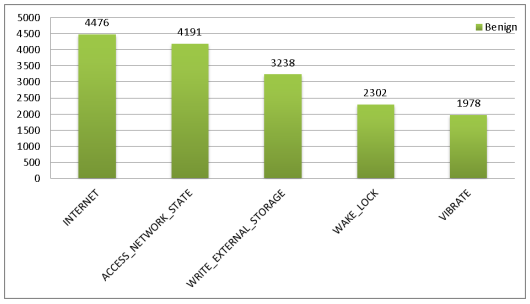

Figure A.5: Top five permissions for benign apps

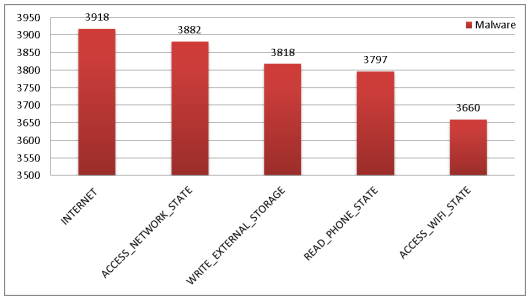

Figure A.6: Top five permissions for malware

B Class Diagrams

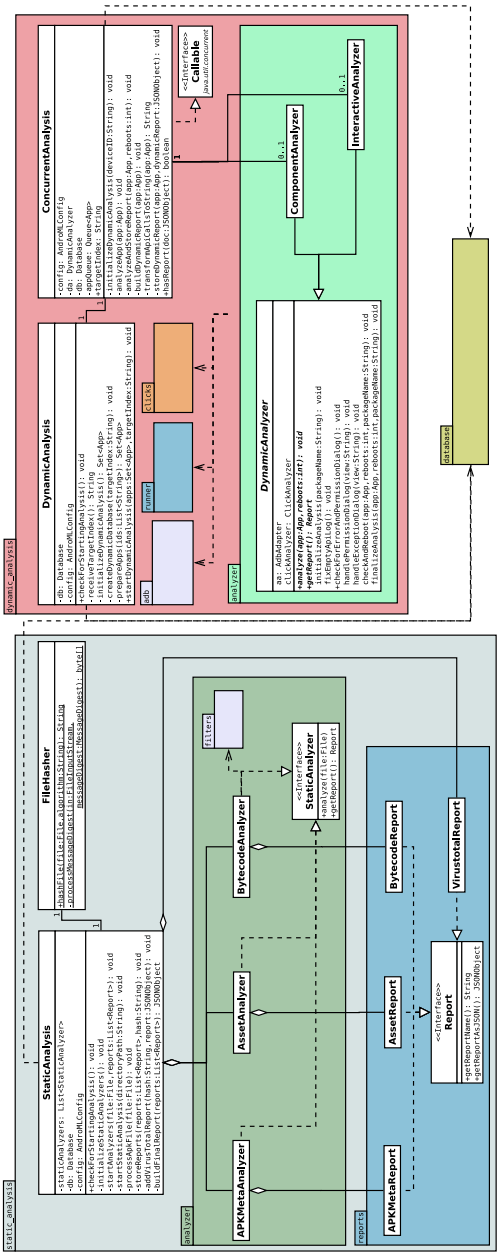

Figure B.1: Class diagram of the analysis prototype

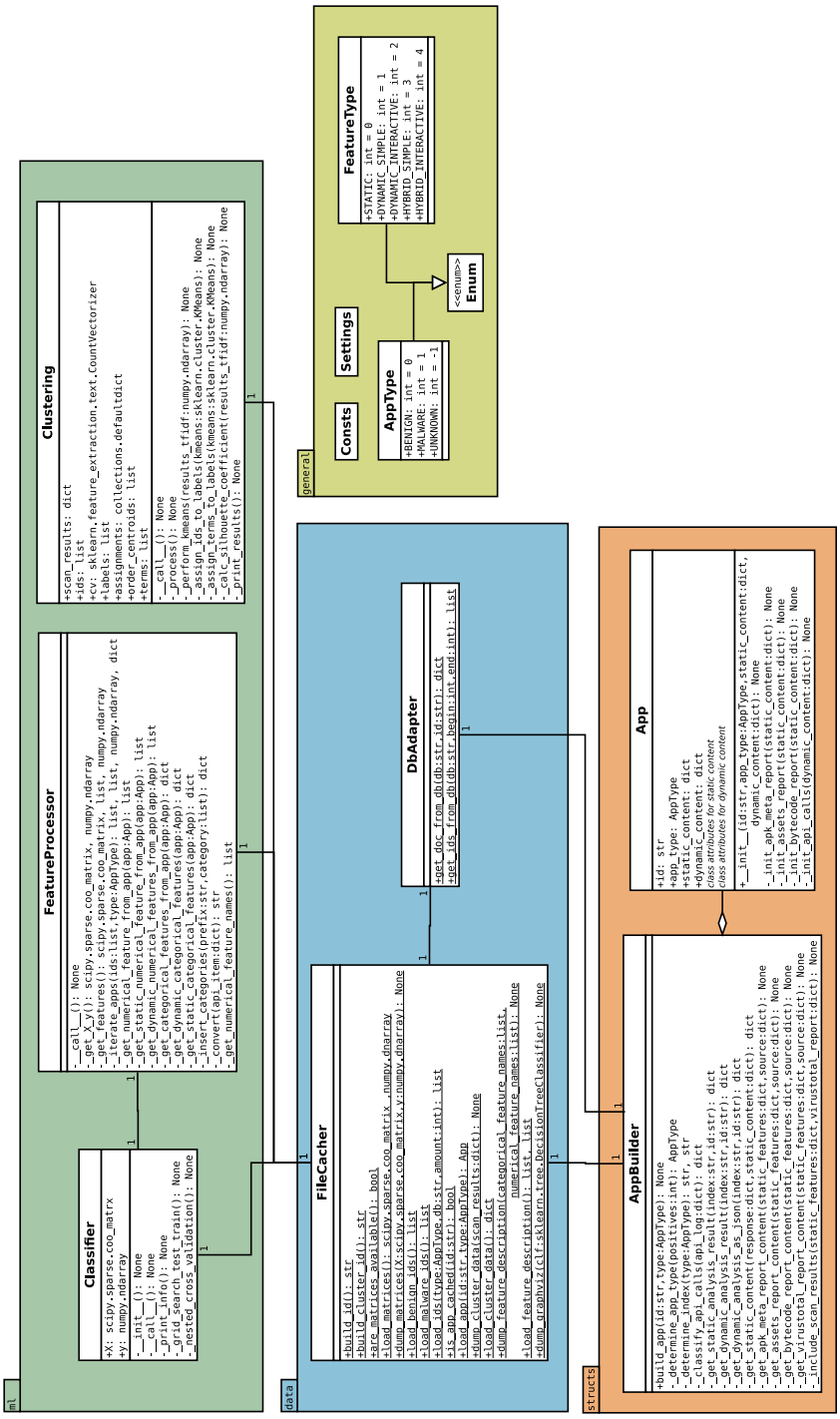

Figure B.2: Class diagram of the machine learning prototype

C Glossary

ABI	Application binary interface
ADB	Android Debug Bridge
AOSP	Android Open Source Project
AOT	Ahead-of-time (compiling)
API	Application Programming Interface
APK	Android Package
ARM	Advanced RISC Machines
ART	Android Runtime
BSD	Berkeley Software Distribution
CART	Classification and Regression Trees
CCP	Common Closure Principle
CLI	Command-line Interface
CRP	Common Reuse Principle
DAI	Distributed Artificial Intelligence
Dex	Dalvik Executable format
DIP	Dependency Inversion Principle
DNS	Domain Name System
FN	False negative
FP	False positive
GMS	Google Mobile Services
GPL	GNU General Public License
GPS	Global Positioning System
GSM	Global System for Mobile communications
HAL	Hardware Abstraction Layer
HTTP	Hypertext Transfer Protocol
ID3	Iterative Dichotomiser 3
IDC	International Data Corporation
IDE	Integrated development environment

IDF	Inverse document frequency
IMSI	International mobile subscriber identity
IoT	Internet of things
IP	Internet Protocol
IPC	Inter-process Communication
JDK	Java Development Kit
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LSA	Latent Semantic Analysis
MIME	Multipurpose Internet Mail Extensions
OCP	Open/Closed Principle
OHA	Open Handset Alliance
OS	Operating System
PPV	Positive predictive value
PUP	Potential Unwanted Program
REP	Reuse/Release Principle
REST	Representational State Transfer
RSA	Rivest-Shamir-Adleman
SDK	Software development kit
SHA	Secure Hash Algorithms
SQL	Structured Query Language
SSH	Secure Shell
SSL	Secure Sockets Layer
SVM	Support vector machine
SVC	Support vector classifier
TCP	Transmission Control Protocol
TDD	Test-driven development
TF	Term frequency
TLS	Transport Layer Security
TN	True negative
TP	True positive
TPR	True positive rate
UI	User interface
UID	Unique user ID
UML	Unified Modeling Language
URI	Uniform Resource Identifier

URL	Uniform Resource Locator
USB	Universal Serial Bus
VM	Virtual Machine
VPN	Virtual Private Network
XML	Extensible Markup Language

List of Figures

List of Tables

List of Code Snippets

Bibliography

[ABKLT16] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 468–471, 2016.

[Alp14] Ethem Alpaydin. *Introduction to Machine Learning, Third Edition.* MIT Press, 2014.

[And17a] Android Open Source Project. Content License. https://source.android.com/setup/licenses, 2017. [Online, last access on 10.05.2018].

[And17b] Android Open Source Project. The Android Source Code. https://source.android.com/setup/, 2017. [Online, last access on 11.05.2018].

[And18a] Android Developers. Android Debug Bridge (adb). https://developer.android.com/studio/command-line/adb.html, 2018. [Online, last access on 10.05.2018].

[And18b] Android Developers. Platform architecture. https://developer.android.com/guide/platform/index.html, 2018. [Online, last access on 10.05.2018].

[And18c] Android Developers. Requesting Permissions at Run Time. https://developer.android.com/training/permissions/requesting.html, 2018. [Online, last access on 13.05.2018].

[And18d] Android Open Source Project. ART and Dalvik. https://source.android.com/devices/tech/dalvik/, 2018. [Online, last access on 10.05.2018].

[And18e] Android Open Source Project. Security. https://source.android.com/security/, 2018. [Online, last access on 10.05.2018].

[And18f] Android Open Source Project. System and kernel Security. https://source.android.com/security/overview/kernel-security, 2018. [Online, last access on 10.05.2018].

[Apv18] Axelle Apvrille. Teardown of a Recent Variant of Android/Ztorg (Part 1). https://www.fortinet.com/blog/threat-research/teardown-of-a-recent-variant-of-android-ztorg-part-1.html, 2018. [Online, last access on 15.05.2018].

[ARF+14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. *ACM SIGPLAN Notices*, 49(6):259–269, 2014.

[ASH+14] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *Ndss*, volume 14, pages 23–26, 2014.

[AT05] Gediminas Adomavicius and Alexander Tuzhilin. Towards the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions. *IEEE Transactions on Knowledge and Data Engineering*, 17(6):734–749, 2005.

[BC04] Leo Breiman and Adele Cutler. Random forest - classification description. https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm, 2004. [Online, last access on 20.04.2018].

[Ben18] Michael Bentley. Lookout discovers new trojanized adware; 20K popular apps caught in the crossfire. https://blog.lookout.com/trojanized-adware, 2018. [Online, last access on 15.05.2018].

[Bre01] Leo Breiman. Random Forests. *Machine learning*, 45(1):5–32, 2001.

[Cat11] Rick Cattell. Scalable SQL and NoSQL Data Stores. *Acm SIDMOD Record*, 39(4):12–27, 2011.

[Che18] Check Point Software Technologies LTD. CuckooDroid Book. http://cuckoo-droid.readthedocs.io/en/latest/, 2018. [Online, last access on 12.05.2018].

[DDF+90] Scott Deerwester, Susan T Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. Indexing by Latent Semantic Analysis. *Journal of the American society for information science*, 41(6):391, 1990.

[Ela18] Elastic. Indices APIs. https://www.elastic.co/guide/en/elasticsearch/reference/current/indices.html, 2018. [Online, last access on 09.05.2018].

[Elg05] Ben Elgin. Google Buys Android for Its Mobile Arsenal. https://web.archive.org/web/20110205190729/http://www.businessweek.com/technology/content/aug2005/tc20050817_0949_tc024.htm, 2005. [Online, last access on 11.05.2018].

[F-S18a] F-Secure. Riskware:Android/SmsReg. https://www.f-secure.com/sw-desc/riskware_android_smsreg.shtml, 2018. [Online, last access on 15.05.2018].

[F-S18b] F-Secure. Trojan:Android/Plankton. https://www.f-secure.com/v-descs/trojan_android_plankton.shtml, 2018. [Online, last access on 15.05.2018].

[FASW15] Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Ainuddin Wahid Abdul Wahab. A review on feature selection in mobile malware detection. *Digital Investigation*, 13:22–37, 2015.

[FS08] Gereon Frahling and Christian Sohler. A fast k-means implementation using coresets. *International Journal of Computational Geometry & Applications*, 18(06):605–625, 2008.

[Goo18a] Google Inc. Android is enabling opportunity. https://www.android.com/everyone/enabling-opportunity/, 2018. [Online, last access on 09.05.2018].

[Goo18b] Google Inc. Application Fundamentals - Android Developers. https://developer.android.com/guide/components/fundamentals.html, 2018. [Online, last access on 09.05.2018].

[Goo18c] Google Inc. Application Security - Android Open Source Project. https://source.android.com/security/overview/app-security, 2018. [Online, last access on 09.05.2018].

[Goo18d] Google Inc. Dashboards. https://developer.android.com/about/dashboards/index.html, 2018. [Online, last access on 11.05.2018].

[Int18] International Data Cooperation. Smartphone OS Market Share, 2017 Q1. https://www.idc.com/promo/smartphone-market-share/os, 2018. [Online, last access on 11.05.2018].

[IRC+17] Fauzia Idrees, Muttukrishnan Rajarajan, Mauro Conti, Thomas M. Chen, and Yogachandran Rahulamathavan. PIndroid: A novel Android malware detection system using ensemble learning methods. *Computers & Security*, 68:36–46, 2017.

[Jai10] Anil K Jain. Data clustering: 50 years beyond K-means. *Pattern Recognition Letters*, 31(8):651–666, 2010.

[KR13] Rafał Kuć and Marek Rogoziński. *Mastering ElasticSearch*. Packt Publishing Ltd., 2013.

[LBP+17] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. Static Analysis of Android Apps: A Systematic Literature Review. *Information and Software Technology*, 88:67–95, 2017.

[Mal18a] Malwarebytes. Android/Adware.MobiDash. https://blog.malwarebytes.com/detections/android-adware-mobidash/, 2018. [Online, last access on 15.05.2018].

[Mal18b] Malwarebytes. Mobile PUP. https://blog.malwarebytes.com/threats/mobile-pup/, 2018. [Online, last access on 15.05.2018].

[Mar08] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, 2008.

[Mar17] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017.

[MG17] Andreas C. Müller and Sarah Guido. *Einführung in Machine Learning mit Python*. O'Reilly Media, Inc., 2017.

[PÇA16] Abdurrahman Pektaş, Mahmut Çavdar, and Tankut Acarman. Android Malware Classification by Applying Online Machine Learning. In *International Symposium on Computer and Information Sciences*, pages 72–80, 2016.

[PSK+17] Paolo Palumbo, Luiza Sayfullina, Dmitriy Komashinskiy, Emil Eirola, and Juha Karhunen. A pragmatic android malware detection procedure. *Computers & Security*, 70:689–701, 2017.

[PVG+11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[RC18] Idan Revivo and Ofer Caspi. CuckooDroid - Automated Android Malware Analysis. https://github.com/idanr1986/cuckoo-droid, 2018. [Online, last access on 09.05.2018].

[RJPD14] Leszek Rutkowski, Maciej Jaworski, Lena Pietruczuk, and Piotr Duda. The CART decision tree for mining data streams. *Information Sciences*, 266:1–15, 2014.

[Rov18] Rovo89, Tungstwenty. XPosed Module Repository. http://repo.xposed.info/module/de.robv.android.xposed.installer, 2018. [Online, last access on 10.05.2018].

[Sci18a] Scikit-learn. KMeans. http://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html, 2018. [Online, last access on 10.05.2018].

[Sci18b] Scikit-learn. RandomForestClassifier. http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html, 2018. [Online, last access on 10.05.2018].

[Sci18c] Scikit-learn developers. About us.

http://scikit-learn.org/stable/about.html, 2018. [Online, last access on 12.05.2018].

[SD02] Marina Skurichina and Robert P.W. Duin. Bagging, Boosting and the Random Subspace Method for Linear Classifiers. *Pattern Analysis & Applications*, 5(2):121–135, 2002.

[SK12] Justin Sahs and Latifur Khan. A Machine Learning Approach to Android Malware Detection. In *2012 European Intelligence and Security Informatics Conference (EISIC)*, pages 141–147, 2012.

[The18] The Apache Software Foundation. Apache Lucene Core. https://lucene.apache.org/core/, 2018. [Online, last access on 10.05.2018].

[TY17] Fei Tong and Zheng Yan. A Hybrid Approach of Mobile Malware Detection in Android. *Journal of Parallel and Distributed Computing*, 103(Supplement C):22–31, 2017.

[Ven18] Dinesh Venkatesan. Android.Lockscreen ransomware now using pseudorandom numbers. https://www.symantec.com/connect/blogs/androidlockscreen-ransomware-now-using-pseudorandom-numbers, 2018. [Online, last access on 15.05.2018].

[Vir18a] Virusshare administrator. VirusShare.com - Because Sharing is Caring. https://virusshare.com/about.4n6, 2018. [Online, last access on 10.05.2018].

[Vir18b] Virustotal. Virustotal - How it works. https://support.virustotal.com/hc/en-us/articles/115002126889-How-it-works, 2018. [Online, last access on 10.05.2018].

[WYZ15] Xiaolei Wang, Yuexiang Yang, and Yingzhi Zeng. Accurate mobile malware detection and classification in the cloud. *SpringerPlus*, 4(1):583, 2015.

[YSMM13] Suleiman Y Yerima, Sakir Sezer, Gavin McWilliams, and Igor Muttik. A New Android Malware Detection Approach Using Bayesian Classification. In *2013 IEEE 27th International Conference on*

Advanced Information Networking and Applications (AINA), pages 121–128, 2013.

[ZDYZ14] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1105–1116, 2014.

[ZJ12] Yajin Zhou and Xuxian Jiang. Dissecting Android Malware: Characterization and Evolution. In *Security and Privacy (SP) on 2012 IEEE Symposium*, pages 95–109, 2012.