Lukas Neugebauer BSc

APKCompare

What Has Changed in Android Applications?

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Dipl.-Ing. Johannes Feichtner

O.Univ.-Prof. Dipl.-Ing. Dr.techn. Reinhard Posch

Institute of Applied Information Processing and Communications (IAIK)

Graz, April 2019

AFFIDAVIT

Signator:	Lukas Neugebauer
Datum:	25.04.2019 21:19

Date

Signature

Abstract

Android applications are frequently updated for various purposes such as fixing security issues or updating the appearance. Therefore several versions of the same application might exist. Although new application versions often provide changelog information, users cannot be sure that the provided information describes all changes that have been made and that the modifications have been thoroughly implemented.

This thesis presents APKCompare, a comparison tool that analyzes two given "Android Application Package" (APK) files which contain the applications and visualizes similarities and differences between apps. APKCompare inspects resource files and the code of given applications and displays the comparison results in an easy to use web interface. The comparison approach is based on hash values, and as several features of classes are analyzed in multiple variants, the comparison results are accurate, even if code obfuscation techniques have been applied to applications. We evaluate APKCompare by comparing various real-world applications and analyze the plausibility and expressiveness of found differences between apps.

Keywords: Android, App Security, Code Comparison, Obfuscation, Static Analysis, Smali

Kurzfassung

Android-Applikationen werden häufig aktualisiert, sei es beispielsweise um Sicherheitslücken zu beheben oder um das optische Erscheinungsbild zu verändern. Aus diesem Grund liegen oft Applikationen in mehreren Versionen vor. Trotz der Änderungshistorie, welche oftmals über die Motive von Updates informiert, können Benutzer nicht sicher sein, dass alle Änderungen beschrieben wurden und die Änderungen gründlich implementiert wurden.

Diese Arbeit stellt die Analyse-Anwendung APKCompare vor. Sie analysiert zwei Android-Anwendungen im APK-Format und präsentiert Gemeinsamkeiten sowie Unterschiede der Applikationen. APKCompare untersucht sowohl Ressource-Dateien als auch den Code der Applikationen und stellt die Resultate des Vergleichs in einer benutzerfreundlichen Weboberfläche dar. Die Herangehensweise des Vergleichs basiert auf Hashwerten. Durch die Analyse von vielen Bruchteilen von Codeklassen in verschiedenen Varianten werden präzise Vergleichsresultate erzielt, selbst dann, wenn Techniken zur Unklarmachung der Applikationen (*obfuscation*) im Einsatz sind. Wir evaluieren APKCompare durch den praktischen Vergleich realer Anwendungen und untersuchen die Plausibilität und Aussagekraft der zwischen den Anwendungen gefundenen Unterschiede.

Schlüsselwörter: Android, Anwendungssicherheit, Code-Vergleich, Obfuscation, Statische Analyse, Smali

Acknowledgements

First of all, I would like to thank my advisor Johannes Feichtner for his support not only during my work on the practical part of this thesis but also during the writing process. Our regular meetings and his fast and precious feedback have provided me with the support I needed to finish this thesis. Additionally, I want to thank Keith Andrews for the LaTeX template[1], which I used to write this thesis, and the IAIK for providing an office room to work in. I also want to express my gratitude to my family and friends for supporting me throughout my years as a student at TU Graz. My special thanks are directed to Carolina.

[1]http://ftp.iicm.tugraz.at/pub/keith/thesis/thesis.zip

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Smartphones and other mobile devices are omnipresent in our daily life. People use them to communicate with each other, for work and entertainment purposes, but also to carry out financial operations and other tasks. A large proportion of smartphones is powered by the Android operating system, which allows users to install applications on their device. Such applications can be installed from the official market, named *Google Play Store*, but also from unofficial markets and other sources.

Many Android applications exist in several different versions. Whenever a new version of an Android application is published, release notes might or might not be available. In fact, the provided release notes are not necessarily accurate. For example, if a publisher states that a known security-relevant issue has been fixed with a recent update, users have to put their trust into that statement. They can, however, not be sure that the mentioned changes have actually been made, and they can also not verify if additional modifications, for example the positioning of advertisements, have been carried out.

There are also many Android applications that originate from other applications by copying and slightly modifying them. Such, so-called, *repackaged applications* are often produced to get applications to work on other devices than originally intended, for example. However, it is also widespread that malware or advertising functionality is added to popular applications. Repackaged applications are usually distributed over third-party channels, but they can also be found in the *Google Play Store*.

Whenever a new version of an existing Android application is published, whenever an application appears to be a repackaged one, and also when a user is simply interested in similarities and differences of two applications, a detailed comparison is necessary. However, comparing two given Android applications manually is a very hard and time-consuming task and it also requires special foreknowledge of the decompilation process of Android applications.

In this thesis, we introduce *APKCompare*, a Java-based tool that compares two given Android applications and presents the results in an easy to use web interface. By using this comparison tool, only the parts that are classified as changed ones have to be interpreted in order to ascertain the differences of the given applications. This provides security experts and programming-minded Android users with a massive speedup when comparing two versions of an application, two seemingly repackaged apps, or arbitrary other applications with each other.

1.1 Problem Statement

Detecting similarities and differences between two Android applications is a challenging task because of several reasons. First of all, when comparing the code of two applications with each other, we are interested in the described behavior, not in details that are not relevant for execution. Moreover, the source code of a large proportion of available Android applications is obfuscated when the project is built, for example by replacing names of classes, fields, and methods with vacuous names. This complicates analyzing and comparing the used code on purpose. As nearly all names, including file names, are vacuous in obfuscated classes, we target a solution that does not rely on names but the code structure.

In addition to the code, used resources, such as layout definitions and images, have an immense impact on applications. Therefore, we are not solely interested in the detection of similarities and differences of the code, but also in changed resources. Also, even small changes in single resource or code files can impact large parts of the built APK file. However, with the comparison results of APKCompare, it should be possible to detect the exact changes that have been made.

1.2 Research Questions

In the following, we present the core research questions that are addressed in our work:

- How can we detect unchanged/renamed/modified/added/deleted resource files?
- How can we succeed in detecting unchanged/renamed/modified/added/deleted classes?
- How can we precisely detect specific changes, such as methods that have been moved from one class to another, or fields that have been modified?
- How well does the comparison work for applications that have been obfuscated?
- How can we visualize detected changes in a clear and intuitive way, even if many modifications have been made?

In order to answer the above-listed research questions, we have to compare applications with APKCompare and discuss the results within the evaluation process. By comparing multiple versions of an intentionally modified application, we can validate if all modifications are listed in APKCompare's comparison results. However, also applications where no source code is available have to be compared against each other so that we can prove that our tool only uses information that is extracted from the applications. To test the tool under realistic conditions, we need to compare different versions of real applications against each other, such that the provided release notes can be verified. Additionally, we need to compare a repackaged application against the original one with APKCompare.

1.3 Approach

All resource files, like images and layout definitions, as well as the code classes that describe the Android application, are provided in so-called "Android Application Package" (APK) [1] files. Files in the APK format can be installed on devices that run the Android operating system. As all the information that describes an Android application is available in the APK file, such files are well qualified to compare the respective applications. When talking about the comparison of APK files, we distinguish between the comparison of resource files and the comparison of code. For both processes, we have to disassemble the APK files in a first step. To do that, we use a tool named *Apktool*.

For the process of comparing resource files, our approach is based on the idea of identifying identical files by checking the files' hash values for equality. Resource files with equal hash values can be categorized as unchanged files, while files where the corresponding hash value cannot be found in the other application have to be listed as modified files or as files that are only available in one of the two given applications.

The code comparison approach is based on the idea of comparing hash values as well. In a first step, we detect unchanged classes by identical hash values. Then, we split the not matched classes into several smaller parts, and check the hash values of the resulting files against each other in order to match similar classes. In addition, we compare classes with all identifier names replaced by placeholders against each other, which allows us to detect similar classes even if identifiers have been renamed or obfuscation techniques have been used. To detect specific changes, for example a method that was moved from one class to another, or code blocks that have been deleted or changed, we extract methods and basic blocks from classes that have not been identified as matches in previous steps and try to identify what has happened to them.

We visualize the results of our tool such that a quick inspection of the results already provides information about the similarity of applications compared with each other. Resource files and classes that have been identified as unchanged or similar are listed in appropriate lists, and a side by side comparison view, like it is known from common file comparison tools, is used to visualize the specific changes that have been detected.

1.4 Outline

The next chapters of this thesis are structured in the following way:

In Chapter 2, we present background information that is of particular importance for the topic of this thesis. First, we have a look at the operating system Android and at APK files, which are used to install applications on the Android system. We also discuss the structure of APK files and emphasize file contents of special relevance. Then, one section is dedicated to files of the DEX format and the runtime environment that is used by the Android system. Also, the Smali language and the tool *Baksmali*, which we have used for several purposes in our work, are introduced in Chapter 2. In Section 2.4, we explain why the code is obfuscated in many Android applications, and we have a general look at code obfuscation techniques. We also describe the tool *ProGuard*, the most popular tool to obfuscate Android apps, in this chapter. Later, a section deals with the topic of hashing files and Merkle trees, which are concepts of high importance for our work. Also in this chapter, we discuss the idea of basic blocks of code and illustrate this concept by giving examples.

Chapter 3 gives an overview of the related research topics. This chapter is divided into three sections. Each section discusses different topics of related work, namely the comparison of APK files, the detection of cloned Android applications, and deobfuscation techniques that can be used on obfuscated Android apps.

Then, we will discuss the approach of our work in Chapter 4. First, we specify the requirements that should be fulfilled by a tool which can be used to compare APK files against each other, and also define prerequisites and constraints. Then, we explain our strategy to find differences and similarities in APK files and introduce the tool we have implemented. We give detailed information about the approach of comparing resources and code, and also discuss the idea behind our visualization strategy. For both comparison topics, resource comparison and code comparison, we describe our ideas in detail and also provide figures that help to become acquainted with our strategy.

In the following chapter, we have a detailed look at the design and implementation decisions and also at tools that have been used to implement APKCompare. To be more accurate, we dedicate one section of Chapter 5 to the extraction of APK files, another one to the discussion of hash algorithm candidates and the selection of a proper algorithm and the rest to other implementation details. Section 5.3 provides detailed information about the Smali disassembler *Baksmali* and the additional options that we implemented in order to fulfill our needs. In the following sections of this chapter, we provide several topics like rewriting DEX files, the detection of basic blocks, and implementation details of the web view.

Chapter 6 treats the evaluation of our tool. We assess APKCompare by comparing several APK files and checking if the expected changes are listed in the results of our tools. To do that, we build multiple versions of an open source Android application and compare the resulting files as a first step. Then, we have a look at comparison results of two versions of well-known applications and at results of a comparison of an unmodified and a repackaged application. We also discuss limitations of our tool in this chapter and provide a general discussion of the evaluation results.

Finally, Chapter 7 concludes our work. We recall the problem of the comparison of APK files and summarize the contribution of our work to solve that problem. Also, we provide ideas for improvement in this chapter.

Chapter 2

Background

In this chapter, we discuss background information that is important for the topic of this thesis. In Section 2.1, the operating system Android is introduced. We highlight on which devices Android can operate and how Android applications can be obtained. Additionally, we have a more detailed look at Android applications and the content of *Android Application Package* files.

Section 2.2 deals with files of the DEX format and the so-called *Android Runtime*. We describe where we can find DEX files, why the files are needed, and where the differences between *Android Runtime* (ART) and *Dalvik Virtual Machine* (DVM) lie. Finally, we give a brief overview of the format of DEX files.

Next, in Section 2.3, we describe the Smali language in general, and we have a look at the syntax of .smali files, which can be generated from DEX files by a tool named *Baksmali*. Additionally, we provide a comparison of the two register naming schemes in Smali, which we refer to as *v-naming scheme* and *p-naming scheme* . Furthermore, we look at the basic structure of Smali classes.

Later, in Section 2.4, code obfuscation techniques are discussed. We explain why code is obfuscated by many developers and how it can be done. Then, we focus on the most widely used tool for obfuscation of Android applications, *ProGuard*. We detail the four steps *ProGuard* can perform and highlight which of them are important for our thesis. To illustrate the results of obfuscation mechanisms, we end the chapter by providing an example of an obfuscated method in Smali code.

In the next section, Section 2.5, we explain the concept of hashing. This section of the thesis serves to illustrate that we solely use hash functions to quickly determine if files are equal or not and that some properties of such functions are not essential for our work. We also introduce the concept of Merkle trees in this section.

Finally, in Section 2.6, we give a short overview of the concept of basic blocks of code and illustrate the basic blocks of a sample method. To round off the section, we describe why basic blocks are of interesting for code analysis.

2.1 Android / Android Applications

The Android operating system (OS) was initially released in 2008. Since then, numerous versions and updates have been released. Android operates on many devices such as watches, TVs and even cars, but is most popular for being the most widely used operating system for mobile phones. According to IDC, 86,6% of all smartphones which have been sold in the third quarter of 2018 run this open source operating system[1].

One of the most interesting aspects and presumably an important reason for the significant success of Android is the fact that individual applications can be run on Android devices. Applications can be installed from so-called markets. The official market for Android, *Google Play Store*, allows users to browse for applications which are either free or require payment. According to statista, 2.6 million apps have been available on *Google Play Store* by December 2018[2]. Android applications, no matter if installed from *Google Play Store* or other sources, run in sandboxed environments. This means that apps are separated from each other and do not share memory.

A majority of Android applications have been developed in Java, but it is also possible to use languages like Kotlin[3] or C#[4] to develop applications for Android. No matter what language was used for development, an *Android Application Package* file with the extension .apk has to be created to distribute and install an Android application. APK files hold all resources, code and further information that describe the application, packed in zip format-type.

A more detailed look at **APK** files reveals the **structure** of such files. In the list provided below, we highlight some of the contents of special relevance for this thesis:

- **res**
  The *res* directory contains those resource files of an application which have not been precompiled. Images in different resolutions, various color definitions, and audio files are part of this directory. Additionally, interface layouts defined in XML format as well as strings in multiple languages can be found in this directory.

- **resources.asc**
  resources.asc is a file that contains precompiled resources like strings and binary XML, which are used by the application.

- **assets**
  This directory contains application assets which can be loaded at runtime by using the AssetManager.

- **AndroidManifest.xml**
  This file in the Android binary XML format contains some fundamental information about the application. The name and the version of the app, as well as access rights, are described in *AndroidManifest.xml*. All activities, the main activity of an application and the permissions of the app are additional parts of this file.

[1]https://www.idc.com/promo/smartphone-market-share/os
[2]https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store
[3]https://developer.android.com/kotlin/
[4]https://docs.microsoft.com/en-us/xamarin/android/

- **classes.dex and other .dex files**
  The *classes.dex* file can be found in every valid APK file, some applications even use multiple files in the DEX format. These files contain all classes of the application, compiled to Dalvik bytecode. In Section 2.2, we provide a more detailed description of DEX files and explain how they can be interpreted.

2.2 Android Runtime / DEX Format

Java compilers produce platform-independent byte code, which can be interpreted by virtual machines. This is usually done by the *Java Virtual Machine* (JVM). Even though most devices that run the Android OS nowadays have powerful hardware, another virtual machine which was optimized to work with constraints like little RAM and slow CPUs is deployed by Android. This virtual machine is named *Android Runtime* (ART) [2, pages 60–62]. ART is the successor of the *Dalvik Virtual Machine* (DVM) and has come with Android since Version 4.4.

While both ART and DVM deal with .dex files that contain Dalvik bytecode, JVM deals with .class files. Files in the .dex format are generated by the Dalvik compiler *dx* by postprocessing the .class files which have been generated by the Java compiler beforehand and are typically drastically reduced in size because of an effective file structure and the elimination of repeatedly declared values. This compilation process is visualized in Figure 2.1 [3].

One of the main differences in JVM and ART is the usage of registers. While Dalvik byte code and therefore ART is register-based, class files work stack-based. This makes no distinction for Android developers, typically they will not even notice the use of a stack-based virtual machine, but it will be interesting when having a look at the interpreted byte code or a disassembled version of it, as we can see in Chapter 2.3.

The specific structure of Dalvik Executable files is not of high relevance for this thesis, but it is important to know that manipulations on .dex files, like the renaming of classes or methods, are possible as long as the file structure stays valid. For a very detailed explanation of the format we recommend having a look at the official Android source website[5].

[5]https://source.android.com/devices/tech/dalvik/dex-format.html

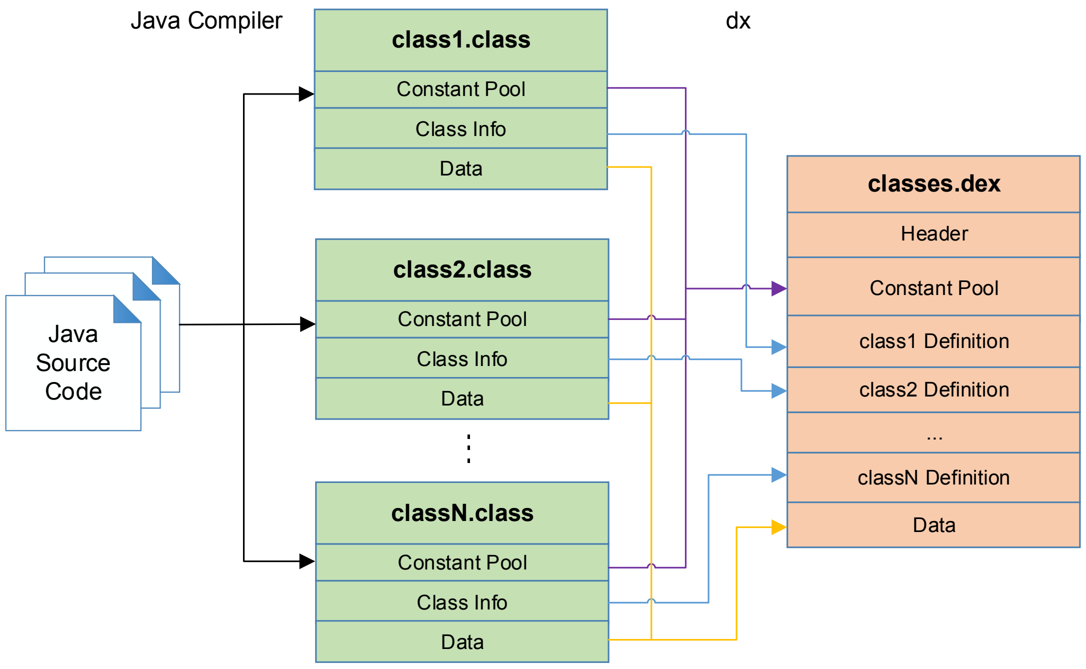


Figure 2.1: Compilation of Android applications [Figure created by [3]]

2.3 Smali Language / Baksmali

Smali code is the human-readable equivalent of Dalvik bytecode. This means that Smali code is not interpreted by Android or any other system, the purpose of the Smali language lies in easier readability. The entire Dalvik instruction set can be described in the language Smali, whose syntax is loosely inspired by the syntax of Jasmin[6] and Dedexer[7] [4].

Files in Smali language can be extracted from DEX files with the tool *Baksmali* [4]. This tool allows to disassemble files of the DEX format and to generate one .smali file for each class with respect to the original Java source code structure. Nested classes are written to separate .smali files as well. Like Davlik byte code, Smali code is register-based. Independent of the data type, **registers** are always 32 bits in size. For types that need 64 bits, namely *Long* and *Double*, two registers are needed. A fresh set of registers is used for every method.

For registers in Smali code, two naming schemes are supported. One scheme, we call it *v-naming scheme*, names registers like it is done in Dalvik byte code. In this scheme, all registers start with the character *v*, regardless of whether a register describes a method argument or not. In the v-naming scheme, arguments have to be placed in the last registers. Next to the fact that argument registers cannot be identified as such at first glance in this naming scheme, there is another unpleasant issue: If someone wants to edit a method manually and has to introduce a new register, all parameter registers have to be renamed as they have to be stored in the last register of the method.

For reasons of convenience, Smali also supports another register naming strategy, we call it *p-naming scheme*. With this scheme, the first method parameter is always stored in register *p0*, the

[6]http://jasmin.sourceforge.net
[7]http://dedexer.sourceforge.net

second parameter register is named *p1* [8] and so on, while local registers start with the character v here as well. When introducing another register in code with this naming strategy, there is no more need to adapt the names of all parameter registers. A comparison of the described naming schemes can be seen in Table 2.1, the example describes the registers of a method with two parameters and two local registers.

Because of better readability, we prefer the *p-naming scheme*. Therefore, we will use this scheme when displaying Smali code snippets in our thesis.

v-naming scheme	p-naming scheme	
v0	v0	first local register
v1	v1	second local register
v2	p0	first parameter register
		[Always holds the class object ('this') for non-static methods]
v3	p1	second parameter register

Table 2.1: Comparison of Smali register naming schemes.
The example refers to a method with two parameters and two local registers.

In **Smali syntax**, many instructions expect the destination register as the first parameter. Examples for that can be found in Listing 2.1. In the first line, an object of the type `java.io.PrintStream` is read from a static field and saved to register `v2`. Line two shows the `const-string` instruction, a reference to the string `Hello Android!` is saved to register `v3` here. In the third line of the snippet, we can see an `invoke-virtual` instruction, the `println` method of the object in `v2` is called here with the string in `v3` as parameter. The `V` at the end of the line indicates that the return type of the println method is void. An overview of the type descriptions in Smali language and their Java equivalents can be found in Table 2.2. For a more detailed insight in Smali syntax, we suggest having a look at the Wiki page[9] of the Smali [4] project.

```
1 sget-object v2, Ljava/lang/System;->out:Ljava/io/PrintStream;
2 const-string v3, "Hello Android!"
3 invoke-virtual {v2, v3}, Ljava/io/PrintStream;->
    println(Ljava/lang/String;)V
```

Listing 2.1: Smali syntax example.

The basic **layout** of **.smali** files is pictured in Listing 2.2. *Baksmali* creates one file per class, each matches roughly the structure of Java files. In the first line of the snippet, we can see the declaration of the class, this is done via modifiers like `public` or `private` and a fully qualified class name. Lines 2-4 define the super class as well as the interfaces the class implements. Next, annotations are written if there are any. After all class members have been declared, direct and

[8] If a parameter is of type Long or Double, two registers are needed for storage, and the names for further parameter registers will be increased by one, therefore.

[9] https://github.com/JesusFreke/smali/wiki/TypesMethodsAndFields

virtual methods are introduced in the file. Direct methods are of type static, private or constructor, while all other methods are virtual ones[10]. For every method, a `.method` keyword indicates the start and the following `.end method` keyword indicates the end of the declaration. The value next to the keyword `registers` tells how many registers, including parameter registers, are used by the method. For an overview of available types and their representation in Smali, have a look at Table 2.1.

```
.class modifiers... Lfully/qualified/Name;
.super Lfully/qualified/SuperclassName;
.implements Lfully/qualified/InterfaceName1;
.implements Lfully/qualified/InterfaceName2;

# annotations
...

# static fields
.field modifiers... staticFieldName1:type
.field modifiers... staticFieldName2:type

# instance fields
.field fieldName1:type
.field fieldName2:type

# direct methods
.method modifiers... directMethodName1(Argument1Type...ArgumentNType)
    return-type
    .registers m

    instructions...
.end method

# ... (more direct methods)

# virtual methods
.method modifiers... virtualMethodName1(Argument1Type...ArgumentNType)
    return-type
    .registers m

    instructions...
.end method

# ... (more virtual methods)
```

Listing 2.2: Structure of a .smali file.

[10]https://source.android.com/devices/tech/dalvik/dex-format.html

Dalvik / Smali Syntax	Corresponding Java Type
V	void (only used for return types)
Z	boolean
B	byte
S	short
C	char
I	int
J	long (64 bits)
F	float
D	double (64 bits)
Lfully/qualified/Name;	fully.qualified.Name
[	Array

Table 2.2: Mapping of type descriptors in Dalvik / Smali
Regarding entry 'Lfully/qualified/Name;': The leading L indicates an object type, the terminating ';' indicates the end of the object name. Everything in between defines the package and the name of the object.
Regarding entry '[': Arrays always start with the character '[', followed by the type of the array, for example '[I' for an array with ints. '[[C' is the equivalent of char[][] in Java.

2.4 Obfuscation / ProGuard

Obfuscation describes the idea of modifying a program in a way that the functionality stays the same while making reverse engineering harder. To obfuscate Android applications, modifications of the source code and modifications on byte code level are possible. Such modifications, regardless if on source code or byte code level, are relevant for our thesis, as we want to distinguish between changes that have actual impact and changes that are only a result of obfuscation. An example of the latter is the simple renaming of a variable, which our tool should detect as unchanged code. This means that obfuscation-independent techniques have to be found for code comparison, Section 4.4 deals with this topic in detail. There are several approaches and a considerable amount of tools to obfuscate code, but as *Android Studio*[11], the official IDE for Android, is shipped with the code obfuscation tool named *ProGuard*[12], we will have a look at the functionalities of that tool.

ProGuard is a tool that allows shrinking, optimization, obfuscation, and preverification of Java code[13]. All of the steps are optional, meaning that a developer can define which of the steps should be executed and which ones should be omitted. A brief explanation of the approaches gives an overview of the work of *ProGuard*:

Shrinking code is performed by *ProGuard* by analyzing the control flow of code and removing unused classes, fields, methods, and attributes. Because of the reduced code, the size of the resulting APK file can also be smaller and the time needed to launch the application can be reduced. Moreover, resources that are not used by the application will not be packed into the resulting APK file, which also leads to a reduced archive size.

[11]https://developer.android.com/studio
[12]https://www.guardsquare.com/en/products/proguard
[13]https://www.guardsquare.com/en/products/proguard/manual/introduction

The goal of **optimization** is to reduce the runtime of the code without changing the functionality. Optimizations like removing unused parameters or inlining of methods are done in this step. As optimizations are already done by the DEX compiler and also because of problems that might occur after executing *ProGuards* optimization step, this step is turned off for Android applications by default and is therefore not of high importance for our thesis.

In the **obfuscation** step, classes, class members and methods are renamed by *ProGuard*. An example of an obfuscated method (shown in Smali code) can be seen in Listing 2.3. As we can discern, identifiers which are not part of the application but are provided by the system, for example packages in `java/*` and the majority of the packages in `android/*`, cannot be obfuscated by *ProGuard* and are therefore used in the unmodified way. Some other members and methods are typically not obfuscated as well, for example getters and setters of Views, as they might be needed in unmodified form so that animations can still work. According to Bichsel et al., "ProGuard obfuscates 86.7% of the program elements on average" [5]. Developers can explicitly define obfuscation rules in a *ProGuard* configuration file.

In the final step, **preverification**, preverification information is added to the classes. Such information is needed for Java 6 and higher, but since it is irrelevant for DEX compilers[14], *ProGuard's* preverification step is useless for Android applications and therefore not relevant for our thesis.

```
.method public b(Landroid/content/DialogInterface;I)V
    .registers 4

    iget-object v0, p0, a:La/a/a/a/a;

    invoke-virtual {v0}, La/a/a/a/a;->d()Landroid/app/AlertDialog;

    move-result-object v0

    invoke-virtual {v0}, Landroid/app/AlertDialog;->show()V

    return-void
.end method
```

Listing 2.3: A method in Smali language.
The tool *ProGuard* was used for obfuscation.

2.5 Hashing

Hash functions are one-way functions that can be used to map input of arbitrary size to a value of fixed size. The output of such functions is named hash value. Hash functions are deterministic, which means that the same input always leads to the same resulting value. Because of that fact, hash functions are often used to determine hash values of files and these values are used to check if two files are equal in content or not. For popular hash functions, it is sufficiently safe to say that two files have the same content if they have the same hash value. The approach of

[14]https://www.guardsquare.com/en/products/proguard/manual/examples#androidactivity

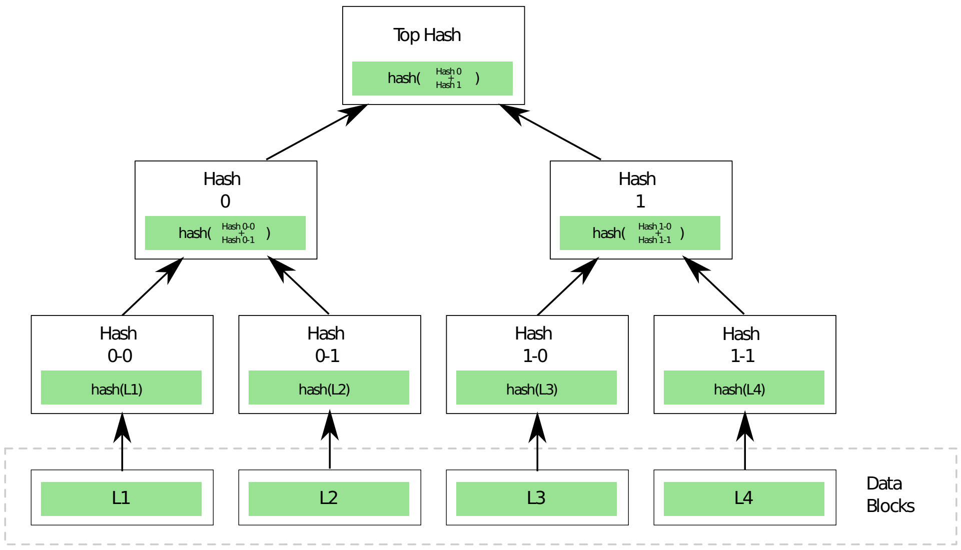


Figure 2.2: Merkle tree concept
[Figure created by Wikipedia user Azagha. (https://commons.wikimedia.org/wiki/File:Hash_Tree.svg)]

comparing file hashes instead of whole files is widely used, as hash values are typically much smaller in size than files.

Moreover, hash functions are often used in cryptography topics, where they have to fulfill some requirements such as that it should be computationally infeasible to find two distinct inputs that hash to the same value. It is strictly advised not to use hash functions that are vulnerable against known attacks, like the MD5 algorithm [6], in areas where cryptography plays an important role. Within the scope of this thesis, the fulfillment of cryptographic requirements of hash functions is not critical, we solely use hash values to determine if files are equal or not.

To compare large data structures, for example directories filled with many files and subfolders, it is not necessary to find matching hash values for all of the files. Instead of that, **Merkle trees** that represent the data structure can be built. In such hash trees, as they are also termed, each leaf node is tagged with the hash value of its content, and each node with children is tagged with the hash value of the concatenation of its child tags. This results both in the fact that editing a leaf impacts the hash values of all ancestor nodes and that descendant nodes do not have to be checked against each other if the hash values of two nodes are equal. Figure 2.2 shows the concept of a Merkle tree. In the context of directory comparison, files can be seen as leaves and their parents represent folders.

2.6 Basic Blocks

In compiler construction, a basic block is a code sequence that has exactly one entry point and one exit point. This means that every instruction of a basic block is executed exactly once and that the instructions are executed in unmodified order.

Formally spoken, a sequence of instructions forms a basic block if the following two definitions are fulfilled:

- Every instruction always executes before instructions in later positions are executed.
- Between the execution of two instructions in the sequence, no other instruction is executed.

Basic blocks qualify well for code analysis, for example to find redundant code or to determine the similarity of two methods. The latter is exactly why basic blocks are relevant to the topic of this thesis. Our tool is capable of identifying such blocks in Smali methods and of finding basic blocks that exist in both provided APK files. The gained knowledge of multiple used basic blocks can be used to classify two methods as potentially similar. Basic blocks can consist of any number of instructions, even of just one instruction, and as small basic blocks are not extraordinarily convincing when looking for similar methods, our tool only uses basic blocks with a definable amount of instructions in this step. More detailed insight on how we use basic blocks in this thesis is provided in Section 4.4, while an illustration of how our tool detects such blocks can be found in Section 5.5.

Listing 2.4 presents the four basic blocks of a method in the language Smali. For each block (line 4-10, 12-14, 16-17 and 19-24) it is guaranteed that all instructions are executed in order and exactly once. In this code, line 4 is a basic block leader because it is the first instruction of the method. Line 12 leads a basic block as it is the first instruction line after the jump instruction on line 10. The lines 16 and 19 lead basic blocks because they are target lines of goto/jump statements.

```
 1 .method private createLayout(Lch/bailu/aat/map/MapViewInterface;
       Lch/bailu/aat/views/preferences/VerticalScrollView;
       Lch/bailu/aat/views/MainControlBar;)Landroid/view/View;
 2     .registers 5

 4     iget-object v0, p0, acontext:Lch/bailu/aat/activities/AbsGpxListActivity;
 5
 6     invoke-static {v0}, Lch/bailu/aat/util/ui/AppLayout;->isTablet(
           Landroid/content/Context;)Z
 7
 8     move-result v0
 9
10     if-eqz v0, :cond_0

12     invoke-direct {p0, p1, p2}, createTabletLayout(
           Lch/bailu/aat/map/MapViewInterface;
           Lch/bailu/aat/views/preferences/VerticalScrollView;)Landroid/view/View;
13
14     move-result-object v0

16     :goto_0
17     return-object v0

19     :cond_0
20     invoke-direct {p0, p1, p2, p3}, createMvLayout(
           Lch/bailu/aat/map/MapViewInterface;
           Lch/bailu/aat/views/preferences/VerticalScrollView;
           Lch/bailu/aat/views/MainControlBar;)Landroid/view/View;
21
22     move-result-object v0
23
24     goto :goto_0

25 .end method
```

Listing 2.4: Basic blocks of a method that is shown in Smali representation

Chapter 3

Related Work

In this chapter, we give an overview of related research topics and focus on aspects of special importance for this thesis. Section 3.1 covers the field of APK comparison. We present two topics that are closely related to our work and describe similarities and differences. In Section 3.2, we discuss topics to detect cloned Android applications and topics that deal with code fingerprinting, both work with similar strategies. Finally, in Section 3.3, we briefly summarize the work of Bichsel et al., which deals with the question of how to deobfuscate Android applications.

3.1 APK Comparison

Finding automated ways of comparing APK files is an important topic for the stated reasons. Therefore, unsurprisingly, other researchers have already investigated that topic. We want to highlight two approaches in this section: First, we describe the approach of Anthony Desnos (2012) in detail and highlight similarities and differences to our work. Then, we follow the same procedure for a thesis written in 2017 by Li et al.

The paper "**Android : Static Analysis Using Similarity Distance**" [7], which was authored by Anthony Desnos, presents an algorithm which is largely based on a similarity distance called the *Normalized Compression Distance* (NCD). The introduced algorithm works on Dalvik byte code level and returns a change indicator that can be used for several applications. The author mainly discusses how the algorithm is used to identify a value that describes how similar two apps are. This value can be used as an indicator of potentially repacked Android applications, but it is also used by the author to evaluate the efficiency of obfuscation tools. Although the author's focus is not placed on the comparison of two versions of an application, this topic which is of special interest for our thesis is still discussed in his work. The main steps of the algorithm are as follows [7]:

1. "Generate signatures for each method"
2. "Identify all methods which are identical"
3. "Identify all methods which are partially identical by using NCD"

As we can see, the algorithm works on method level. Therefore, also the detected elements are methods, namely identical ones, similar ones as well as new and deleted methods. For method comparison, a checksum that is based on a modified instruction sequence is used. This checksum is based on a subset of the method instructions as information like registers or the offset of jump instructions are removed. Methods with equal checksum are considered as equal and removed from further comparisons. For remaining methods, signatures are created. These signatures include the control flow graph of the methods, but also for example strings and packages, and can be used to calculate the NCD between methods. Methods with a short distance are classified as similar, remaining methods as new or deleted.

With this algorithm, the author calculated change indicators in the range of 0.0 to 100.0 that describe how similar two applications are. In addition to that, Desnos introduced a way to identify particular changes between two versions of an application: After splitting methods into basic blocks and identifying identical blocks by using the NCD, added and removed instructions of similar basic blocks are extracted by using the longest common subsequence (LCS) algorithm.

The author describes that basic block instructions are transformed into strings and the LCS algorithm is used to determine the added and removed instructions. To test his approach of comparing two versions of an application, the versions *1.0.0.831* and *1.0.0.983* of the *Skype* app have been compared against each other. Their tool, namely *androsim.py*, which was part of the *AndroGuard*[1] project but is not anymore, identified more than 8,000 identical methods, but also 165 similar ones as well as 14 new and 7 deleted methods. The 165 similar methods were further investigated, and because of the changes in basic blocks, the author could show that a bug relating to file modes was fixed with the update.

While the author of the described paper solely used method byte code to detect similarities and differences of applications, we have a look at the whole content of APK files for comparison, meaning also resources in addition to the whole application code in Smali format. A big benefit in comparison on byte code level lies in easier adaptability for other languages. However, we decided to compare on Smali level as it is easier to read and still describes the full functionality. Moreover, we do not use checksums only for a quick determination of similar code parts and other techniques like the NCD afterward, but gradually reduce the code to compare and determine via hashes if parts are equal or not. Both approaches split methods into basic blocks to obtain better resulting matches.

A more recent study on the topic of the comparison of APK files was done by Li et al. in 2017 [8]. In their paper "**SimiDroid: Identifying and Explaining Similarities in Android Apps**", they introduce an open source framework[2] that allows the pairwise comparison of Android applications on multiple levels. Like us, the authors of the work did not only aim at finding out if two applications are similar or not, but they also tried to present the actual similarities and differences. According to the authors, "*SimiDroid* is designed as a plugin-based framework integrating various comparison methods" [8] that works in the following three steps:

1. Extraction of necessary features
2. Generation of a similarity profile
3. Mining of changes, based on the similarity profile

In the first step, characteristic key-value pairs are extracted for both given applications. The content of such pairs depends on the plugin, but independent from the actual plugin those pairs

[1]https://github.com/androguard/androguard
[2]https://github.com/lilicoding/SimiDroid

are always used in step two, the similarity comparison. To generate a similarity profile, four metrics are used. Items are classified as

- **identical**, when the exact same pair was extracted from both applications.
- **similar**, when a key exists in both applications, but the corresponding value is different.
- **new**, when a key which exists in the second application does not exist in the first application.
- **deleted**, when a key which exists in the first application can not be found in the second application.

Independent of the concrete plugin, a similarity score, which tells how similar two given applications are, is calculated based on these metrics [8]:

$$similarity = max\left\{\frac{identical}{total-new}, \frac{identical}{total-deleted}\right\}$$

where

$$total = identical + similar + deleted + new$$

In the last step, particular changes are determined by *SimiDroid*. This process is based on the extracted similarity profile for every plugin, however, the specific implementation is plugin dependent. Plugin developers can provide code that is executed before the similarity comparison, for example the exclusion of common libraries, as well as code that is executed afterward, like inferring changes between similar methods.

There are currently three plugins that can be used with *SimiDroid*. The **RPlugin** is resource-based, meaning that resources are used to detect similar applications. To do that, the key/value pairs are constructed with the resource path as key and a hash value of the file content as value. A similar approach is followed by the **CPlugin**, which is component-based. The key/value pairs are built with information that can be found in the *AndroidManifest.xml* file, for example component names, but also other component capabilities like *action* and *category*. The third plugin, **MPlugin**, can be used to detect identical, similar, new and deleted methods. Method signatures as well as abstract representations of statements are used to map key/value pairs by this plugin, while the mentioned representation is derived from the statement's type, for example `invoke-statement`. Because of that, basic obfuscation approaches like identifier renaming do not affect the outcome. Also, constant strings and numbers are used as comparison features by this plugin. In addition to the already described work, the authors of the thesis also came up with specified analysis implementations that identify changes like new method calls or the replacement of constant numbers and strings.

The approach of Li et al. is similar to ours in many aspects, like the separation of comparison on resource and code level and the fact that hash values are used to determine unchanged resources. However, there are also some differences in similarity detection. While we appreciate the plugin concept of Li et al., we see an advantage in the overall comparison on resource and code level for a pair of applications, like we implemented it in our tool. There is also a mentionable difference when it comes to code comparison. While the *MPlugin* extracts features at the *Jimple* code level [9], we use Smali code. Both approaches found mechanisms that are resilient to obfuscation techniques.

3.2 Clone Detection

Besides the introduced theses that deal with the question of how to compare APK files against each other, there is also a considerable amount of research papers that deal with topics that are related to ours, namely how to detect repackaged applications and how to generate fingerprints from code. We want to highlight some of these works.

Chen et al. [10] extract geometry characteristics out of dependency graphs of methods in applications. Then, a centroid-based approach is used to detect application clones. The authors evaluated their work on more than 150,000 applications and achieved high accuracy in detection of cloned applications, but unlike us, they did not evaluate where the specific differences in similar applications lie. Chen et al. also work with code in Smali representation, in their work they compute centroids out of all given methods in the analysis step. In their clone detection step, each method of a single application is compared with all the methods that have been analyzed before.

Another research team, Wang et al., introduced a two-phased approach to detect clones in a set of Android applications [11]. Two-phase detection is used to speed up the process massively. In the first phase, Wang et al. identify relevant candidates by comparing light-weight static semantic features. In the second phase, only applications found beforehand are compared in a more detailed way to detect application clones. Like us, this team also extracts the needed features from the code in Smali language, but unlike our approach, they do not compare two given Android applications with each other, but they analyze features of a set of apps to detect application clones.

In the paper "*Plagiarizing Smartphone Applications: Attack Strategies and Defense Techniques*" [12], Potharaju et al. describe how they identify repacked Android applications. In addition to symbol coverage, which is not suitable for obfuscated applications, they use the "Abstract Syntax Tree" (AST) of applications to detect clones. AST distance and AST coverage are described measurements for their approach. They also investigated the meta-information of 158,000 apps and ascertained "that 29.4% of the applications are more likely to be plagiarized because of the permission rights they provide to an attacker" [12], which encouraged us in our idea to allow users to compare the *AndroidManifest.xml* files of two given applications with each other.

"DroidKin: Lightweight Detection of Android Apps Similarity", which was written by Gonzales et al. [13], describes their strategy of how to detect similarities between Android applications. Like us, they do not only consider the code of applications but also use information about resource files to detect similar apps. Their tool *DroidKin* extracts frequency vectors from the DEX files of applications and uses the hash values as well as metadata like size and creation date to describe resource files. The gained information is used to describe the relation of two applications by classifying them into categories like *Twins*, *Siblings*, or *Cousins*, but contrary to our approach, no information about the specific changes is provided. Like Gonzales et al., we also use hash values to detect unchanged resource files.

Another work we want to mention here is the master's thesis of Christof Rabensteiner. In his work "*Android Library Identification*" [14], Rabensteiner deals with the topic of identifying used libraries in Android applications, no matter if the source code has been obfuscated or not. To do that, a simplified AST that does not include identifier names is built from Smali code and used to generate a fingerprint, which is compared to extracted fingerprints of known libraries. While we do not use ASTs in our work, we also work with Smali code and use a similar concept to counter obfuscation techniques.

The work of Backes et al. [15] is similar to Rabensteiner's work. To detect used third-party libraries in Android applications, they use the package hierarchy structure as well as method signatures. To make their approach immune to identifier renaming, names are removed before signatures are created. The idea of removing all names also influenced design decisions in our work. However, we do not remove identifier names in all comparison steps, but only when classes or methods cannot be matched in their original form.

A paper that also deals with fingerprints, but not in the context of libraries, is *"OpSeq: Android Malware Fingerprinting"* [16]. Ali-Gombe et al. describe a fingerprinting approach which can deal with obfuscated code. To calculate similarity between apps, normalized opcode sequences as well as permissions that are described in the *AndroidManifest.xml* file are used to create fingerprints. The authors evaluated their approach with a data set of more than 1,500 Android applications and state that their tool *OpSeq* can detect malware footprints with an accuracy of 97.5%. Unlike them, we do not mix information from the code in DEX files with permissions that can be found in the manifest. However, their impressive accuracy results encouraged us in our aim of comparing not only code but also *AndroidManifest.xml* and the other resources that can be found in the provided APK files.

3.3 Deobfuscation of Android Applications

Research has shown that well-chosen identifier names have a significant impact on the intelligibility of source code [17, 18]. As many Android applications are obfuscated and therefore hard to analyze, Bichsel et al. presented an approach to deobfuscate such apps in 2016.

In their paper "*Statistical Deobfuscation of Android Applications*" [5], they describe how they learn from a large code base of non-obfuscated Android applications and how the gained probabilistic model is used to deobfuscate code. To train their model, Bichsel et al. used the source code of 1,684 freely available applications. 100 other apps that have been obfuscated with the tool *ProGuard* were used to evaluate their approach which relies on the fact that "names of methods part of the Android API and the names of classes referenced in static files" are kept intact by *ProGuard* [5]. According to the authors, their tool "*DeGuard*" could be used to recover 79.1% of the element names. Their tool is available online[3] and makes it easier to analyze obfuscated APK files.

While the goal of our approach is to find reliable ways to compare two APK files with each other and to provide the results in a structured way, *DeGuard* aims to edit obfuscated identifier names to meaningful ones and to generate a deobfuscated version of the provided application. Their visualization of results highly influenced our decision to show similarities and differences in a web browser. However, unlike them, we decided to present results in Smali code instead of Java code per default because tests have shown that Java decompilers struggle with reliable decompilation of classes.

[3]http://www.apk-deguard.com

Chapter 4

Approach

In this chapter, we introduce our solution to the problem of comparing two Android applications with each other and present our tool **APKCompare**, which implements said solution. In Section 4.1, we specify the requirements that should be fulfilled by our tool, and we also define its prerequisites and constraints.

In the following section, 4.2, we give a brief overview of the phases that our tool processes. The basic concepts of resource comparison, code comparison and the visualization of results are explained in this section, and we explain the coherence of said phases.

Section 4.3 deals with our approach of comparing resources that can be found in given APK files. We provide examples that show how important resource comparison can be, describe why our tool uses Merkle trees and explain how comparison results are saved.

Then, we have a detailed look at the code comparison strategies in Section 4.4. We discuss why code comparison is an important task and present our strategy to detect classes, methods and basic blocks that can be found in both given applications, even if obfuscation techniques have been applied. The subsection 4.4.1 provides information about important modifications of files in .smali format, which is a fundamental part of our approach. In the following section, 4.4.2, we introduce the comparison on class level. In three subsections, we explain our strategy of hashing several slightly modified files to detect similarities. Detailed figures and code examples help the reader to become acquainted with our approach. Then, in Section 4.4.3, we present our strategy of code comparison on method level and provide an example in an accurate figure.

Finally, in Section 4.5, we discuss visualization strategies of APKCompare. We explain why our visualization is based on HTML and Javascript, introduce the particular categories of matches, and state the functionalities of the web interface. Furthermore, we present four screenshots of APKCompare's web framework.

4.1 Requirements

We design a tool named **APKCompare**, which is able to list similarities and differences of two given APK files. Before being able to clarify the design concept of said tool, it is important to define how APKCompare shall be used and which requirements the tool shall fulfill.

The requirements that have been defined can be divided into requirements for **visualization** as well as **resource comparison** and **code comparison**. We will start with the category of **visualization** specifications:

- Results shall be presented in a clear and structured way. For code and resources, a side by side comparison shall be provided, so that users can quickly ascertain differences and similarities of the two given Android applications.
- A quick overview, which makes it easy to discern the similarity value of two given APK files, shall be provided by APKCompare.
- When code is shown to users, they shall have the possibility to choose if they want to investigate Smali code or the corresponding Java representation.

For the **resource comparison** process, we have defined the following three requirements in order to facilitate convincing results:

- If two given APK files share common resources, these files shall be detected as unchanged, even if the path of the files is not equal.
- If there are resource files in both given APK files that have the same path but are not equal in content, APKCompare shall categorize such files as changed resources.
- If there are resource files that occur only in one of the given APK files, such files shall be categorized appropriately.

In the category of **code comparison** specifications, we define the following four points:

- Equal .smali files that can be extracted of the given APK files shall be detected as unchanged, even if they differ in their paths.
- Classes that are fundamentally equal, but differ only in the naming of identifiers (for example because of **code obfuscation**), shall be listed as such.
- Classes that share methods or parts of methods with each other shall be detected by APKCompare. For example, it shall be possible to detect methods that have been moved from one class to another.
- Classes that are available in only one of the two given APK files shall be detected as such.

We also need to specify prerequisites that must be met and define usage constraints for the use of APKCompare, namely:

- The tool can be used to compare two Android applications, but it is not possible to compare an Android app with an app written for another platform or to compare two applications for other platforms with each other. The reason for this prerequisite is that the formats of application files for different platforms differ highly from each other.
- When presenting results, we allow a side by side comparison of text files and images. Other file types can not necessarily be presented side by side by our tool. This is due to the fact that all the code can be displayed in text format and that the bulk of resources falls into the categories of text files or images.

- The tool can be used to present equal and similar files, but it does not tell how changes affect the behavior of the application. The interpretation of changes has to be done by the user, as APKCompare is not intended to analyze the effect of code, but to present differences and similarities of two given APK files.

4.2 Comparison Workflow

This section provides a brief overview of the phases of our approach to compare two Android applications with each other in a way that the defined requirements can be fulfilled, while the subsequent sections, 4.3 - 4.5, deal with the approach of said phases in detail.

First, in the resource comparison phase, all resource files that are part of the given APK files are compared against each other. APKCompare writes information about unchanged, changed, deleted and new resources to separate result files. Users that are not interested in the differences and similarities of resources but only in code modifications can skip this phase.

In another phase, the code comparison phase, files in .smali format are extracted from the .dex files that can be found in the given APK files. We compare these .smali files against each other in order to detect unchanged classes. For changed classes, resulting .smali files are gradually reduced, and the reduced variants are compared against each other so that we can detect similar code parts. Several result files for changed and unchanged classes are written in this phase as well. While the comparison solely relies on Smali code, also code in Java representation has to be extracted in this phase, so that both representation variants can be shown to the user in later steps. This phase can also be skipped by users, which is beneficial if the interest lies solely in the comparison of resources.

Finally, we make use of the files that have been generated in the above-mentioned phases by generating a view that permits users to list similar and equal files. This view does not only list files that have been declared as matches but also allows to compare these files side by side.

4.3 Resource Comparison

The comparison of resource files is an important task. Changed files can have a far-reaching impact on the behavior of an Android application. At the same time, deleted and newly added files can change the appearance as well as the behavior massively. For example, two applications which are identical in code but differ only in the *AndroidManifest.xml* file can have different entry points and therefore completely different behavior. App permissions, which are defined in the aforementioned XML file, can also have a heavy impact on the application. Another example of changed resource files is the simple replacement of images or layout files. Such changes should not be underestimated, as for example the replacement of a stop sign image by an image of a go sign could have drastic impacts. For stated reasons we decided to endow APKCompare with the possibility of resource comparison, which is done as follows:

In a first step, all files that are part of the given APK files are extracted. Files in the .dex format are removed as we inspect them in the source comparison step, all other files are fed into Merkle trees (see 2.5). These trees can then be used to compare the files against each other quickly. Nodes with identical hash values are considered as equal, and for matching directory nodes the successors do not have to be checked anymore because of Merkle tree characteristics.

All matching nodes that are found in this step are written to a result file and removed from both Merkle trees right afterward. An example of such reduced trees can be seen in Figure 4.1. As the folders `/a/c` of `1.APK` and `/a/x` of `2.APK` are equal in content and do not have to be checked in following steps, the respective nodes are deleted from the Merkle trees.

In the next step, files with equal paths in both given APK files are detected. Resulting matches are written to a file, and the respective nodes are removed from both trees. To stick to the example in Figure 4.1, files with path `/a/b/1.txt` are available in both APKs, but they have different content as their hash values do not match.

In the final step, all files that are still represented in the first Merkle tree can only be found in `1.APK`, while files in the other tree can only be found in `2.APK`. APKCompare writes these resource files, `/a/b/2.txt` and `/a/b/99.txt` in our example, to separate result files which can be used later to visualize our findings.

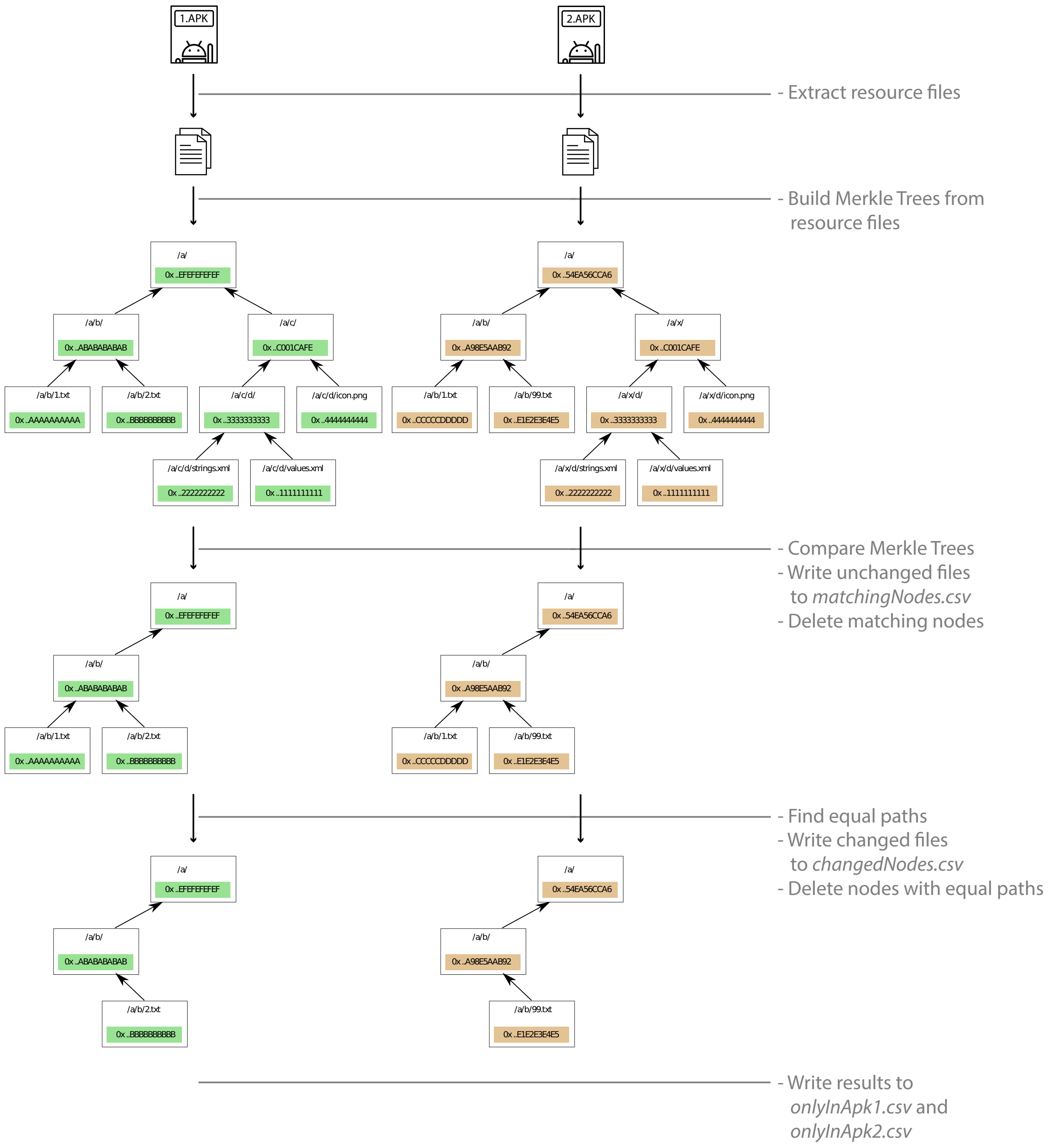


Figure 4.1: High level visualization of the resource comparison algorithm

4.4 Code Comparison

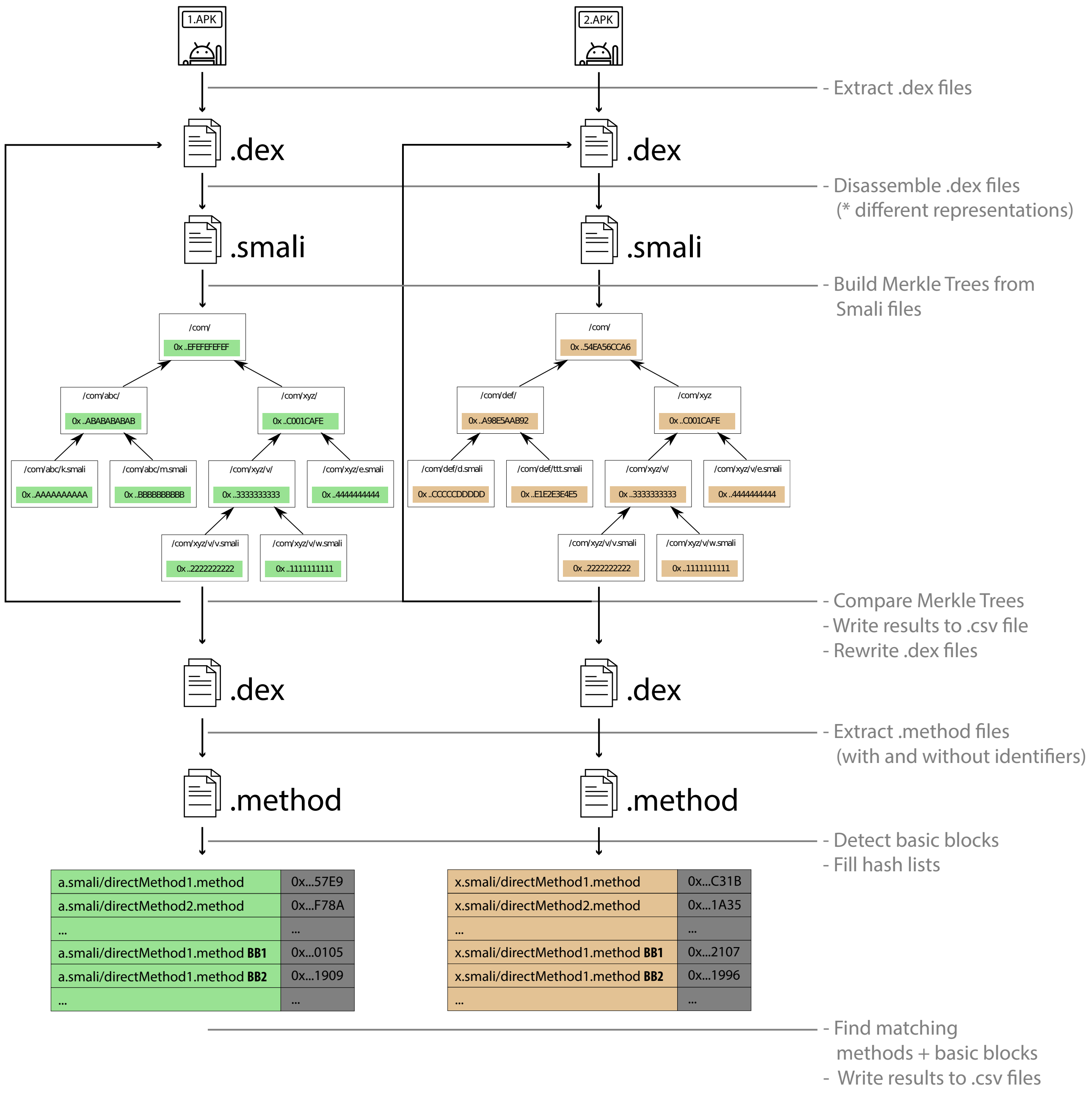
The comparison of code is an important task without any doubt. In conjunction with resource files, the code of an application specifies both, appearance and behavior. When comparing the code of two applications against each other, there are several questions of crucial importance we want to keep track of. One of these questions is how much code is shared by two applications, and where this code can be found. Large parts of shared code indicate that two applications are similar to each other, which can result from reusing publicly available code for example, but also from copying an application and only changing or adding small amounts of code. Another important question is which code can be found in only one of two given applications. Depending on the actual context of comparison, this question is of high significance for different reasons. When comparing two seemingly different applications with each other, code that can only be found in one of them describes unique characteristics. At the same time, unique parts of code can indicate for example added or removed features when comparing two versions of the same application. It is also of high importance to determine similar code, especially when comparing two versions of the same app. Even small changes like the adaption of one method parameter or the modification of a string can have a large impact on the application's behavior and are therefore important to detect.

APKCompare compares code in Smali language (see 2.3), as the full instruction set used by Android can be mapped to Smali code and as it is easier to read than Dalvik bytecode. Remember that .smali files, which can be generated for example by the disassembler *Baksmali*, describe one class of an application each. In the context of code comparison, this means that the determination of identical classes can be achieved relatively easy by comparing the hash values of .smali files, while it is a hard task to precisely tell the differences if there are any. Moreover, there is a big obstacle which complicates the comparison of Smali code, namely obfuscation. Obfuscation tools like *ProGuard* (see 2.4) typically rename classes, fields, and methods, which leads to different hash values of files obviously, while the behavior of the class stays exactly the same. For that and other reasons, APKCompare relies on an algorithm that gradually reduces Smali classes and replaces identifier names to be invariant against basic obfuscation techniques. The following sections provide details about the mentioned algorithm. A high-level visualization of the algorithm can be examined in Figure 4.2.

4.4.1 Code Comparison Fundamentals

APKCompare analyzes code in Smali language to detect code similarities and differences. The concept of our algorithm is based on hashing Smali code and comparing the resulting values in order to determine if code is equal or not. If needed, we use multiple versions of Smali code of one and the same class to detect matching ones. For unmatched classes, we generate multiple files per method and we even split these methods into finer parts if needed, again with the goal to hash the content and find equal parts. A detailed explanation of the procedure on class level (4.4.2) and method level (4.4.3) is provided in the following sections, but as both rely on Smali code, we want to cover some basic design decisions beforehand.

Whenever Smali code is used by APKCompare, **debug information** is removed from this code. While debug information like line numbers and parameter names can be helpful for understanding code, this information does not have any effect when the code is executed, but it influences the hash value. As we use hash functions to evaluate if classes, methods, or basic



a.smali/directMethod1.method	0x...57E9
a.smali/directMethod2.method	0x...F78A
...	...
a.smali/directMethod1.method **BB1**	0x...0105
a.smali/directMethod1.method **BB2**	0x...1909
...	...

x.smali/directMethod1.method	0x...C31B
x.smali/directMethod2.method	0x...1A35
...	...
x.smali/directMethod1.method **BB1**	0x...2107
x.smali/directMethod1.method **BB2**	0x...1996
...	...

Figure 4.2: High level visualization of the code comparison strategy

blocks are equal, there is no room for debug information in our approach.

Because of similar reasons, we use **implicit** method and field **references** for elements from the current class when comparing Smali code. This means in detail that the name of the current class is not used as a prefix of method and field names. A short example of the usage of implicit references can be seen in Listing 4.1. By using this reduced variant, it is possible to detect equal code parts even if the name of the current class is different.

```
iput-object p1, p0, La/b;->a:Ljava/lang/Object;
```

```
iput-object p1, p0, a:Ljava/lang/Object;
```

Listing 4.1: Comparison of Smali code without (top) and with (bottom) implicit references

The naming scheme of labels in Smali code is also noteworthy in this context. Instead of using bytecode address information to name labels (for example targets of jump instructions), we apply a **sequential** numbering scheme for each **label**. This results in easier comparison with hash values, as the bytecode address, which relies on multiple factors, does not affect the content of Smali code in this specific area anymore, while the functionality stays the same.

The **sorting** of defined **fields and methods** in Smali classes is, like for example in Java code, not important in general. However, the order of methods and fields has an impact on the hash value of a file. This means that two .smali files with the exact same set of methods and fields, but in a different order, are associated with different hash values even though they contain the exact same executing code. As a countermeasure, we link name-invariant values to every field and method in a class and use these values to sort them accordingly. Said values depend for example on used types and access flags, the detailed composition is described in Section 5.3.3. With the introduction of a naming-invariant order of methods and fields, we facilitate the process of hash-based comparison, this is why APKCompare writes the content of .smali files in the described order.

Next to the described measures that facilitate the hash-based comparison, namely removing debug information, using implicit references, the sequential numbering of labels and the name-independent sorting of fields and methods, which APKCompare always makes use of, there are also four more measures that can be applied. If those actions, with all their advantages and detriments, are performed or not depends on the user's choice. We have a look at all of them:

As stated in Section 2.3, Smali works register-based. Technically it is not important which particular registers are used by methods, therefore compilers can choose arbitrary ones. For example, the behavior of a method does not change if two registers which are used for 32-bit values each and are not used as parameter registers are switched with each other. However, hashing the two said variants of a method would result in totally different hash values. Therefore, we provide the possibility to use registers in a deterministic way, independent of the registers that have been chosen by the compiler. This measure relies on the sequential numbering of the registers, in the order they are used in methods. An example of such a sequential naming-scheme can be seen in Listing 4.2. Besides the advantage of this deterministic naming-scheme, namely that it is independent of the registers chosen by the compiler and that the usage of different registers, therefore, results in the same hash value, this of course also involves a danger. Imagine two substantial methods that are nearly equal and differ only in one addition that is made rather at the beginning of the method. The **rearrangement of registers** might lead to a shifting of all used registers, which means that all basic blocks of the method cannot be identified as matches anymore. As the inspection of several applications has shown that compilers tend to pick identical registers in most cases, this measure is turned off by default. For the comparison of some applications, for example a non-obfuscated one with an obfuscated one, using this option can be very useful however.

```
.method public static a
   (...)Z
    .registers 5

    const/4 v2, 0x1

    const/4 v1, 0x0

    if-eqz p0, :cond_a

    ...

    move-result v0

    if-nez v0, :cond_17

    :cond_a
    move v0, v2

    ...
```

```
.method public static a
   (...)Z
    .registers 5

    const/4 v0, 0x1

    const/4 v1, 0x0

    if-eqz p0, :cond_a

    ...

    move-result v2

    if-nez v2, :cond_17

    :cond_a
    move v2, v0

    ...
```

Listing 4.2: Comparison of Smali code without (left) and with (right) rearranged registers

A very similar approach to the above one is to **replace** all **registers** with a static placeholder. Again, advantages and disadvantages concerning this strategy can be found. The positive aspect of this approach is obviously that hash values of methods and basic blocks are completely independent of the specific registers that have been used. Also, for changes like the above-listed example of a newly introduced addition at the beginning of a method, the remaining basic blocks of such a method can be detected as unchanged with this approach. On the other hand, we can present an example where this strategy leads to undesirable behavior: Think of a method that does multiple computations and returns a value at the end. If this method was error-prone before and could be fixed in an updated application version by simply returning the value of another register, this change can not be detected if all registers are replaced by a placeholder. Several tests have shown that this strategy can be useful in some cases, however, it is turned off by default as accuracy in comparison is lost when using it.

```
const v0, 0x7f02027f
...
const v2, 0x7f0a0240
```

```
const v0, **APKCOMPARE.drawable.my_credit**
...
const v2, **APKCOMPARE.string.events_birthday_today**
```

```
<public type="drawable" name="my_credit" id="0x7f02027f" />
...
<public type="string" name="events_birthday_today" id="0x7f0a0240" />
```

Listing 4.3: Comparison of Smali code without (top) and with (middle) resolved resources replaced. The bottom listing shows the appropiate entries in */res/public.xml*

Another optional measure which we introduced to facilitate the process of hash-based comparison is the **replacement** of **resolved resources**. An example of such a replacement can be seen in Figure 4.3 and shows how resource identifiers are used in .smali classes. As resource identifiers can change between two versions of an application and as we are usually interested in the type and the content instead of the identifier, APKCompare supports the replacement of identifiers with the actual type and content that can be found in the `/res/default.xml` file. The mentioned XML file is part of the application's APK file. Tests have shown that this strategy usually simplifies the comparison process, this is why it is turned on by default.

A final action that is definable by the user and should be mentioned here is the **exclusion** of classes defined in the **`android`** package. With this action set, classes of the named package are not disassembled and therefore not used for code comparison. This leads to a faster runtime and a more clear visualization of results but can be a disadvantage when trying to get an overview of the code base. In the default behavior, APKCompare does exclude classes of said package when comparing code.

4.4.2 Comparison on Class Level

Finding equal and similar classes with APKCompare is based on the idea of gradually trimming classes in order to compare them with each other and to exclude already matched classes from following comparisons. APKCompare carries out the analysis of equality by comparing hash values, to be more precise we make use of the benefits of Merkle trees again when comparing code on class level. For our work, we use the disassembler tool *Baksmali* to generate .smali files from all the .dex files that can be found in the given applications. To generate files with various levels of detailedness, we adapted *Baksmali* and introduced several parameters that lead to a different output. A detailed look at said parameters is provided in Chapter 5.3.

In general, we can say that the comparison on class level is divided into three subtasks, namely the comparison of unmodified .smali classes, different variants with slight adaptions, and of files where nearly all identifier names have been replaced with placeholders. In every comparison step, resulting matches are written to a file, and all classes that are considered as matches are excluded from later comparison steps. We have a more detailed look at the process in the following three sections.

4.4.2.1 Detecting Identical Classes

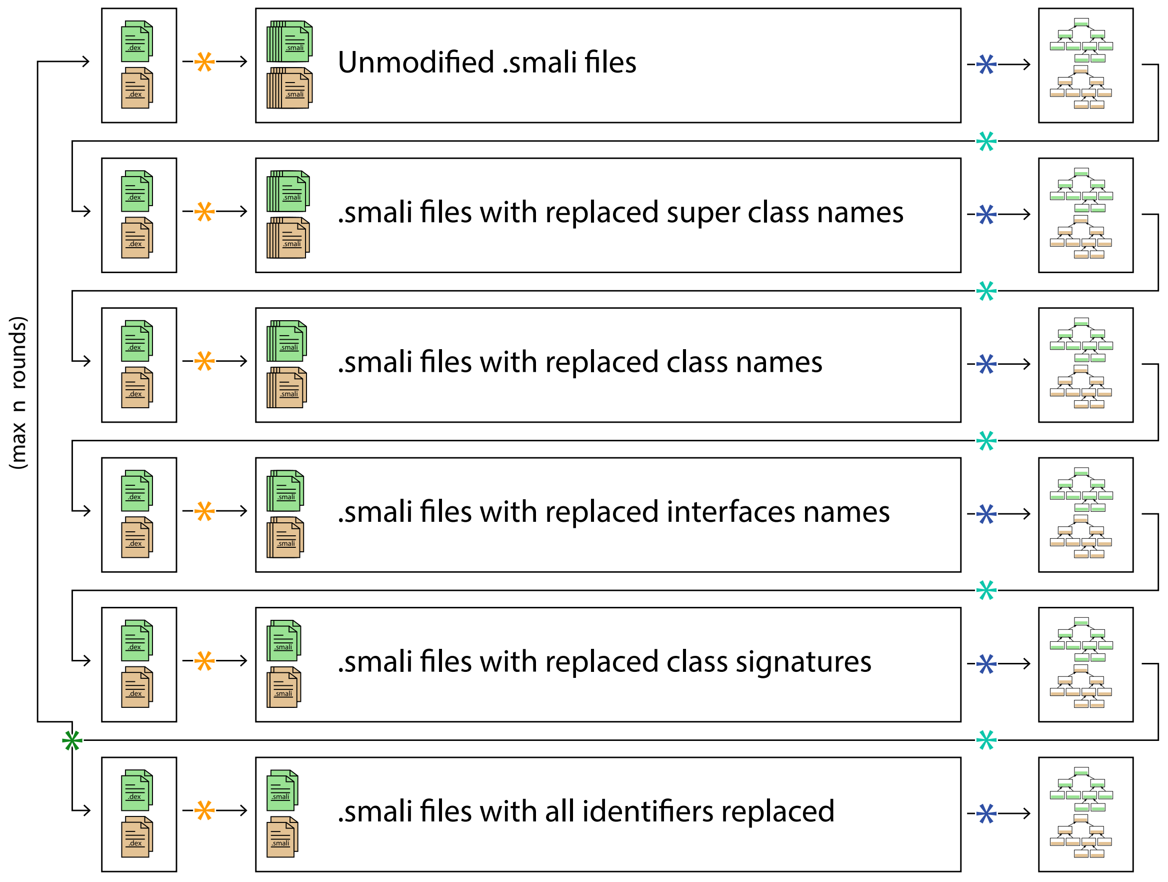
Let's remind ourselves that the comparison on class level is done in multiple steps with different levels of detailedness. We start with the comparison of unmodified .smali classes, as they provide the most meaning. To work with said files, we use the tool *Baksmali* without any options that lead to the replacement of certain values. However, we want to highlight again that the measures which are explained in Section 4.4.1 are applied to the whole source comparison process. Once all .dex files of both provided APK files have been disassembled, representative Merkle trees are built up. These two trees are compared against each other and nodes with equal hash values are considered as matches. The paths of equal elements are written to a result file immediately after the comparison. The specific structure of result files is not of high importance, but they have to contain both paths, that in APK1 and that in APK2, of equal nodes. If a match describes a directory, the entry in the result file also has to involve the number of subfolders and the number of files that are contained in the matching directory. As we do not want already matched classes to be considered in later comparison steps again, we write the matched classes to two separate files, one for each APK file. Only one list of already matched classes would not be sufficient, as classes might have different namespaces or names in the provided APK files, and as our custom version of *Baksmali* needs the paths per APK file in later steps.

4.4.2.2 Detail Variants

Now that we could detect equal classes in both provided applications' APK files, we want to compare slightly adapted versions of the files with each other. First, we detect classes that only differ in the names of the used super classes. To do that, our adapted version of *Baksmali* is executed with a parameter which causes that the names of the super classes are replaced by the placeholder string `_SUPERCLASS_PLACEHOLDER_`. As a result, .smali files that only differed in these names have the same hash value now and can, therefore, be detected as matches. Again, we write the results to a separate file, which can be used later to list all classes where only the names of super classes have been changed. Results from this step are also added to the lists of classes that shall be ignored in later comparison steps.

Then, a very similar action is executed, but this time we replace the name of the class instead of the super class name, which results in the possibility to detect renamed classes when comparing the Merkle trees. Also in this step, all results are written to a separate file and the lists of classes to ignore is updated again.

In the next step, class names and names of used super classes are unmodified, but the names of interfaces are replaced as *Baksmali* is executed with a custom parameter again. You can see an example of such a modified file in Listing 4.4. Just as in previous steps, we create and update said files here.



(max n rounds)
Unmodified .smali files
.smali files with replaced super class names
.smali files with replaced class names
.smali files with replaced interfaces names
.smali files with replaced class signatures
.smali files with all identifiers replaced

* Disassembling of .dex files, with respect to the lists of classes to ignore
* Generation of Merkle Trees
* Comparison of Merkle Trees + Writing results to files + Updating lists of classes to ignore
* Rewriting of .dex files + Resetting lists of classes to ignore

Figure 4.3: Visualization of the code comparison algorithm on class level

```
.class public final Lcom/a/b/c/d;
.super Ljava/lang/Object;

# interfaces
.implements Lcom/a/b/a;
.implements Lcom/a/b/b;
```

```
.class public final Lcom/a/b/c/d;
.super Ljava/lang/Object;

# interfaces
.implements _INTERFACE_PLACEHOLDER_
.implements _INTERFACE_PLACEHOLDER_
```

Listing 4.4: Comparison of Smali code without (top) and with (bottom) replaced interface names

Then, a combination of the previous three replacement variants is made to find similar classes. The names of classes, super classes, and the implemented interfaces are replaced by placeholders in this step. We call this a replacement of class signatures in Figure 4.3, which illustrates the whole process of code comparison on class level. Said figure also exposes that although the same .dex files are used in the respective steps, the number of resulting files in .smali format potentially shrinks in every step. This is caused by the mentioned lists of classes that shall not be disassembled as they have already been matched.

We assume that all matching classes that have been identified in any of the previous variants represent the same class in the two given APK files. If any of the said variants, including the comparison of unmodified .smali files, led to at least one match, we profit from this knowledge by rewriting all .dex files of both given applications' APK files. This is best explained by taking the example of two classes, `La/b/c;` in APK1 and `Lx/y/z;` in APK2, which have been matched in any of the above steps. In every of the .dex files of APK1, the class `La/b/c;` is renamed to `Lrenamed/by/apkcompare/number1;` and the same is done for the class `Lx/y/z;` in .dex files of APK2. When the said .dex files are disassembled again, every instruction that used the classes `La/b/c;` or `Lx/y/z;` respectively beforehand, uses the `Lrenamed/by/apkcompare/number1;` class now. This again leads to the same hash of two .smali files if the files only differed in these instructions before.

As we can see in Figure 4.3, we can make use of .dex file rewriting after the step of replacing class signatures and repeat the five described steps again with the newly edited .dex files. This process can be done several times, in fact, it makes sense as long as matching classes could be found in the previous round. However, as the rewriting process of .dex files takes some time, users of APKCompare can define how many rounds of this process should be done at maximum. We also want to mention that the content of the lists that hold the classes which can be ignored for later steps can be reset after every round. As all matching classes have been renamed to `Lrenamed/by/apkcompare/numberX;`, it is sufficient to have one entry in each of the said lists at the beginning of a round, namely `Lrenamed/by/apkcompare/`.

After one or more rounds of the above-described steps have been executed, a final code comparison step on class level is performed. This step is based on the replacement of nearly all identifier names and is described in detail in the following section.

4.4.2.3 Overcoming Obfuscation

With the above-listed steps, we can match classes that are almost equal but differ in details like the name of the class. The round system that has been introduced supports to rename already matched classes in the .dex files and to run the said steps more often in order to gain more matching results. However, classes where for example fields or methods have been renamed, no matter if by developers or by obfuscation tools, can not be identified as matches by the presented comparison steps. Therefore we introduce another step, where nearly all identifier names are replaced before we hash the files and compare the resulting values with each other. As it can be investigated in Figure 4.3, this step is executed only once. This is due to the fact that multiple executions of this step, including rewriting the .dex files, would not lead to any more resulting matches as all unknown classes had already been replaced when the step was executed the first time.

The disassembling process is done with our custom version of *Baksmali* again, a newly introduced parameter indicates that identifier names like field names, method names or type names shall be replaced by static placeholder strings. However, we do not replace all said names, but use the following strategy:

```
.method public final run()V
    .registers 3

    iget-object v0, p0, b:Lb/f;

    iget-object v0, v0, Lb/f;->a:Lb/e;

    iget-object v1, p0, a:Lrenamed/by/apkcompare/number124;

    invoke-static {v0, v1}, Lb/e;->a(Lb/e;
       Lrenamed/by/apkcompare/number124;)V

    return-void
.end method
```

```
.method public final _METHOD_NAME_PLACEHOLDER_()V
    .registers 3

    iget-object v0, p0, _FIELD_NAME_PLACEHOLDER_:_TYPE_PLACEHOLDER_

    iget-object v0, v0, _CLASS_PLACEHOLDER_->_FIELD_NAME_PLACEHOLDER_:
       _TYPE_PLACEHOLDER_

    iget-object v1, p0, _FIELD_NAME_PLACEHOLDER_:
       Lrenamed/by/apkcompare/number124;

    invoke-static {v0, v1}, _CLASS_PLACEHOLDER_->
       _METHOD_NAME_PLACEHOLDER_(
       _TYPE_PLACEHOLDER_Lrenamed/by/apkcompare/number124;)V

    return-void
.end method
```

Listing 4.5: Comparison of Smali code without (top) and with (bottom) replaced identifier names

- We replace **method names** by the string `_METHOD_NAME_PLACEHOLDER_` unless the original method name is `<init>` or `<clinit>`. The former name is used for constructors, the second one for static initializers. We do not replace these names as they can not be renamed by obfuscation tools or developers.

- We replace all **field names** by a static string in this step, as all of them could have been renamed.

- When replacing **class names**, we do not modify classes of certain packages like `java/lang/`, `java/util/` or `android/`. The reason for that is that they can not be renamed. If we renamed those classes before hashing the files, we would lose some information. The same is valid for classes of the `renamed/by/apkcompare/` package. All classes of that package are already known, replacing them would be a loss of information.

- The replacement of **types** follows similar rules. If a certain type is an object, the same rules as for class name replacement are used. Moreover, the basic datatypes, namely `V,Z,B,S,C,I,J,F` and `D` (see Table 2.2 on page 11) are not replaced as they can not be modified by obfuscation tools.

A comparison of two identical Smali methods, one without and one with identifiers replaced, can be found in Listing 4.5.

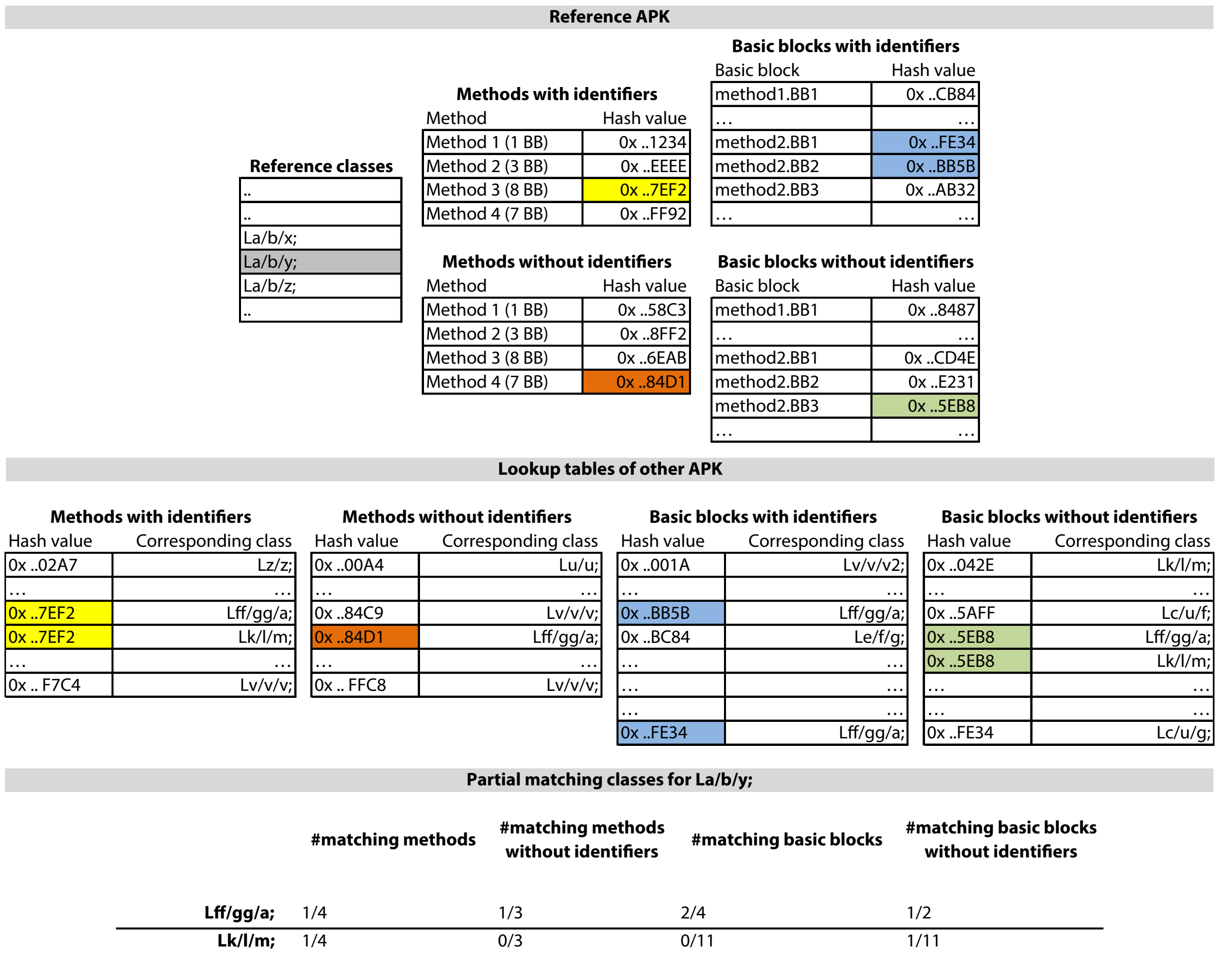
Once all modified .smali files have been extracted from the given .dex files, two Merkle trees are built and compared against each other. Matching results are written to a file again like it was done for previous steps. After that step, we are done with the comparison on class level, and we have to investigate the methods of the classes which could not be matched yet.

4.4.3 Comparison on Method Level

Once the comparison on class level is done, there will typically remain many classes that could not be matched. Concretely spoken, at least all classes where one or more methods have been added, removed or actually edited, will not be matched with the techniques that have been introduced. Therefore APKCompare starts a comparison on method level, and with respect to the general comparison strategy, only classes that could not be identified as matches beforehand are inspected in this step.

As a first step in code comparison on method level, we rewrite all .dex files of the given applications' APK files again. All classes that could be identified as matches beforehand are renamed in all .dex files of both applications, similar as described in Section 4.4.2.2. Then, two files for each method in every class, one in original form and one with identifiers replaced like described in Section 4.4.2.3, of both provided APK files are extracted. To do that, we use the customized version of *Baksmali*. This version expects a parameter which tells how many instructions a method must consist of at minimum in order to be seen as descriptive enough. Methods with the minimum amount of instructions are written to files in the .method format, methods with fewer instructions are saved in .tinyMethod files and methods without any instructions are written to .emptyMethod files. This method instruction threshold can be defined by users of APKCompare and has a large impact on the results, as we will discuss later on in this chapter.

For all classes that do not comprise at least one method with the defined minimum number of instructions, an entry is written to a corresponding result file. This file can be used later to list all classes that have not been investigated on method level because of the method instruction threshold. For all other classes, APKCompare saves the hash values of all methods, identifies all basic block leaders (see Section 2.6) of those methods and saves the hash value of the resulting basic blocks. Also for basic blocks, users of APKCompare can set an instruction threshold. We describe the advantages and disadvantages of high thresholds at the end of this section. When identifying basic blocks of methods, it is sufficient to run the identification algorithm on method files with identifiers. As methods with and without identifiers solely differ in some names, the start and end line numbers of basic blocks are identical and we do not need to identify basic block leaders in both variants.



Reference APK

Reference classes

..
..
La/b/x;
La/b/y;
La/b/z;
..

Methods with identifiers

Method	Hash value
Method 1 (1 BB)	0x ..1234
Method 2 (3 BB)	0x ..EEEE
Method 3 (8 BB)	0x ..7EF2
Method 4 (7 BB)	0x ..FF92

Basic blocks with identifiers

Basic block	Hash value
method1.BB1	0x ..CB84
…	…
method2.BB1	0x ..FE34
method2.BB2	0x ..BB5B
method2.BB3	0x ..AB32
…	…

Methods without identifiers

Method	Hash value
Method 1 (1 BB)	0x ..58C3
Method 2 (3 BB)	0x ..8FF2
Method 3 (8 BB)	0x ..6EAB
Method 4 (7 BB)	0x ..84D1

Basic blocks without identifiers

Basic block	Hash value
method1.BB1	0x ..8487
…	…
method2.BB1	0x ..CD4E
method2.BB2	0x ..E231
method2.BB3	0x ..5EB8
…	…

Lookup tables of other APK

Methods with identifiers

Hash value	Corresponding class
0x ..02A7	Lz/z;
…	…
0x ..7EF2	Lff/gg/a;
0x ..7EF2	Lk/l/m;
…	…
0x .. F7C4	Lv/v/v;

Methods without identifiers

Hash value	Corresponding class
0x ..00A4	Lu/u;
…	…
0x ..84C9	Lv/v/v;
0x ..84D1	Lff/gg/a;
…	…
0x .. FFC8	Lv/v/v;

Basic blocks with identifiers

Hash value	Corresponding class
0x ..001A	Lv/v/v2;
…	…
0x ..BB5B	Lff/gg/a;
0x ..BC84	Le/f/g;
…	…
…	…
…	…
0x ..FE34	Lff/gg/a;

Basic blocks without identifiers

Hash value	Corresponding class
0x ..042E	Lk/l/m;
…	…
0x ..5AFF	Lc/u/f;
0x ..5EB8	Lff/gg/a;
0x ..5EB8	Lk/l/m;
…	…
…	…
0x ..FE34	Lc/u/g;

Partial matching classes for La/b/y;

	#matching methods	#matching methods without identifiers	#matching basic blocks	#matching basic blocks without identifiers
Lff/gg/a;	1/4	1/3	2/4	1/2
Lk/l/m;	1/4	0/3	0/11	1/11

Figure 4.4: Example method lookup of code comparison

Once the hash values of methods and basic blocks that fulfill the threshold conditions could be determined, matches are identified. For each class of the reference application, APKCompare tries to find potential matching classes in the other application, and comparisons in up to four different levels of detailedness are applied to do that. First, hash values of reference methods with identifiers are looked up in the sorted list of methods with identifiers of the other application. If at least one finding could be made, we remember the corresponding classes of the other application. An example of such a finding is highlighted in yellow in Figure 4.4, which demonstrates the comparison on method level for the class `La/b/y;`.

If no method in the other application matches the hash value, we repeat that process but use the hash value of the method without identifiers this time and look for the value in the according table built for methods of the other application. Again, we remember the corresponding classes of the other application for every match. The orange highlighted areas in Figure 4.4 show such a match.

If there was still no match, we make use of the basic blocks of the unmatched method. For each basic blocks' hash value, we look into the corresponding sorted lookup table of the other application. If basic blocks with identifiers have equal hash values, we remember the corresponding class. If there was no matching value, we do a lookup without identifiers. The blue and green areas in the example mark matching basic blocks.

With these four steps, classes that have some parts in common can be identified. In the example provided in Figure 4.4, we have two partially matching classes for the `La/b/y;` class of the reference application. The class `Lff/gg/a;` matches fairly well. One of four methods is identical, one is nearly identical but was not detected as such in the first step because of renaming, two of the remaining four basic blocks are equal, and one more basic block only differs in identifier names. Only one basic block of the reference class could not be matched in the `Lff/gg/a;` class of the other application. The other partially matching class, namely `Lk/l/m;` also shares some code with the reference class. One of four methods is identical, and from the remaining eleven basic blocks in the reference class, one matches if identifier names are not considered. Of course, all partially matching classes that can be identified are written to a file again, together with the detailed information of the number of matching methods and basic blocks, with and without identifiers.

However, searching for matching methods and basic blocks with only one application as reference is not enough. Methods and basic blocks that are only part of the other APK are not considered in this step, therefore the comparison on method level is done twice by our tool. Once with one APK file as reference and the other APK file to fill the lookup tables, and once the other way round.

Also noteworthy here is that all classes that could not be matched on method level are written to files. These files can be used later to list all those classes as such that seem to appear only in one of the given applications.

Because of the high impact on the results, let us again discuss the influence of the thresholds for method instructions and basic block instructions:

The **method instruction threshold** describes the minimum number of instructions for a method. Methods with fewer instructions are ignored when comparing classes on method level. A low value means that also short methods like getters and setters are compared with each other, which results in potentially better matches but also in more wrong findings. A high value means that short methods are not considered when comparing classes on method level. This results in a lower number of wrong matches, but will provide less potential matches. APKCompare's default value for this threshold is six.

The **basic block instruction threshold** describes the minimum number of instructions for a basic block, blocks with fewer instructions are ignored. Also here, a low value means that also short basic blocks are compared with each other, resulting in potentially better matches, but also in more wrong findings, while a high value leads to a lower number of wrong matches, but will provide less potential matches. The default value is three.

4.5 Visualization

The visualization of results is of high importance for us, as APKCompare is designed to list similarities and differences of two given APK files and to compare resources as well as source files side by side. To visualize results that have been gained in prior steps, a solution that is based on HTML and JavaScript is introduced. We made this decision so that the results can be shown on any device that runs a browser, and also because of the idea of providing APKCompare as a web service in the future.

APKCompare generates a web page that is able to present comparison results, navigate through result categories, compare images and text files side by side, and much more. The dashboard provides an overview of the parameters that have been used to run the tool and describes the impact of these parameters. Also, quick overviews that allow to easily ascertain which proportions of resources and code are changed or unchanged are shown in the form of interactive pie charts.

Files that can be compared against each other are organized in groups. There are two main categories, namely **Resources** and **Code**, with multiple groups each. These groups are filled with the content of result files which have been written by APKCompare in the resource comparison and code comparison steps. All listed matches, independent of the respective category, consist of two file paths. Resources can be of arbitrary file types, for example .xml, .png or .txt, while all entries listed in the code category are of file type .smali. Matching files can be compared side by side by clicking on them. For image files, a view that shows both images is visualized, which allows users to examine them quickly. Text files are not only shown side by side, however, because the web view interface also offers the possibility to view the exact changes of two text files directly in the web browser. Matches that describe whole directories are listed in said categories as well, clicking on them provides detailed information of the numbers of subfolders and files that are part of the directory. Files and folders that have been extracted from the first declared APK file are shown on the left side, while files and folders of the second APK file are always shown on the right side. While it will be sufficient to compare files that have been identified as matches by APKCompare in most cases, the web interface also allows comparing arbitrary files of the given applications with each other. This can be done by right-clicking the according files and choosing to show them on the left or the right side of the side by side comparison view.

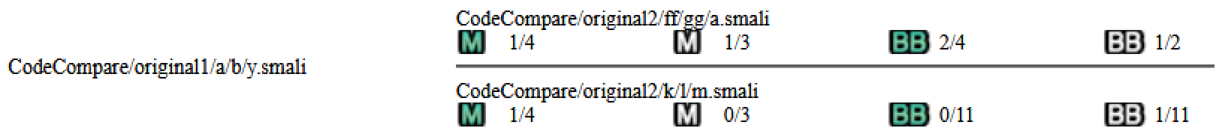
The matches that are listed in the web interface are read from files which have been written before and are categorized accordingly. The specific categorization is as follows:

- **Resources**
  - Unchanged Files
  - Changed Files
  - Only in APK1
  - Only in APK2
- **Code**
  - Unchanged Files
  - Different super class names
  - Different class names
  - Different interface names
  - Different class signature
  - Unchanged but renamed/obfuscated (Round x)
  - Potentially unchanged but renamed/obfuscated
  - Classes of APK1, not checked on method level
  - Classes of APK2, not checked on method level
  - Partial matches APK1->APK2
  - Partial matches APK2->APK1

– Only in APK1
– Only in APK2

For users that are not familiar with the meaning of certain categories, a text that describes the entry is provided for each category. This text is shown when the mouse cursor is moved over the according entry. As APK files can contain lots of resource files and classes, which leads to many entries in the different groups, it is possible to toggle the visibility of whole groups that are not of interest. The web view also allows us to filter the list of matches, which constitutes a very handy and time-saving feature when looking for certain files.

For partially matching classes that have been identified like described in Section 4.4.3, it is not sufficient to list classes next to each other, but also the number of matching methods and basic blocks (with and without replacing identifier names each) has to be shown. In APKCompare's web interface, we present these values as shown in Figure 4.5. The sample entry relates to the classes and values shown in Figure 4.4.



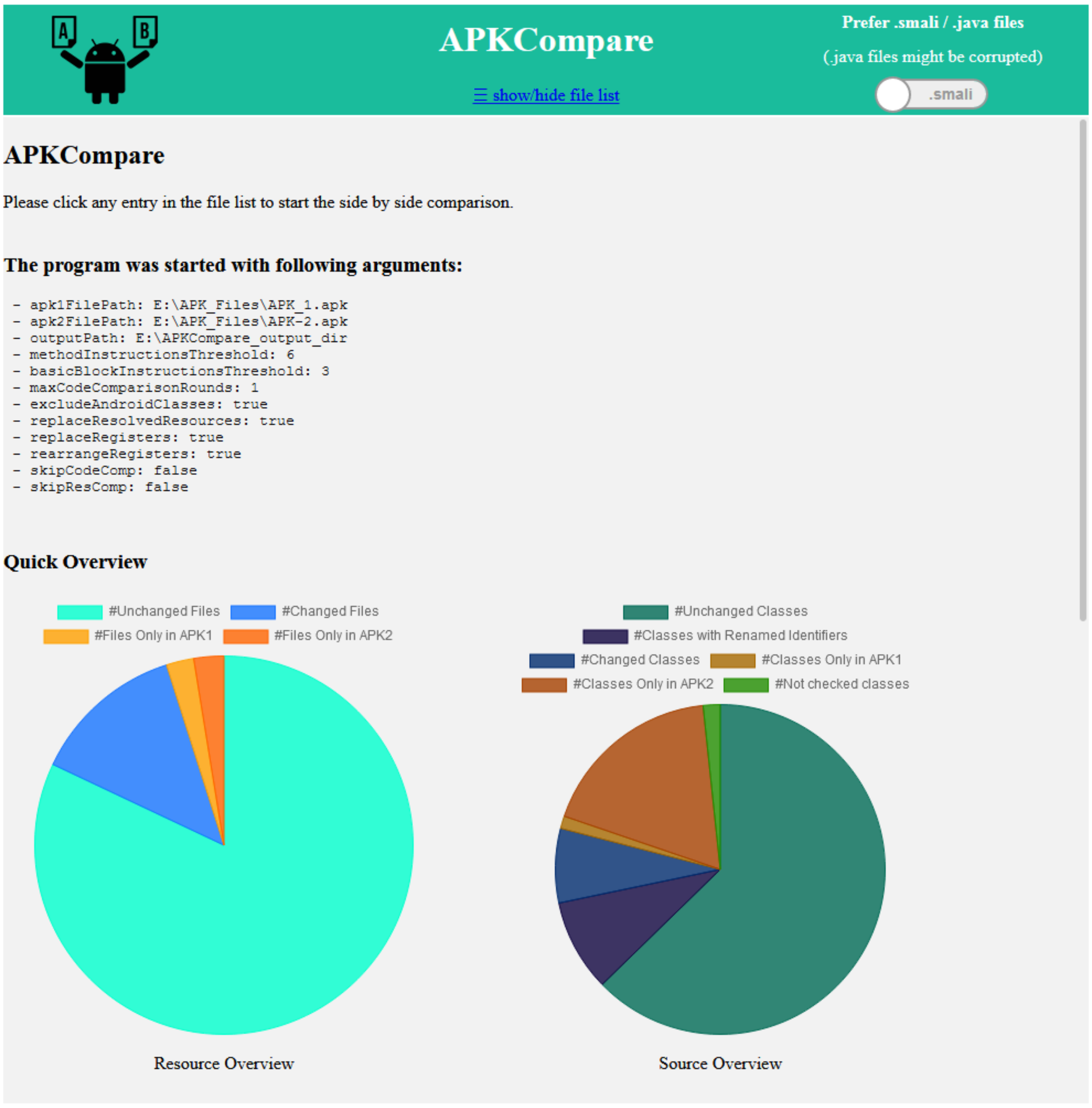
CodeCompare/original1/a/b/y.smali	CodeCompare/original2/ff/gg/a.smali	M 1/4	M 1/3	BB 2/4	BB 1/2
	CodeCompare/original2/k/l/m.smali	M 1/4	M 0/3	BB 0/11	BB 1/11

Figure 4.5: Example entry of partially matching classes

When comparing the code of two give applications, only code in the Smali language is considered. However, as many potential users are not familiar with reading Smali code, we added a switch that allows selecting the preference of Smali/Java code to the web interface. Depending on the user's selection, classes can be inspected in the according variant. Independent of the chosen variant, files are always presented in an unmodified way. This means for example, that even if all identifier names have been replaced by APKCompare in order to find matching classes, the files which are presented to the user contain the original naming information.

All text files that are presented in the web interface are searchable directly in the browser, which allows a quicker workflow if users already know what they are looking for.

The screenshots provided in Figures 4.6 - 4.9 on the pages 39 - 42 give an overall overview of the web interface that is generated by APKCompare.



APKCompare

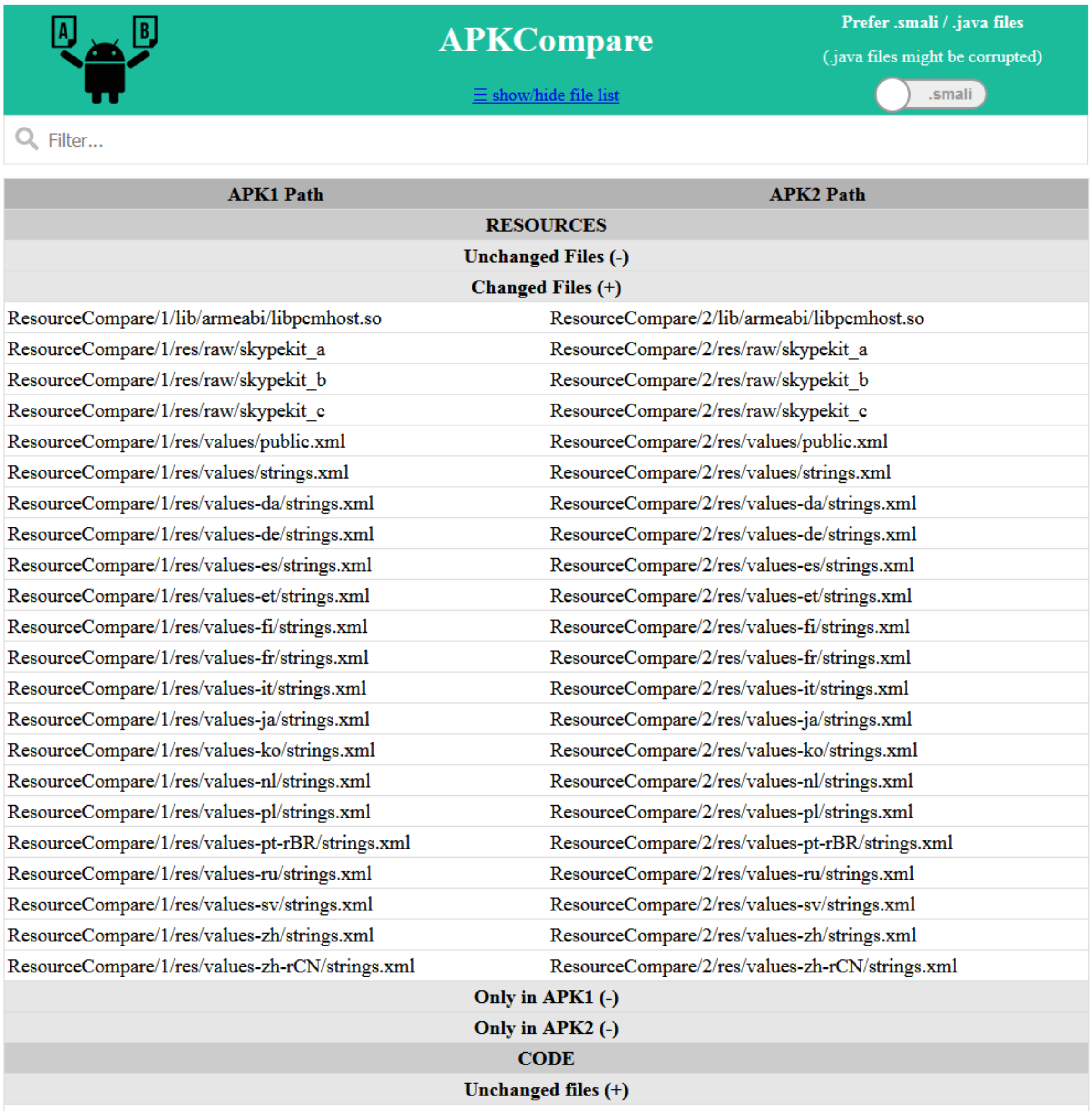
Please click any entry in the file list to start the side by side comparison.

The program was started with following arguments:

```
- apk1FilePath: E:\APK_Files\APK_1.apk
- apk2FilePath: E:\APK_Files\APK-2.apk
- outputPath: E:\APKCompare_output_dir
- methodInstructionsThreshold: 6
- basicBlockInstructionsThreshold: 3
- maxCodeComparisonRounds: 1
- excludeAndroidClasses: true
- replaceResolvedResources: true
- replaceRegisters: true
- rearrangeRegisters: true
- skipCodeComp: false
- skipResComp: false
```

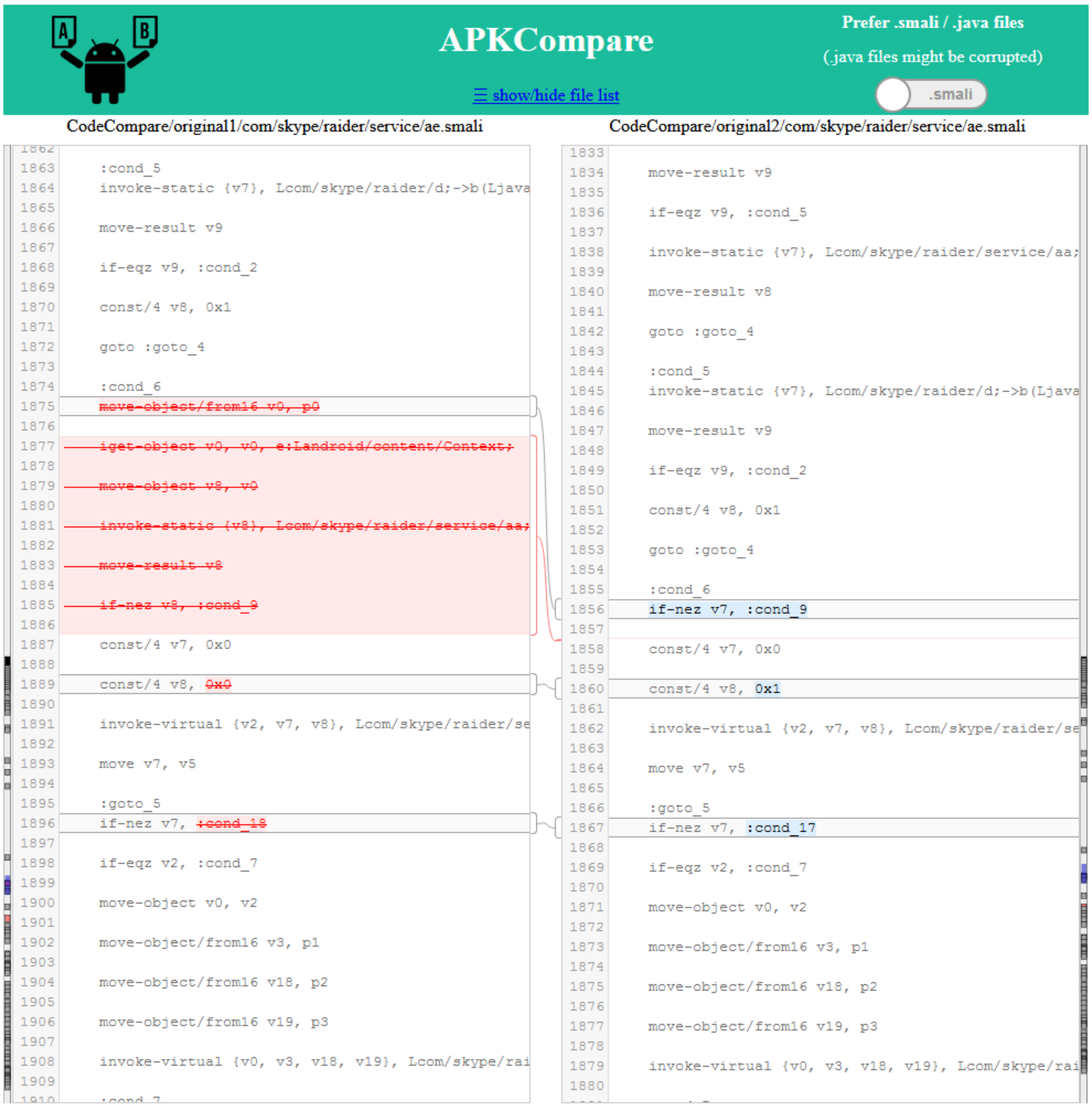
Quick Overview

Figure 4.6: Screenshot of APKCompare's dashboard, which informs about used parameters and provides a quick overview of comparison results



APK1 Path	APK2 Path
RESOURCES	
Unchanged Files (-)	
Changed Files (+)	
ResourceCompare/1/lib/armeabi/libpcmhost.so	ResourceCompare/2/lib/armeabi/libpcmhost.so
ResourceCompare/1/res/raw/skypekit_a	ResourceCompare/2/res/raw/skypekit_a
ResourceCompare/1/res/raw/skypekit_b	ResourceCompare/2/res/raw/skypekit_b
ResourceCompare/1/res/raw/skypekit_c	ResourceCompare/2/res/raw/skypekit_c
ResourceCompare/1/res/values/public.xml	ResourceCompare/2/res/values/public.xml
ResourceCompare/1/res/values/strings.xml	ResourceCompare/2/res/values/strings.xml
ResourceCompare/1/res/values-da/strings.xml	ResourceCompare/2/res/values-da/strings.xml
ResourceCompare/1/res/values-de/strings.xml	ResourceCompare/2/res/values-de/strings.xml
ResourceCompare/1/res/values-es/strings.xml	ResourceCompare/2/res/values-es/strings.xml
ResourceCompare/1/res/values-et/strings.xml	ResourceCompare/2/res/values-et/strings.xml
ResourceCompare/1/res/values-fi/strings.xml	ResourceCompare/2/res/values-fi/strings.xml
ResourceCompare/1/res/values-fr/strings.xml	ResourceCompare/2/res/values-fr/strings.xml
ResourceCompare/1/res/values-it/strings.xml	ResourceCompare/2/res/values-it/strings.xml
ResourceCompare/1/res/values-ja/strings.xml	ResourceCompare/2/res/values-ja/strings.xml
ResourceCompare/1/res/values-ko/strings.xml	ResourceCompare/2/res/values-ko/strings.xml
ResourceCompare/1/res/values-nl/strings.xml	ResourceCompare/2/res/values-nl/strings.xml
ResourceCompare/1/res/values-pl/strings.xml	ResourceCompare/2/res/values-pl/strings.xml
ResourceCompare/1/res/values-pt-rBR/strings.xml	ResourceCompare/2/res/values-pt-rBR/strings.xml
ResourceCompare/1/res/values-ru/strings.xml	ResourceCompare/2/res/values-ru/strings.xml
ResourceCompare/1/res/values-sv/strings.xml	ResourceCompare/2/res/values-sv/strings.xml
ResourceCompare/1/res/values-zh/strings.xml	ResourceCompare/2/res/values-zh/strings.xml
ResourceCompare/1/res/values-zh-rCN/strings.xml	ResourceCompare/2/res/values-zh-rCN/strings.xml
Only in APK1 (-)	
Only in APK2 (-)	
CODE	
Unchanged files (+)	

Figure 4.7: Screenshot of APKCompare's visualization of matching files



APKCompare

☰ show/hide file list

Prefer .smali / .java files

(.java files might be corrupted)

.smali

CodeCompare/original1/com/skype/raider/service/ae.smali

```
:cond_5
invoke-static {v7}, Lcom/skype/raider/d;->b(Ljava

move-result v9

if-eqz v9, :cond_2

const/4 v8, 0x1

goto :goto_4

:cond_6
~~move-object/from16 v0, p0~~

~~iget-object v0, v0, e:Landroid/content/Context;~~

~~move-object v8, v0~~

~~invoke-static {v8}, Lcom/skype/raider/service/aa;~~

~~move-result v8~~

~~if-nez v8, :cond_9~~

const/4 v7, 0x0

const/4 v8, ~~0x0~~

invoke-virtual {v2, v7, v8}, Lcom/skype/raider/se

move v7, v5

:goto_5
if-nez v7, ~~:cond_18~~

if-eqz v2, :cond_7

move-object v0, v2

move-object/from16 v3, p1

move-object/from16 v18, p2

move-object/from16 v19, p3

invoke-virtual {v0, v3, v18, v19}, Lcom/skype/rai

:cond_7
```

CodeCompare/original2/com/skype/raider/service/ae.smali

```

move-result v9

if-eqz v9, :cond_5

invoke-static {v7}, Lcom/skype/raider/service/aa;

move-result v8

goto :goto_4

:cond_5
invoke-static {v7}, Lcom/skype/raider/d;->b(Ljava

move-result v9

if-eqz v9, :cond_2

const/4 v8, 0x1

goto :goto_4

:cond_6
if-nez v7, :cond_9

const/4 v7, 0x0

const/4 v8, 0x1

invoke-virtual {v2, v7, v8}, Lcom/skype/raider/se

move v7, v5

:goto_5
if-nez v7, :cond_17

if-eqz v2, :cond_7

move-object v0, v2

move-object/from16 v3, p1

move-object/from16 v18, p2

move-object/from16 v19, p3

invoke-virtual {v0, v3, v18, v19}, Lcom/skype/rai

```

Figure 4.8: Screenshot of APKCompare's side by side comparison view for text files. The left side shows a file that was extracted from the first given application, the right side shows the corresponding file from the second app. Detected changes are color-coded.

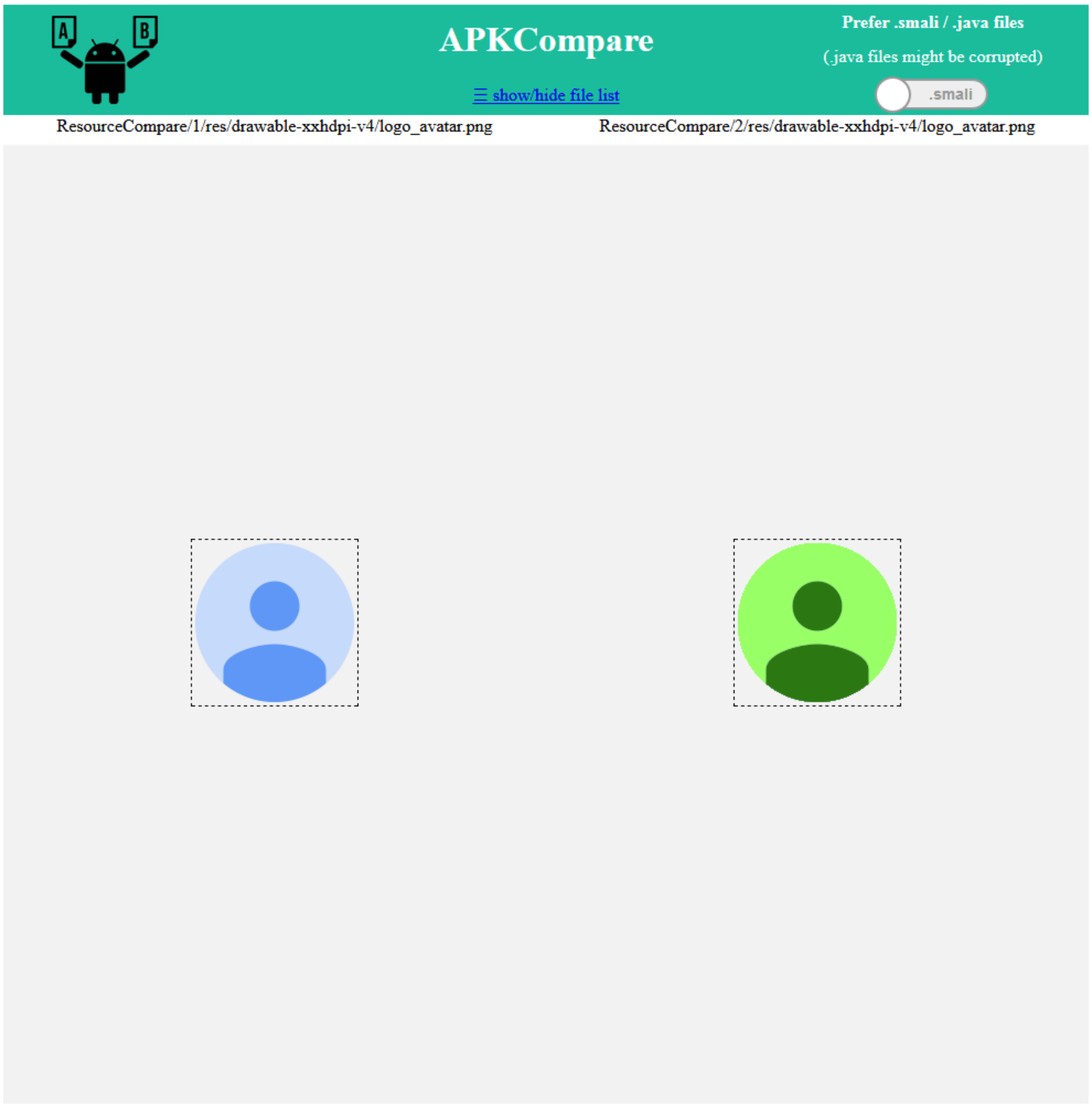

APKCompare
☰ show/hide file list
Prefer .smali / .java files
(.java files might be corrupted)
.smali
ResourceCompare/1/res/drawable-xxhdpi-v4/logo_avatar.png
ResourceCompare/2/res/drawable-xxhdpi-v4/logo_avatar.png

Figure 4.9: Screenshot of APKCompare's side by side comparison view for images. The left side shows an image that was extracted from the first given application, the right side shows the corresponding file from the second app.

Chapter 5

Implementation Details

While the general approach of our solution has been discussed in Chapter 4, this chapter provides information about more detailed design and implementation decisions, tools that have been utilized and alternative ways that could have been used for our solution.

In Section 5.1, we describe why and how APK files are extracted. The tool that was used for this approach, *Apktool*, is also presented in this section.

Then, in Section 5.2, we have a look at two hash algorithms that have been candidates for our project. We discuss how we decided on which algorithm to use and also provide a detailed look at our implementation of Merkle trees.

Next, we have a very detailed look at the Smali disassembler *Baksmali* in Section 5.3. Next to the description of basic functionality of the tool, we also discuss why we adapted the project so that it fits our needs and we provide some examples of generated code. Additionally, we give information about options that have been used and describe their effects on generated code in the Subsection 5.3.1, and introduce a custom option that is used to exclude certain classes from disassembling in the following one. Then, Subsection 5.3.3 deals with the topic of name-invariant sorting of methods and fields. After explaining how we construct descriptive values, an example of such values for one method and one field is provided. The next subsection, 5.3.4 describes the implementation of replacing identifiers. All four identifier replacement options that have been added to *Baksmali* are introduced in this subsection. In the following one, another new option, which allows the replacement of resolved resources, is described. Then, in Subsection 5.3.6, two more custom *Baksmali* options are explained. Said options allow the replacement and the rearrangement of registers in Smali code. We describe which registers are actually affected by these options and discuss potential problems that might arise when using said options. To round off the section about *Baksmali*, we give an overview of our implementation of splitting classes and writing methods to separate files in Subsection 5.3.7.

Section 5.4 deals with the topic of rewriting DEX files. We describe which parts of DEX files are actually rewritten and how it is done. We also have a look at some strategies that have been realized because of optimization reasons.

The next section, 5.5, is dedicated to the topic of basic block detection. Here, we describe our approach of two steps that were implemented to detect basic block leaders and to use these leaders to build up the basic blocks. We also discuss some special lines of basic blocks that are ignored when determining hash values.

Then, Section 5.6 gives an overview of our implementation of the extraction of Java code.

We describe how we use the tools *dex2lib* and *Java Decompiler* to extract Java code from DEX files and also have a short look at alternative tools that could have been used.

Finally, in Section 5.7, we discuss implementation details of the web view that is used to present results. Next to the explanation of used third-party libraries, we have a look at the structure of CSV files and introduce some features that have been implemented as they turned out to be very convenient.

5.1 Extracting APK Files

The goal of the tool APKCompare is to compare resource files as well as code of two given applications, but as APK files are not compared against each other directly, these files need to be extracted beforehand. To do this, we rely on a widely used tool, namely *Apktool*[1], which is open source and licensed under the Apache 2.0 License. *Apktool* provides functionalities for reverse engineering of Android APK files, for example it can be used to disassemble resources and code and to rebuild the modified application again. In our work, we use this tool solely to extract all files from given APK files. To do that, we start *Apktool*'s `decode` mode, and as we inspect the .dex files by ourselves in the code comparison step, we use the `--no-src` option, which prevents the disassembly of the DEX files. Figure 5.1 visualizes this process.

Apktool is started directly from APKCompare, once for both declared APK files that shall be compared. The resulting files are used in the comparison steps, which have been introduced in the sections 4.3 and 4.4.

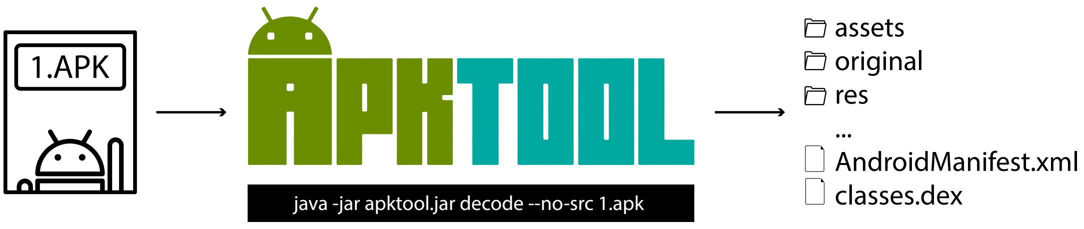


Figure 5.1: APK file extraction with *Apktool*
[*Apktool* icon created by https://ibotpeaches.github.io/Apktool/]

5.2 Hash Algorithms

Hashing files is a fundamental task in our approach, and as this operation is done so often, we have used two different algorithms in our implementation, namely **MD5** and **MurmurHash3**[2], and tested which one is more convenient for our needs. Both algorithms return a hash value of 128 bits, which can be displayed in 32 characters when printing it in hexadecimal notation.

[1]https://github.com/iBotPeaches/Apktool
[2]https://github.com/yonik/java_util/blob/master/src/util/hash/MurmurHash3.java

To decide which algorithm is better suited for our approach, we used both of them to create Merkle trees and compared the average run times. The files that were used to build up the Merkle trees have been chosen representatively. To be more detailed, resource files that were extracted from an APK file have been used for one test, and .smali files of the same APK have been used for another. We decided on this strategy as it might be the case that one algorithm performs better for text files but is worse when hashing binary files, or vice versa. In total, we used 1,864 resource files of different file types, including .png, .xml and .jpg, with approximately 15MB of size in total and 7,523 .smali files with approximately 100MB in size in total to build up Merkle trees with both hash algorithms. Both algorithms have been used to build Merkle trees of the stated files 20 times each, while the run time was measured and the mean value was calculated. For the roughly 1,800 resource files, the average run time to create a Merkle tree with the MurmurHash3 algorithm to hash files was 225 milliseconds on our test machine, while it lasted 260 milliseconds on average to create said tree when using the MD5 message-digest algorithm. Also for building up Merkle trees for .smali files, the MurmurHash3 algorithm resulted in better run times than MD5. The average run time was 939 milliseconds, while it lasted 20% longer, 1,129 milliseconds when using MD5 to determine hash values of files. As a result of the tests, we chose the MurmurHash3 algorithm to hash files in our work.

5.2.1 Hash Lookup with Merkle Trees

Merkle trees have been used in our approach when comparing resources of two given applications, but also when comparing code on class level. Merkle trees are well-suited for these tasks, as they allow a quick comparison of directories. For our tool, we implemented the functionality to build, compare and manipulate Merkle trees by ourselves, as no existing library that matches our needs could be found. Because of the high impact of such trees on our work, we also implemented unit tests that allow inspecting if the behavior of our implementation works as expected.

Like all other components of APKCompare, except the web framework that visualizes the results, we wrote the code that handles Merkle trees in the programming language Java. We implemented a `MerkleNode` class, which holds the path of a file or folder, the hash value, the parent node, and all child nodes. Objects of this class can be checked for equality against each other, whereby objects are considered as equal if the hash value is identical. Also, we implemented a method that compares two given MerkleNode objects with each other. This method compares nodes that represent folders in a first step. To do this, the hash values of folder nodes of the first given MerkleNode, we call it *nodeA*, starting with the top level node, are compared against the hash values of folder nodes of the second MerkleNode, *nodeB*. If nodes are matching, we mark them as such, because they and their child nodes do not have to be considered in later comparison steps. Otherwise, child nodes of *nodeB* which represent folders are compared with *nodeA* recursively. If no match for *nodeA* could be found, the same procedure is repeated for all its child nodes that represent folders. After the comparison of folders, all nodes that could already be matched are removed from both Merkle trees. Then, the reduced trees which contain only nodes that represent unmatched files and folders, are compared against each other again, but this time file nodes are checked for equality. As the directory structure is not of importance in this step anymore, all node elements that represent files of *nodeB* are stored in a sorted list, where the nodes are arranged with respect to their corresponding hash values. This trick allows us to do a fast binary search in the list of *nodeB* file nodes for every file node of *nodeA*. To speed this procedure up even more, matching nodes are removed from the sorted list so that they do not have to be checked when looking for matches of the other file nodes. Also, all matching nodes are marked as such in both Merkle trees again. All matches, no matter if they represent files or folders, are returned in a list by the method. Once all nodes that have been marked as already matched are removed from the Merkle trees, only nodes that represent files and folders which could not be matched remain. We use this characteristic for example when comparing the resources of two applications, as it can be read in Section 4.3.

The introduced implementation has large benefits. Next to the fact that a comparison based on Merkle trees can be done considerably faster than a comparison of the hash values of all files, the resulting list of matches is also shorter and easier to read. As matching directories are recognized as such, it is sufficient to list the directory paths as a match, while an approach without the usage of Merkle trees would lead to a large list of matching files without providing more information in the result. On the other hand, there is also a design decision in this implementation which could result in restrictions in certain cases. Whenever two or more elements are qualified as matches because of the same hash value, our approach only detects one. Such multiple potential matches can result from hash collisions as well as from identical files. The former is highly unlikely as we use hash values with 128 bits in size, the latter is accepted because of the following reasons: In the first place, no files are lost in the comparison results. Even if a file could not be detected as a match, it is listed in the resulting web interface and can be compared by hand if needed. Furthermore, the alternative solution would even be worse according to our opinion, because if all potential matches were listed, we would have to remove all of them when reducing the Merkle trees. Resulting from this, the results of further lookups would be affected as well.

5.3 Baksmali

The tool *Baksmali*, which was already introduced in Section 2.3, is used very often in our approach. Baksmali is part of the open source project *Smali* [4] and implements multiple commands like for example `dump`, which prints an annotated hex dump for given DEX files, or `list`, which can be used to list objects like strings in a DEX file. For our work, only the `disassemble` command is used. This command is called with one DEX file and several options that define the specific behavior and writes files in .smali format to a defined output directory. As the original version of *Baksmali* does not provide options for all variants that are needed for our approach, we forked the git repository[3] and adapted the *Baksmali* project to our needs. The following subsections deal with specific tasks that are accomplished by using our custom version of *Baksmali* and give insight into some relevant implementation details.

5.3.1 Minimizing .smali Classes

In our approach, files that are extracted with *Baksmali* are compared by checking if their hash value is equal or not. As we are interested in the behavior that is described by code, we try to minimize Smali code so that the described functionality stays the same while unneeded information which affects the resulting hash values is removed. To do that, the following standard options of *Baksmali*'s `disassemble` command are used:

[3]https://github.com/JesusFreke/smali

`--debug-info false` defines that debug information like the names of parameters, information about the line numbers, or similar information is not included in resulting files.

`--accessor-comments false` is used so that helper comments for synthetic accessors are not written to resulting files.

By using the `--implicit-references` option, the name of the current class is not used as a prefix for methods and fields. This does not affect the behavior but results in the same generated code if only the name of the class has changed. A comparison of resulting code with and without the usage of the named option can be seen in Listing 4.1 on page 27.

The `--sequential-labels` option is used for a reason that has already been explained in Section 4.4.1. With this option, *Baksmali* renames labels in Smali code. Instead of using the bytecode address to name labels, the naming scheme relies on sequential numbering. This is convenient for our approach as it raises the probability that methods with same behavior generate the same code and therefore result in the same hash value.

5.3.2 Exclusion of Classes

As we already described in Section 4.4 in detail, one of the basic concepts of APKCompare is that already matched classes are not considered in later steps anymore. In our approach, we do not only exclude those matched classes from the comparison, but because of better run times, classes which have already been matched are not even disassembled by *Baksmali*. To do that, we introduced the `--classesToExcludeFile [file]` option for the `disassemble` command, which allows defining classes as well as whole packages that shall be excluded from the disassembling process. The structure of files that can be read by this parameter is very easy, all lines starting with # are ignored, while all other lines have to contain the name of one package or one class. Classes have to be defined in Smali syntax, meaning that they have to start with `L` and end with `;`, while packages have to end with a trailing `\`. A valid example file can be seen in Listing 5.1.

```
#Classes considered as matches have been renamed to '
    Lrenamed/by/apkcompare/numberX' and can be ignored.
Lrenamed/by/apkcompare/
#Classes starting with 'Landroid/', because command line parameter '
    excludeAndroidClasses' was set.
Landroid/
#Further classes that do not have to be disassembled:
Lcom/a/b/c;
Lcom/a/b/d;
Lcom/a/b/e;
```

Listing 5.1: Example content of a file used by the –classesToIgnore parameter

5.3.3 Sorting Methods and Fields

In this subsection, we want to provide a more detailed look at our implementation of sorting methods and fields in Smali code. The reasons why we decided to sort methods and fields with respect to name-invariant values were already covered in Section 4.4.1, but are briefly explained again to facilitate understanding. The order of methods in .smali files is not of relevance and the same holds for the order of fields. However, two .smali files that have the exact same methods and fields, but distinguish in the order of them, have completely different hash values and can therefore not be detected as equal on class level by our tool. Because of that, name-invariant values that describe fields and methods have been introduced, and our customized version of *Baksmali* sorts methods and fields with respect to these values if the option `--sort-name-independent true` is used.

The values have been designed with the goal to describe the method or field in a very detailed manner without considering its name. Methods and fields with low values are written first, such with higher values are written afterward.

For **methods**, the following features are considered to determine a descriptive value:

1. Access flag of the method
2. Number of method parameters and their types
3. Return type of the method
4. Method index defined in .dex file

For **fields**, the descriptive value consists of similar features:

1. Access flag of the field
2. Type of the field
3. Field index defined in .dex file

Each of the features can be described as an integer value. For the return type and the parameter types, the first nine types defined in Figure 2.2 are mapped to the values 1 to 9, for other types the value 0 is used. For example, a parameter of type boolean is mapped to value 2, an integer parameter to value 6, and La/b/c; to value 0. The access flag value was used from existing *Baksmali* code[4], as it is already described as an integer value and provides exactly what we need. Method index as well as field index do not provide descriptive information about the elements but are needed to decide on the order of two or more fields or methods if they had the same values otherwise. Said index is read from the DEX file and tells at which position the elements have been defined. It is important to understand that equal methods/fields in two different APK files typically do not have the same index, which also means that the order of fields and methods can also differ for actually identical classes, despite the introduced values to sort them. This can happen if and only if there are several items/methods with the same features, but changed indices. However, this is also not a major issue for methods as said classes can be matched when comparing classes on method level (see Section 4.4.3).
The name-independent value is a concatenation of the above-listed values. A very descriptive example of said values of a field and a method is provided in Listing 5.2.

[4]https://github.com/JesusFreke/smali/blob/master/dexlib2/src/main/java/org/jf/dexlib2/AccessFlags.java

```
.field final synthetic a:Lb/e;
#nameIndependentHashValue: 41120
#access flags: 4112        (=final synthetic)
#type: 0                   (=other)
#fieldIndex: 78


.method public constructor <init>(La/b/c;La/b/d;J)V
#nameIndependentHashValue: 655377001
#access flags: 65537       (=public constructor)
#param types: 0 / 0 / 7    (=other / other / long)
#return type: 1            (=void)
#methodIndex: 718
```

Listing 5.2: Examples of name-independent hash values for methods and fields

5.3.4 Replacing Identifier Names

In Section 4.4.2.2, we already discussed our approach of comparing .smali files in different detail variants. In this section, we will describe which options we implemented for *Baksmali*'s `disassemble` command in order to generate said variants. All of the mentioned options can be used in combination as well of course.

- The option **`--replace-superclass-names true`** was introduced to generate Smali classes where the name of the super class is replaced by the string `_SUPERCLASS_PLACEHOLDER_`.
- With the **`--replace-class-names true`** option set, files where the name of the class is replaced with the string `_CLASS_IDENTIFIER_PLACEHOLDER_` are generated. An example of the impact of the parameter can be seen in Listing 5.3.
- An example of the impact of the **`--replace-interface-names true`** option can be investigated in Listing 4.4 on page 31. With this option, the names of implemented interfaces are replaced by the string `_INTERFACE_PLACEHOLDER_`.
- The option **`--replace-all-identifiers true`** has the largest impact of the four listed options. When this option is chosen, our customized version of *Baksmali* tries to replace all identifiers that could be affected by obfuscation techniques. To do that, names of fields, methods and types are replaced by following the **replacement rules** that are provided in Section 4.4.2.3. You can find a very descriptive example of the impact of this option on page 33 in Listing 4.5.

```
.class public final La/a;
.super Ljava/lang/Object;
```

```
.class public final _CLASS_IDENTIFIER_PLACEHOLDER_
.super Ljava/lang/Object;
```

Listing 5.3: Comparison of Smali code without (top) and with (bottom) replaced class name

5.3.5 Replacing Resolved Resources

Because of reasons stated in Section 4.4.1, the replacement of resource identifiers has a positive impact on our comparison approach. Unfortunately, the original version of *Baksmali* does not provide an option to replace such identifiers, however, a similar option, `--resolve-resources [prefix] [xml file]`, is supported. This option can be used to add names of resources that can be found in the given file as comments in generated Smali code files. As the original resource identifier is still part of the resulting files and therefore impacts the generated hash values, we had to introduce a similar option which is not only inspired by the original option but also very similar in implementation.

Said option, **`--replace-resolved-resources [xml file]`**, in fact replaces resource identifiers that can be found in the given XML file. An example of code generated with this option set can be found in Figure 4.3 on page 29. Users of APKCompare do not have to define the XML file to read the values from, as the file is extracted from given APK files and used as input for said option per default.

5.3.6 Manipulating Registers

To manipulate registers in Smali code, we implemented two disassemble options, namely **`--replace-registers true`** and **`--rearrange-registers true`**. Both options are very similar, and advantages as well as disadvantages of using these options have already been discussed in Section 4.4.1.

Both options only affect local method registers, which means that parameter registers are unchanged. Parameter registers can be recognized by their prefix `p`, while local registers start with the character `v`. The option which replaces all local registers was straightforward to implement, as all registers are simply replaced by the string `v<REGISTER_PLACEHOLDER>`. For the other option, we always use `v0` as the first register that is used in a method, `v1` as the second, and so on. In order that the semantics of generated code remain unchanged, a map which holds the original registers and the replaced ones is used to generate files. An illustrative example of such generated code with rearranged registers can be found on page 28 in Figure 4.2.

For both options, instructions that work with a range of registers have to be rewritten. As an example of such an operation, `invoke-direct/range {v17 .. v20}, methodtocall` can be provided. It is obvious that the registers that define the range are useless when replacing all registers. However, when rearranging registers, we also have to manipulate the written string to `{v?? .. v??}`. The reason for this decision is that the originally defined register range is not valid anymore as we rearranged them. All main changes for said options have been made in *Baksmali*'s `RegisterFormatter.java` class.

5.3.7 Splitting Classes

In Section 4.4.3, we described our approach of comparing classes on method level. In order to determine the hash values of methods, we decided to implement another option to *Baksmali* which allows generating one file per method instead of one file per class. As the number of instructions a method has gives a hint about the expressiveness of the method, the introduced option, **`--only-write-methods [threshold]`**, is called with a threshold parameter that indicates the minimum amount of instructions a method must contain in order to write it to a `.method` file. Methods that contain less instructions are written to `.tinyMethod` files, for methods that do not contain a single instruction, for example abstract methods, files of the `.emptyMethod` type are created. The handling of the generated files is done by APKCompare.

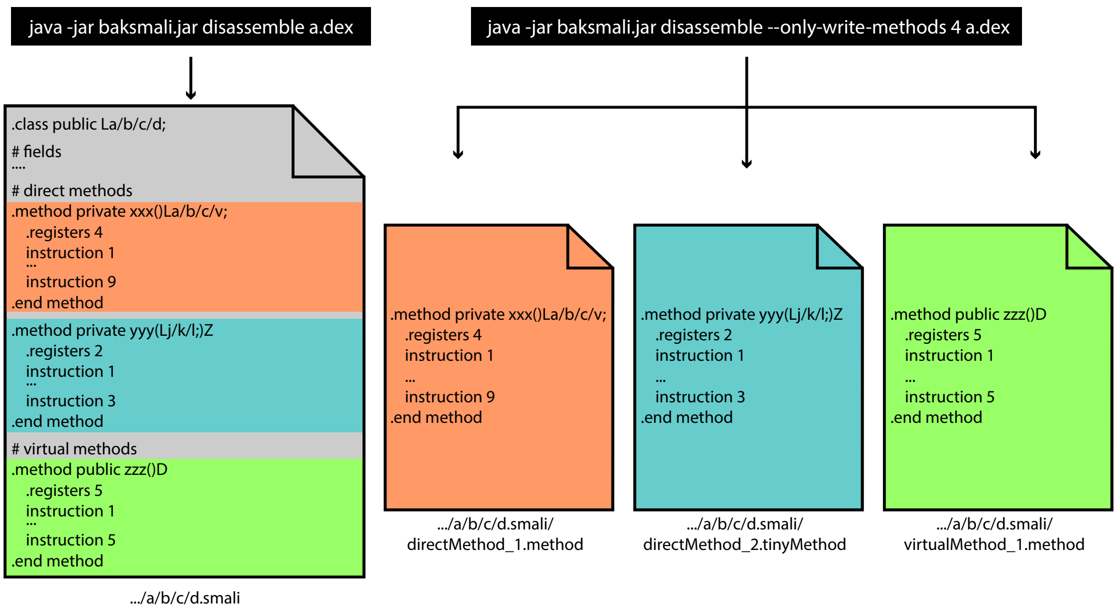


Figure 5.2: Visualization of *Baksmali*'s --only-write-methods option

An example of the usage of the introduced option can be investigated in Figure 5.2. Imagine a DEX file that only holds one class, namely `La/b/c/d;`. This class consists of several fields and three methods, two of them are direct methods, and one is a virtual one. When *Baksmali* disassembles the DEX file without the introduced option, exactly one file is created, and this file describes the whole class. In the example in Figure 5.2, the resulting .smali file can be seen on the left side. However, when *Baksmali* disassembles the same DEX file with the parameter `--only-write-methods [threshold]` with 4 as the chosen value in our example, three files are created. For each of the methods, exactly one file is written, and depending on their instruction count, the file type is `.method`, `.tinyMethod` or `.emptyMethod`. In the given example, the methods which are highlighted orange and green are written to `.method` files, as they fulfill the threshold value condition with nine, respectively five instructions, while the turquoise highlighted method is written to a `.tinyMethod` file, as it consists of only three instructions. As the file name is extraneous for the hash value, we decided to name the files depending on whether the method is a direct or a virtual one, and added a running number.

5.4 Rewriting .dex Files

Rewriting DEX files is an important step in our approach, and some of the comparison techniques we have chosen would not work without doing so. Therefore, we implemented a component which allows renaming instances in DEX files. When doing this, not only definitions but also all references to said instances are renamed, which also means that the resulting DEX file is still valid and fully functional. Our implementation is inspired by a sample snippet that is provided by the developer of the Smali project [4] on GitHub[5]. The said component is able to rename instances with respect to a given list that holds all original names and the corresponding new ones. To do that efficiently, the passed list of name pairs has to be provided alphabetically sorted, so that a fast binary search can be executed when looking for entries in the list. Entries of said list cannot only contain names of classes, but also names of packages, which allows renaming all class instances of a package with only one entry in the list.

We implemented a component that takes a DEX file as a basis and creates a new DEX file with renamed instances which is saved at a given path. It also returns a boolean value that tells if the resulting file was actually edited or not, which can be used by APKCompare to determine if more comparison rounds on class level (see Section 4.4.2.2) are justified or not. For optimization reasons, we do not rewrite the same instances multiple times, but always modify the latest version of already rewritten DEX files.

5.5 Basic Block Detection

To detect basic blocks and save the corresponding hash values, we implemented an approach consisting of two steps. In the first step, the basic block leaders (see Section 2.6) of all methods are determined, while in the second step the specific basic blocks strings are built and hashed. We will have a more detailed look at both said steps in this section.

Our implementation of the detection of basic block leaders is partly based on code that was written by Johannes Feichtner as part of the *Cryptoslice* [3] project. Said code parts were kindly provided to us and helped to implement the basic block leader detection algorithm, which works as follows for each given *.method* file:

1. Read the .method file and save the content of every line as well as the corresponding line number to a list.
2. Add the value 2 to the list of basic block leaders, as the second line of the file holds the first instruction of the method and is therefore always a basic block leader.
3. Find all labels and fill a data structure that maps labels to the corresponding line numbers in the .method file.

[5]https://gist.github.com/JesusFreke/c557ab4dd0fffcd5f545

4. Detect all instructions of the types JMP (for example `if-eq` or `if-gt`) and SWITCH (`packed-switch` and `sparse-switch`) and add the line numbers of directly following lines to the list of basic block leaders.

5. Find all instructions of the type GOTO and add the directly subsequent line numbers as well as the line numbers of lines that are the target of the instructions to the list of basic block leaders.

6. Detect all JMP instructions and add the target line numbers of the instructions to the basic block leaders list.

7. For all instructions of type SWITCH, add the line numbers of directly subsequent lines to the list of basic block leaders.

8. For all entries of switch tables, add the line numbers of lines that define the target label to the basic block leader list.

9. Detect all lines that start with `.catch` or `.catchall` and add all directly following line numbers as well as the target line numbers of the instructions to the list of basic block leaders.

10. Check the list of basic block leader line numbers and increment values that describe empty lines by one, so that all values describe a non-empty line.

11. Sort the list of basic block leader line numbers.

12. Remove duplicate entries of the basic block leader list, if there are any.

13. Return list.

With the knowledge of all basic block leaders of a given *.method* file, basic blocks can be built and the corresponding hash values can be saved. When building basic block strings that can be used for hashing, we do not use all lines of the original *.method* file, but remove the empty ones as they do not hold any information. The last line of the last basic block of a method is also not taken into consideration, as it always holds the string "`.end method`" and is not actually describing the basic block. The number of lines of basic block strings is equivalent to the number of instructions. This number can be used to decide if the hash value of the string is saved or not, as basic blocks with very few instructions do not hold enough information to be relevant for comparison (see Section 4.4.3).

We want to highlight again that the basic block leaders for the same methods, once with and once without identifiers renamed, are exactly the same. Therefore it is sufficient to detect only the basic block leaders of methods with identifiers and use the gained information twice in the second step when the hash values of the basic blocks with and without identifiers renamed are determined.

In Section 2.6, we provide a listing that shows a sample *.method* file and the detected basic blocks. In the last paragraph of said section, you can find detailed information about the basic blocks in this listing, for all basic block leaders a description that informs why the line actually is a leader is provided.

5.6 Extracting Java Code

In the whole process of comparing two given APK files, the Java representation of source code does not play a role at all. Whenever the code of the two applications is compared, the Smali representation is used, as it is already explained in Section 4.4. However, we are also interested in the Java representation of the applications' classes, as the programming language Java is widely used and more easy to read. In order to provide the possibility to show users the code in Java language, we have to extract .java files from all DEX files. To do that, we need to execute two steps for every DEX file, which are defined as follows.

At first, we create a JAR file from the given DEX file. We use the open source tool *dex2jar*[6] to fulfill this task. This tool is called with the path of a DEX file and creates a JAR file, which holds one .class file per class as well as some metadata, all together packed in zip format-type. *dex2jar* also creates a file that lists information about all classes that could not be written to the JAR file, however we do not use any information of said error file as we do not rely on Java code anyway and classify every class that can be displayed in Java code as a piece of bonus information for the user.

Then, .java files have to be extracted from the resulting JAR file. To do that, many Java decompilers are available, we want to mention the tools *Java Decompiler*[7], *Procyon*[8], *CFR*[9] and *Krakatau*[10] as some of many examples. All of these tools can be used to decompile a JAR file, and all of them have their advantages and disadvantages. We chose to use the tool *Java Decompiler* in our implementation, as it is fast and easy to use, and also because tests showed that a high proportion of classes can be decompiled with this tool. However, this tool also struggles with the decompilation of some classes, which results in commented out byte code in some resulting .java files. This is fine for our needs, however, as the .smali files for all classes are available anyway. Listing 5.4 shows an example of such code that could not be decompiled. *Java Decompiler* creates one .java file for each class, just as *Baksmali* creates one .smali file for each class. Because of that behavior, it is easy to map the resulting files to each other in a later step when we provide users with the possibility to choose their favorite way of code representation.

In Listing 5.5, two representations of the same method can be seen. In the top part, the method is described in Smali language, while the bottom part shows Java code that could be obtained by using the said tools *dex2jar* and *Java Decompiler*.

[6]https://github.com/pxb1988/dex2jar

[7]http://jd.benow.ca

[8]https://bitbucket.org/mstrobel/procyon/wiki/Java%20Decompiler

[9]https://www.benf.org/other/cfr/

[10]https://github.com/Storyyeller/Krakatau

```
  /* Error */
  public void run()
  {
    // Byte code:
    //   0: new 213 java/io/BufferedReader
    //   3: dup
    //   4: new 215 java/io/InputStreamReader
    //   7: dup
    //   8: invokestatic 221  java/lang/Runtime:getRuntime  ()Ljava/lang
       /Runtime;
    //   ...
```

Listing 5.4: Java code that could not be decompiled

```
.method public final e()V
    .registers 3

    iget-object v0, p0, d:Lb/y;

    if-eqz v0, :cond_0

    iget-object v0, p0, a:Ljava/util/List;

    iget-object v1, p0, d:Lb/y;

    invoke-static {v0, v1}, Ljava/util/Collections;->sort(
       Ljava/util/List;Ljava/util/Comparator;)V

    :cond_0
    return-void
.end method
```

```
public final void e()
{
  if (d != null) {
    Collections.sort(a, d);
  }
}
```

Listing 5.5: Comparison of a method in Smali (top) and Java (bottom) representation

5.7 Web View

When it comes to the visualization of results, we decided to implement a web interface that supports several features, including the side by side comparison of resources and source code.

The basic approach, the functionality as well as the specific list of result categories and some screenshots of the interface have already been provided in Section 4.5. This section will mainly deal with implementation details of the web interface.

To provide users with a possibility to quickly determine how similar two given APK files are, two pie charts are displayed on the dashboard of the web view. One of the charts presents the similarity of the resources and shows four categories, namely the number of unchanged files, the number of changed files, as well as the number of files that could only be found in APK1 and APK2 respectively. The other pie chart presents the similarity of source code. Here, the categories do not exactly match the categories in the file list (see Section 4.5), but for reason of a clear visualization the categories are as follows: number of unchanged classes, classes with renamed identifiers, changed classes, classes that could only be found in APK1/APK2 and unchecked classes. The numbers that are needed to visualize the results are calculated by APKCompare in the comparison steps and written to a file. The interactive pie charts are drawn by using the open-source JavaScript library *ChartJS*[11]. *ChartJS* is free to use and licensed under the MIT license. This library also supports many other types of charts, but in our project it is only used to display pie charts.

In order to fill the categories in the file list, files of the CSV format have to be read. For each category, APKCompare writes at least one CSV file, and depending on the applications that are compared with each other, such files can have up to several thousand lines. Therefore, we have been looking for a JavaScript library that allows quick parsing of CSV files and we could find a suitable one, namely *Papa Parse*[12]. This library is freely available and licensed under the MIT license. In our project, we use this library to parse CSV files and fill the file list in the web view accordingly. Even with CSV files of several gigabytes in size, *Papa Parse* performs well, which makes this library a very suitable match for our web interface.

The structure of generated CSV files is deliberately kept simple. Each line holds one match that results in one line in the file table. All values are separated by the character `;`, while the first two values describe the relative paths of the files or folders. For folders, the number of subfolders and files that are described by the given paths is told by the third and fourth values of the line, for files these values are always zero of course. Some of the resulting CSV files, more precisely exactly these that describe results of code comparison on method level, also have four more values per line. These values describe the numbers of matching methods and basic blocks, with and without identifiers renamed. An example of such an entry is provided here. This entry of a CSV file results in a web view entry that can be seen in the top of Figure 4.5 on page 38.

```
CodeCompare/original1/a/b/y.smali;
CodeCompare/original2/ff/gg/a.smali;
0;0;1/4;1/3;2/4;1/2
```

Another open source JavaScript library that is used in our web view is named *Mergely*[13]. This library allows viewing changes between documents directly in a web browser. The exact changes that are shown do not have to be provided beforehand, but can be detected by *Mergely* when two strings are provided. In our implementation, the contents of the left-hand side and right-hand side diff views are replaced by the content of files whenever an entry in the file list is clicked, and the detection of differences and the update of the view is triggered right afterward. *Mergely* also provides functionality to merge files, but we do not use it as APKCompare is

[11]https://www.chartjs.org/

[12]https://www.papaparse.com/

[13]http://www.mergely.com/

intended to be a tool for inspecting similarities and changes only. The diff view fits exactly for our needs, as it is intuitive and looks immediately familiar to people that are well-acquainted with other diff tools. Next to the highlighting of changed, added and removed content, this library also allows to search in displayed files if according add-ons are used. *Mergely* is used in our web view whenever two files that are not images are compared side by side.

A side by side approach is also used for the comparison of images in our approach. To do this, no external library was used, however, but the images are simply shown next to each other, which allows straightforward manual comparison. An example of a side by side comparison of two image files can be seen in Figure 4.9 on page 42.

Depending on the applications that have been compared with each other, there might be a large number of entries in the web view and finding relevant entries can therefore be a difficult task. As a solution, we implemented two functionalities that allow for a more clear view of results. On the one hand, all categories in the file list are collapsible, which allows hiding all matches of a certain category, for example unchanged resources, very quickly. On the other hand, we implemented a feature that allows filtering the list of files. This feature turned out to be very convenient, for example when a user is looking for a specific file, but also for only showing files of a certain file format for example.

Another feature that turned out to be very convenient is the comparison of arbitrary files of the given applications. While the comparison of files that have been identified as matches by APKCompare will be sufficient in most circumstances, sometimes it can be useful to compare other files side by side. Our web view allows to right-click entries in the list of matches and set them manually as files to display on the left or the right side. With this feature, it is also possible to compare two files of the same application with each other. We used JavaScript to implement this feature.

The feature that allows users to select if they prefer viewing source code in Smali or Java representation was also implemented in JavaScript. In both representations, a file always describes a class, which allows an easy mapping of files that describe the same class. Because of that fact and due to a smart directory structure, it is sufficient to replace a tiny part of the file paths to receive the path of the same class in the appropriate representation. For resource files, the selection of the favorite representation does not have any effect.

Chapter 6

Evaluation

In this chapter, we evaluate how well our tool, APKCompare, can be used to detect similarities and differences of two given APK files.

Section 6.1 covers the evaluation of the detection of specific changes that we have made intentionally. First, we have a look at changed, added and deleted resources, and then we evaluate how well code modifications can be detected. In Section 6.2, APKCompare is evaluated by comparing publicly available Android applications. First, we compare different versions of two applications in order to investigate if the provided changelog information is truthful or not. Then, we compare a downloaded application against a repackaged version of the same app that might contain malware and check if our tool can present all relevant differences. Sections 6.3 and 6.4 deal with the limitations that attracted our attention in the evaluation process, and present our general conclusion of the evaluation results.

6.1 Detection of Specific Changes

In this section, we will evaluate our tool by manually creating different variants of APK files and checking if the integrated changes can be detected by APKCompare. In order to compare applications of typical size, we did not create our own application, but we used a publicly available one for our evaluation approach, namely the *GLT Companion*[1] application, which is an Android app that was developed for the "Grazer Linuxtage" conference in Graz. The source code of said application is freely available on Github[2] and was used to develop various variants of the application on our machine by using the build automation system *Gradle*. For most variants, *ProGuard* (see Section 2.4) was used as a shrinking and obfuscation tool. For all comparisons of APK files in this section, we used an unmodified version of *GLT Companion* as the reference application.

6.1.1 Resource Comparison

To evaluate the detection of modified resources, we built two additional variants of the *GLT Companion* application. In one variant, no resource files have been added or deleted, but several files have been modified. The other variant contains new resource files like images, and some files

[1]https://play.google.com/store/apps/details?id=at.linuxtage.companion

[2]https://github.com/linuxtage/glt-companion

have been deleted on purpose. In both cases, we did not modify the build process, but the standard configuration that was found in the git repository of *GLT Companion* has been used. This means that also the *ProGuard* option `shrinkResources true` was in use when the APK files were built. When APKCompare was executed to determine similarities and differences, the command line argument `skipCodeComparison` was used as we are not interested in code changes in this evaluation step.

6.1.1.1 Detection of Changed Resources

In order to evaluate how well APKCompare can be used to detect unchanged and changed resources, we applied the following changes before the modified APK file which was compared against the unmodified one could be built:

- In `AndroidManifest.xml`, we made one change that defines the requirement of a touch-screen.
- We changed the orientation definition of a layout by modifying a value in `layout/fragment_tracks.xml` from vertical to horizontal.
- We edited the image `ic_launcher.png` in the folders `drawable-mdpi`, `drawable-hdpi`, `drawable-xhpi` and `drawable-xxhdpi`. The green hat in the image was replaced by a yellow one.
- We renamed one image file that existed in four folders to `time.png`, but we did not modify it.
- In `layout/fragment_event_details.xml`, we had to update two values because of the file renaming mentioned above.

When APKCompare is used to compare the APK file of the unmodified application (APK1) with the APK file with above-listed changes (APK2), the quick overview in the generated web interface already shows that the vast majority of resources is unchanged, that no resource files have been added or deleted, and that only ten files have been changed. These results are provided in the pie chart in Figure 6.1. The chart is similar to that in the web interface, however, we decided to present an adapted version for better readability. When having a look at the file list which can be seen in Figure 6.2, more detailed information is provided, and we can inspect that all files affected by changes that are described above, except the renamed files, are listed in the category *Changed Files*. Said renamed files can also be found in the web interface, but as the content is unmodified, they are listed in the category *Unchanged Files*, as it can be seen in Figure 6.3.

When comparing the listed files side by side, we can see that also the specific changes in the resource files are shown correctly. We highlight some the changes of the file `AndroidManifest.xml` in Listing 6.1 and present the side by side comparison view of the image `drawable-xxhdpi/ic_launcher.png` in Figure 6.4.

Next to the files that have been edited manually, also one more resource file is listed as a changed one by APKCompare. This file, `values/public.xml`, was not edited by us but actually differs in content because other resource files have been changed. Therefore, the categorization of said file is correct as well. Some of the changes that have been made in this file are presented in Listing 6.2.

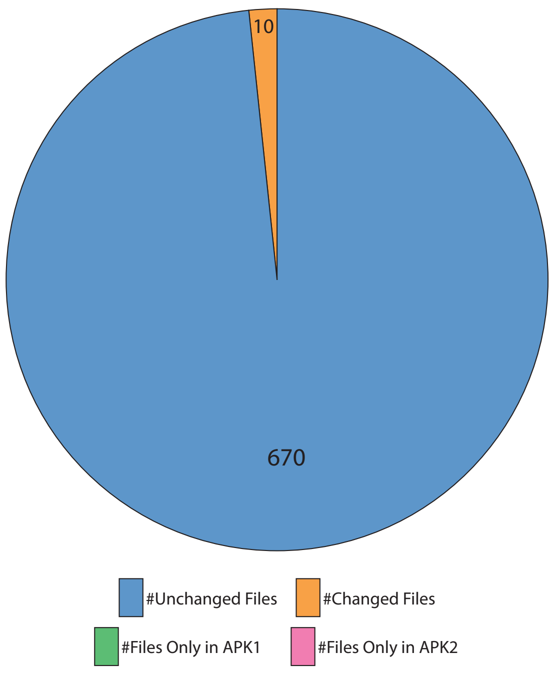


Figure 6.1: Quick overview of resource comparison results

6.1.1.2 Detection of Added and Deleted Resources

In this evaluation scenario, the approach was very similar to the above described one, but instead of modifying resource files, we added some files and deleted some others. To verify if all changes are detected by APKCompare, we will list the specific files again and check the results afterward:

- We deleted the images
  `room_gap149013.png`, `room_gap149017.png` and `room_gap149042.png`
  from the folder `res/drawable-hdpi`. These are not files of specific relevance, but we picked them randomly.

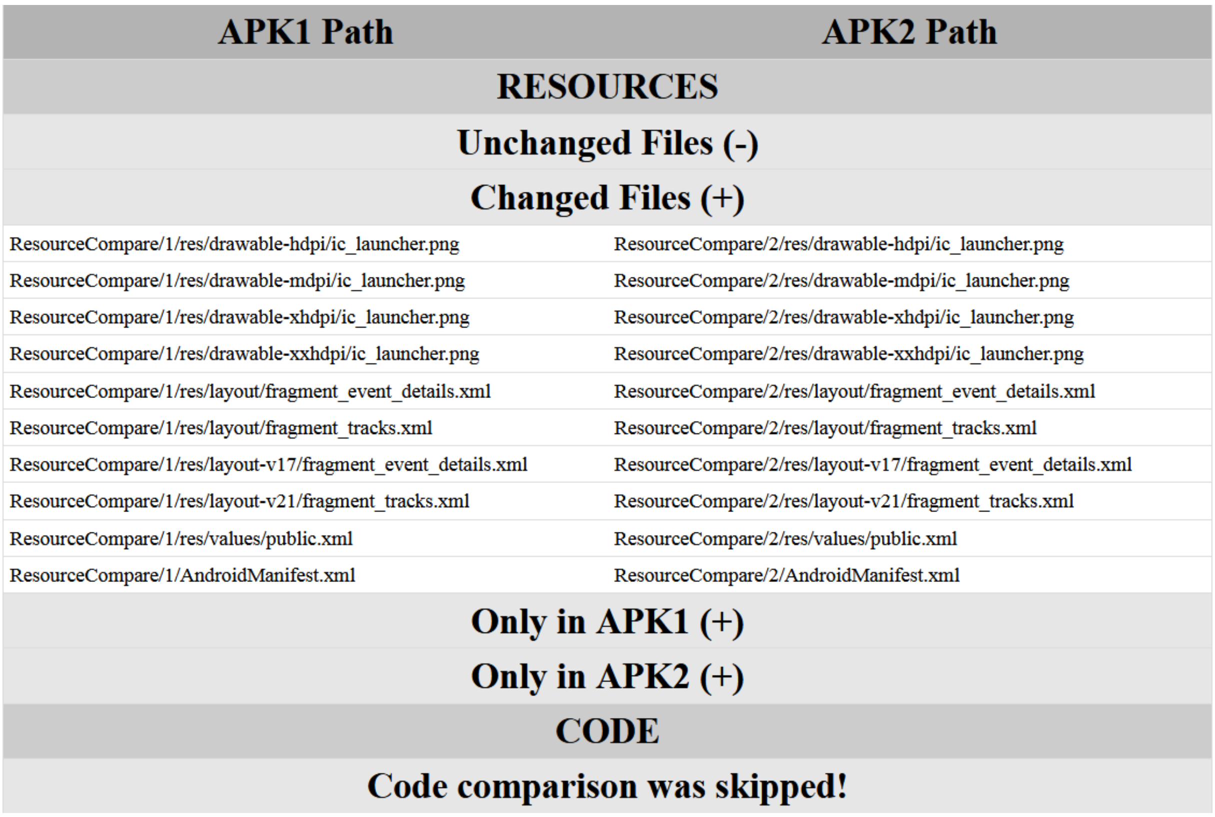
- In `res/drawable-xxxhdpi`, we added two files:
  `room_new_file_1.png` and `room_new_file_2.png`.

- We added three new files to `res/drawable-hdpi`:
  `room_new_file_3.png`, `room_new_file_4.png` and `room_new_file_5.png`.

- We created the file `res/values/special_colors.xml`. This file holds the definition of a newly added color:
  `<color name="my_special_color">#1ABC9C</color>`

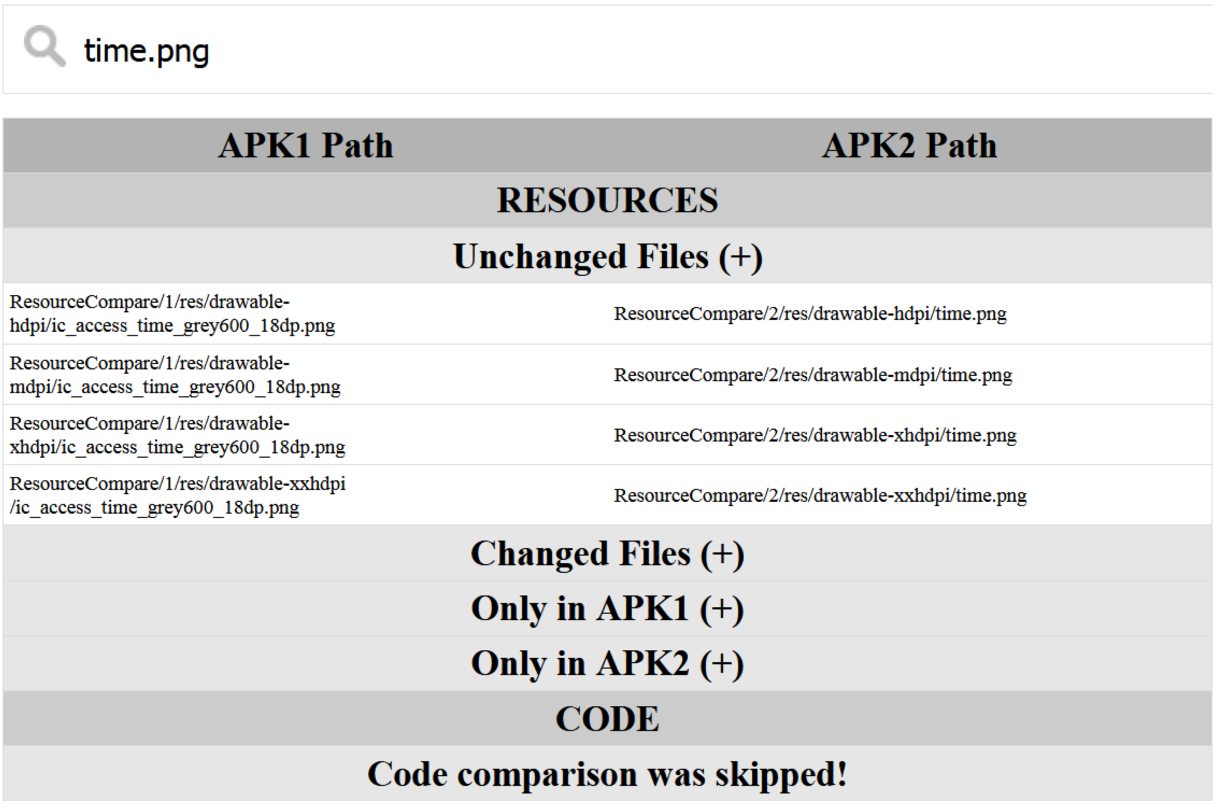
In the resulting web interface, we have a look at the resource list again. In Figure 6.5, which shows all entries except those of the category *Unchanged Files*, we can see that all three files listed in the category *Only in APK1* have been identified correctly. The files that have been deleted are also listed correctly, they appear in the category *Only in APK2*.

However, two files are also identified as changed ones, which looks like an error at first glance. Nevertheless, these files are also listed correctly. Because of changed resources, the file `values/public.xml` does not hold the same values anymore, and because of an optimization process, the color value that was defined in `res/values/special_colors.xml` was merged into the file `res/values/colors.xml`. As you can see in Listing 6.3, the entry *my_special_color* and the according value can be found in this file now.



APK1 Path	APK2 Path
RESOURCES	
Unchanged Files (-)	
Changed Files (+)	
ResourceCompare/1/res/drawable-hdpi/ic_launcher.png	ResourceCompare/2/res/drawable-hdpi/ic_launcher.png
ResourceCompare/1/res/drawable-mdpi/ic_launcher.png	ResourceCompare/2/res/drawable-mdpi/ic_launcher.png
ResourceCompare/1/res/drawable-xhdpi/ic_launcher.png	ResourceCompare/2/res/drawable-xhdpi/ic_launcher.png
ResourceCompare/1/res/drawable-xxhdpi/ic_launcher.png	ResourceCompare/2/res/drawable-xxhdpi/ic_launcher.png
ResourceCompare/1/res/layout/fragment_event_details.xml	ResourceCompare/2/res/layout/fragment_event_details.xml
ResourceCompare/1/res/layout/fragment_tracks.xml	ResourceCompare/2/res/layout/fragment_tracks.xml
ResourceCompare/1/res/layout-v17/fragment_event_details.xml	ResourceCompare/2/res/layout-v17/fragment_event_details.xml
ResourceCompare/1/res/layout-v21/fragment_tracks.xml	ResourceCompare/2/res/layout-v21/fragment_tracks.xml
ResourceCompare/1/res/values/public.xml	ResourceCompare/2/res/values/public.xml
ResourceCompare/1/AndroidManifest.xml	ResourceCompare/2/AndroidManifest.xml
Only in APK1 (+)	
Only in APK2 (+)	
CODE	
Code comparison was skipped!	

Figure 6.2: APKCompare's overview of changed resource files



time.png

APK1 Path	APK2 Path
RESOURCES	
Unchanged Files (+)	
ResourceCompare/1/res/drawable-hdpi/ic_access_time_grey600_18dp.png	ResourceCompare/2/res/drawable-hdpi/time.png
ResourceCompare/1/res/drawable-mdpi/ic_access_time_grey600_18dp.png	ResourceCompare/2/res/drawable-mdpi/time.png
ResourceCompare/1/res/drawable-xhdpi/ic_access_time_grey600_18dp.png	ResourceCompare/2/res/drawable-xhdpi/time.png
ResourceCompare/1/res/drawable-xxhdpi /ic_access_time_grey600_18dp.png	ResourceCompare/2/res/drawable-xxhdpi/time.png
Changed Files (+)	
Only in APK1 (+)	
Only in APK2 (+)	
CODE	
Code comparison was skipped!	

Figure 6.3: Filtered overview of unchanged resource files

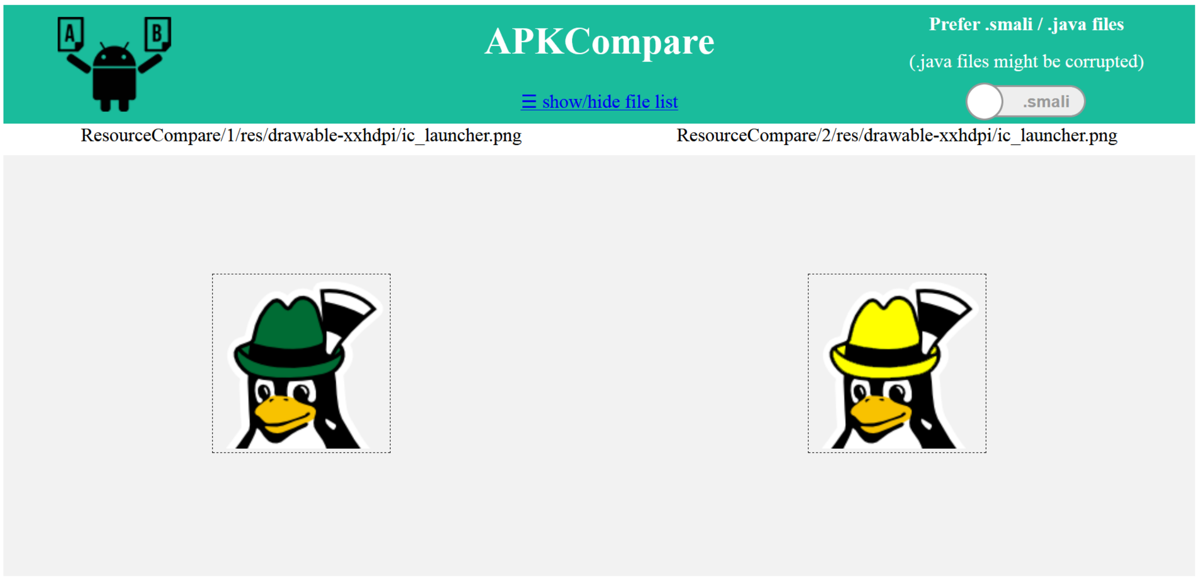

APKCompare
Prefer .smali / .java files
(.java files might be corrupted)
☰ show/hide file list
.smali
ResourceCompare/1/res/drawable-xxhdpi/ic_launcher.png
ResourceCompare/2/res/drawable-xxhdpi/ic_launcher.png

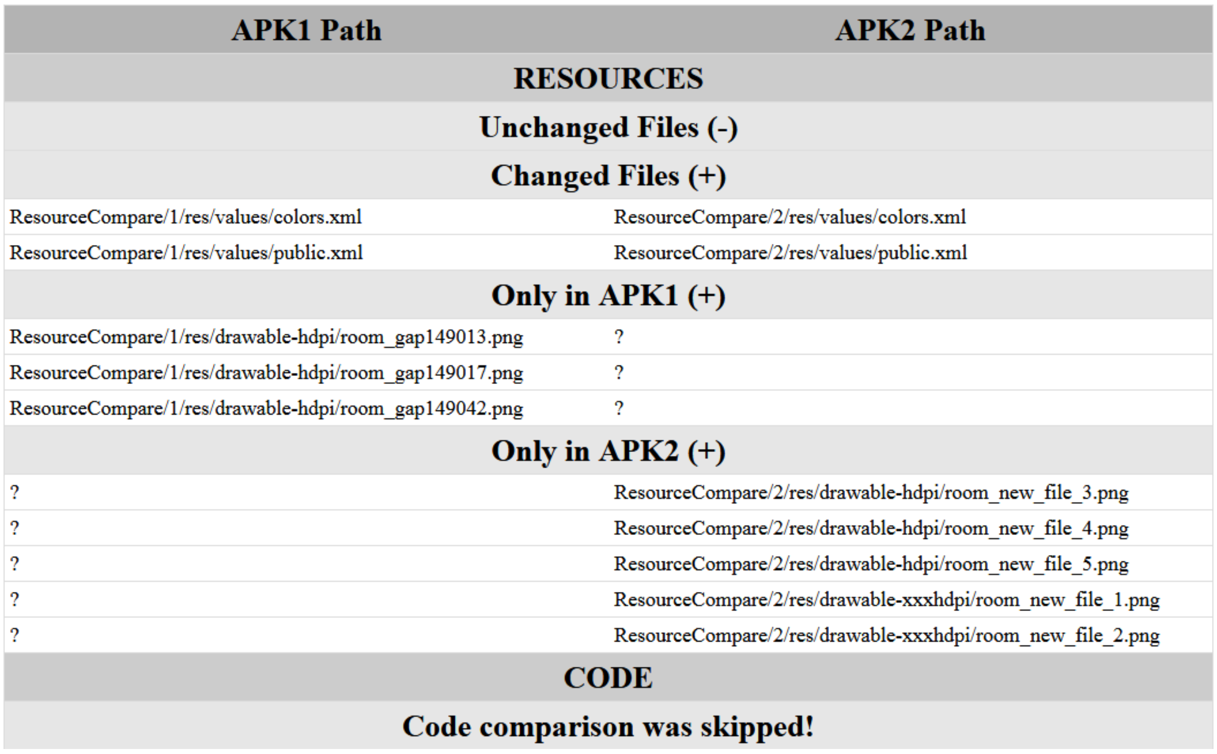
Figure 6.4: Side by side comparison of a modified image

```
@@ diff: AndroidManifest.xml <-> AndroidManifest.xml
   ...
   <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
   <uses-permission android:maxSdkVersion="18" android:name="android.permission.VIBRATE"/>
-  <uses-feature android:name="android.hardware.touchscreen" android:required="false"/>
+  <uses-feature android:name="android.hardware.touchscreen" android:required="true"/>
   <application android:allowBackup="true" android:icon="@drawable/ic_launcher" android:label
       ="@string/app_name" android:name="at.linuxtage.companion.GLTApplication" android:
       supportsRtl="true" android:theme="@style/AppTheme">
       <activity android:label="@string/app_name" android:name="at.linuxtage.companion.
           activities.MainActivity" android:theme="@style/AppTheme.NoActionBar.
           WindowDrawsSystemBarBackgrounds">
           <intent-filter>
               <action android:name="android.intent.action.MAIN"/>
               <category android:name="android.intent.category.LAUNCHER"/>
           </intent-filter>
   ...
```

Listing 6.1: Modifications that have been detected in *AndroidManifest.xml*

```
@@ diff: /res/values/public.xml <-> /res/values/public.xml
   ...
   <public type="drawable" name="fosdem_title" id="0x7f08006b" />
   <public type="drawable" name="glt_title" id="0x7f08006c" />
-  <public type="drawable" name="ic_avd_bookmark_24dp" id="0x7f08006e" />
+  <public type="drawable" name="ic_avd_bookmark_24dp" id="0x7f08006d" />
-  <public type="drawable" name="ic_bookmark_grey600_24dp" id="0x7f08006f" />
+  <public type="drawable" name="ic_bookmark_grey600_24dp" id="0x7f08006e" />
-  <public type="drawable" name="ic_bookmark_outline_white_24dp" id="0x7f080070" />
+  <public type="drawable" name="ic_bookmark_outline_white_24dp" id="0x7f08006f" />
-  <public type="drawable" name="ic_bookmark_white_24dp" id="0x7f080071" />
+  <public type="drawable" name="ic_bookmark_white_24dp" id="0x7f080070" />
-  <public type="drawable" name="ic_delete_white_24dp" id="0x7f080072" />
+  <public type="drawable" name="ic_delete_white_24dp" id="0x7f080071" />
   ...
   ...
```

Listing 6.2: Modifications that have been detected in *res/values/public.xml*



APK1 Path	APK2 Path
RESOURCES	
Unchanged Files (-)	
Changed Files (+)	
ResourceCompare/1/res/values/colors.xml	ResourceCompare/2/res/values/colors.xml
ResourceCompare/1/res/values/public.xml	ResourceCompare/2/res/values/public.xml
Only in APK1 (+)	
ResourceCompare/1/res/drawable-hdpi/room_gap149013.png	?
ResourceCompare/1/res/drawable-hdpi/room_gap149017.png	?
ResourceCompare/1/res/drawable-hdpi/room_gap149042.png	?
Only in APK2 (+)	
?	ResourceCompare/2/res/drawable-hdpi/room_new_file_3.png
?	ResourceCompare/2/res/drawable-hdpi/room_new_file_4.png
?	ResourceCompare/2/res/drawable-hdpi/room_new_file_5.png
?	ResourceCompare/2/res/drawable-xxxhdpi/room_new_file_1.png
?	ResourceCompare/2/res/drawable-xxxhdpi/room_new_file_2.png
CODE	
Code comparison was skipped!	

Figure 6.5: Overview of resource comparison results

```
@@ diff: res/values/colors.xml <->
@@       res/values/colors.xml
   ...
   <color name="material_grey_600">#ff757575</color>
   <color name="material_grey_800">#ff424242</color>
   <color name="material_grey_850">#ff303030</color>
   <color name="material_grey_900">#ff212121</color>
+  <color name="my_special_color">#ff1abc9c</color>
   <color name="notification_action_color_filter">#ffffffff</color>
   <color name="notification_icon_bg_color">#ff9e9e9e</color>
   <color name="primary_dark_material_dark">@android:color/black</color>
   <color name="primary_dark_material_light">@color/material_grey_600</color>
   ...
```

Listing 6.3: Modifications that have been detected in *res/values/colors.xml*

6.1.2 Code Comparison

Our approach to evaluating how properly code changes can be detected by our tool was very similar to the resource comparison evaluation strategy. We built different variants of the *GLT Companion* application before we checked if the specific changes can be seen on the result page that is provided by APKCompare.

In detail, we built four variants of the application. We label them as follows:

- **APK1** describes an unaltered version of *GLT Companion*. When building the application, *ProGuard*'s `minifyEnable` option was set to false, which means that the code was not obfuscated.
- **APK2** is not minified either, but code changes that are described below have been applied.
- **APK3** describes an unmodified version of the application. In this variant, *ProGuard*'s `minifyEnable` option was set to `true`, and classes, as well as packages, have been obfuscated in the naming scheme a-z, which means that identifiers have been replaced by 'a' if possible, followed by 'b' and so on.
- **APK4** is a minified and modified version. The same source code as for APK2 was used here. For obfuscation, z-a has been used as a naming scheme.

In the evaluation process, the not minified variants (APK1 and APK2) have been compared against each other, while the variants that have been built with *ProGuard* options (APK3 and APK4) have been used as files to compare. Before having a look at the results, we have to describe the **specific code changes** that we have made for APK2 and APK4. We decided upon these modifications as they can be seen as typical examples of code changes between two releases of the same application.

- **Introduction of a new class**

  We created a new class, `at/linuxtage/companion/parsers/TimeHelper.java`.
- **Introduction of a new method**

  In the class `at/linuxtage/companion/utils/StringUtils.java`, we added a new method, `toLower`.
- **Movement of an existing method**

  We moved two methods, namely `getHours` and `getMinutes`, from `at/linuxtage/companion/parsers/EventParser.java` to the newly created `TimeHelper` class.
- **Modification of existing methods**

  In three existing methods, we made some changes. These adjustments include modified values, different method calls, and other typical changes.
- **Introduction of a new field**

  In the class `at/linuxtage/companion/utils/StringUtils.java`, we introduced a field. This field is used in one of the methods that are defined in the class.

- **Modification of an existing field**

  In `at/linuxtage/companion/activities/SearchResultActivity.java`, we renamed the field with the name `MIN_SEARCH_LENGTH`, and the value was changed from 3 to 5.

- **Introduction of a new resource file**

  We added a new resource file, namely `res/drawable-xxxhdpi/room_new_file_1.png`. This was done so that resource identifier values in the file `res/values/public.xml` are different, which also results in changes in classes that use resources with new identifier values.

Now that we have defined the specific changes that have been made and the four variants that have been built, we can use APKCompare to evaluate how well the changes can be detected. The next section provides detailed information about the gathered comparison results.

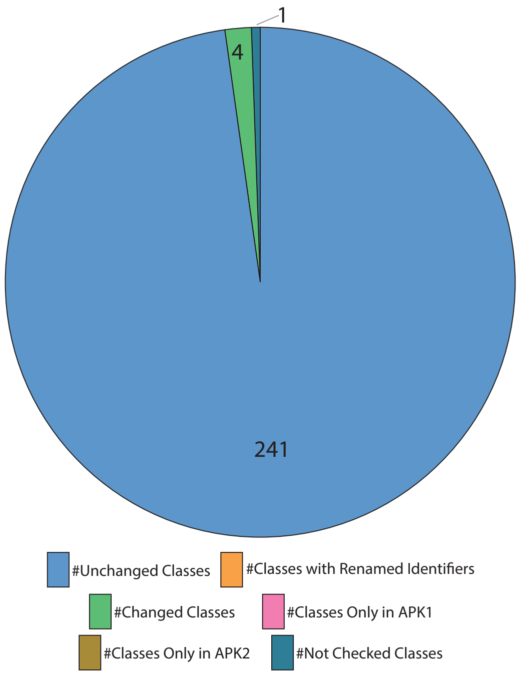
6.1.2.1 Detection of Code Modifications

In this section, we will try to evaluate if the code changes that have been mentioned above can be detected by APKCompare. To do that, we compare the not obfuscated application files (APK1 and APK2) and the obfuscated ones (APK3 and APK4) with each other and check if the modifications are listed. In both runs, APKCompare was parametrized in the following way:

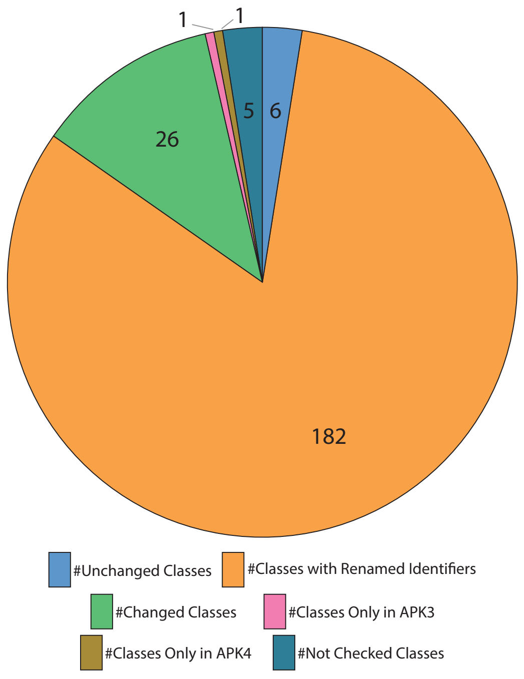
```
- methodInstructionsThreshold:     6
- basicBlockInstructionsThreshold: 3
- maxCodeComparisonRounds:         10
- excludeAndroidClasses:           true
- replaceResolvedResources:        true
- replaceRegisters:                false
- rearrangeRegisters:              true
```

When we have a look at the quick code comparison overviews, which are provided in Figure 6.6, we can already see that the given APK files have high similarity in both runs. In 6.6a, which shows the quick overview of the not obfuscated variants, we can observe that nearly all classes, in numbers 241 of 246, are classified as unchanged. The chart also says that four classes are recognized as changed ones and that only one class was not checked because it did not fulfill the defined thresholds for method instructions and basic block instructions. The other figure, 6.6b, shows the quick overview of the APK3-APK4 source comparison results and tells much about the similarity of the application files as well. While eight classes are entirely unchanged, 182 only differ in identifier names and 26 more could be classified as changed ones. Only five classes have not been checked because of the given parameters, and two classes have been falsely classified as classes that can be found in only one of the two applications.

The presented pie charts already indicate that the source comparison results might match the specific changes quite well. However, a detailed look at the classes that are matched by APKCompare has to be made to make a qualified assessment of the comparison results. In such a detailed examination, we could verify that all of the code changes mentioned above can be detected in both result pages. While the results of the code comparison between APK1 and APK2



(a) APK1 - APK2 code comparison overview



(b) APK3 - APK4 code comparison overview

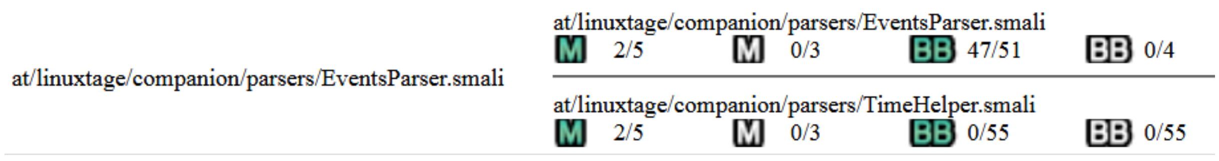
Figure 6.6: Overview of different code comparison results.
Figure (a) shows the quick overview of the comparison result of two APK files that have not been obfuscated. (APK1 - APK2)
Figure (b) shows the quick overview of the same applications, but built with two different obfuscation schemes. (APK3 - APK4)

represent precisely the changes that have been made, the results of the comparison between APK3 and the application that was modified and built with the usage of a totally different obfuscation naming scheme, APK4, also contains all made changes, but also lists some more classes as changed ones although they have not been modified on purpose. At this point, we also want to mention that the modification of values in the `/values/public.xml` file did not result in any wrong classifications, as APKCompare was called with the `replaceResolvedResources true` option.

Because of reasons of clarity, we do not provide all detected code changes in this section but only show and describe the most significant ones. Because of the same reason and for better readability, most of the provided diff views are shown in the Java representation.

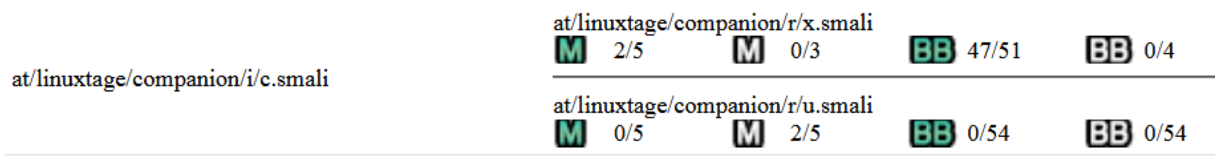
Figure 6.7 and the appertaining listings provide much information about the comparison results. In 6.7(a), which is taken from the APK1-APK2 comparison results, we can see that parts of the `EventsParser` class in APK1 can be found in the same class in APK2. It is also shown that some parts, in detail two of the five methods, have been moved to a class named `TimeHelper`. The next figure, 6.7(b), shows the same entry for the comparison of APK3 and APK4. As these application files have been built with different *ProGuard* obfuscation dictionaries, the file names do not match anymore. However, APKCompare was able to correctly determine that two methods have been moved here as well, as it can be seen in Listing 6.5, where the class named `c` in APK3 conforms to the `EventsParser` class, while `u` in APK4 conforms to `TimeHelper`. While Listing 6.4 (taken from the APK1-APK2 comparison) shows which methods have been removed from one class, Listing 6.5 (from APK3-APK4) shows that the same methods have been added to another.

Other comparison results that classify as very expressive ones can be found in Figures 6.8 and 6.9, and in Listing 6.6. The first of said figures lists the classes that could be matched on method level when comparing the files APK1 and APK2. We can see that the listed classes correspond to the code modifications that have been made. In Figure 6.9, we can see that APKCompare could match classes that are totally different in naming. Although the naming scheme a-z was used as obfuscation dictionary for APK3 and the exactly reverted dictionary was used for obfuscation in APK4, classes could be matched. The listing on page 70, Listing 6.6, shows changes that have been found when comparing APK1 with APK2. We can see that the method *toLower* was introduced in the shown class and that another method was modified so that the newly introduced method is called.



at/linuxtage/companion/parsers/EventsParser.smali	at/linuxtage/companion/parsers/EventsParser.smali			
	M 2/5	M 0/3	BB 47/51	BB 0/4
	at/linuxtage/companion/parsers/TimeHelper.smali			
	M 2/5	M 0/3	BB 0/55	BB 0/55

(a) Entry for the *EventsParser* class
left: APK1 entry right: APK2 entries



at/linuxtage/companion/i/c.smali	at/linuxtage/companion/r/x.smali			
	M 2/5	M 0/3	BB 47/51	BB 0/4
	at/linuxtage/companion/r/u.smali			
	M 0/5	M 2/5	BB 0/54	BB 0/54

(b) Entry for the *EventsParser* class
left: APK3 entry right: APK4 entries

Figure 6.7: Detection of moved methods.

(a) shows results on method level for the *EventsParser* class. We can see that two of the five methods in APK1 can be found in the same class in APK2, and that two methods have moved to the *TimeHelper* class.

In (b), we can also see the *EventsParser* class on the left, but because of obfuscation, it is named *c* in APK3. Although a different naming strategy was used to obfuscate code in APK4, APKCompare could detect that two of the five methods have moved to the *u* (= *TimeHelper*) class.

```
@@ diff: at/linuxtage/companion/parsers/EventsParser.java <->
@@       at/linuxtage/companion/parsers/EventsParser.java
   ...
   private String currentRoom;
   private Track currentTrack;

-  public static int getHours(String paramString)
-  {
-    return Character.getNumericValue(paramString.charAt(0)) * 10 +
-      Character.getNumericValue(paramString.charAt(1));
-  }
-
-  public static int getMinutes(String paramString)
-  {
-    return Character.getNumericValue(paramString.charAt(3)) * 10 +
-      Character.getNumericValue(paramString.charAt(4));
-  }

   protected boolean parseHeader(XmlPullParser paramXmlPullParser)
   ...
```

Listing 6.4: Modifications that have been detected in the *EventsParser* class. The red highlighted areas visualize lines that were in the class in APK1, but can not be found in APK2 anymore.

```
@@ diff: at/linuxtage/companion/i/c.java <->
@@       at/linuxtage/companion/r/u.java
   ...
-  public class c
-    extends d<at.linuxtage.companion.h.c>
+  public class u
   {
-  ...

-  public static int c(String paramString)
+  public static int a(String paramString)
   {
     return Character.getNumericValue(paramString.charAt(0)) * 10 +
       Character.getNumericValue(paramString.charAt(1));
   }

-  public static int d(String paramString)
+  public static int b(String paramString)
   {
     return Character.getNumericValue(paramString.charAt(3)) * 10 +
       Character.getNumericValue(paramString.charAt(4));
   }

-  ...
   }
```

Listing 6.5: Modifications that have been detected in the obfuscated versions of the classes *EventsParser* in APK3 (*c.java*) and *TimeHelper* in APK4 (*u.java*)

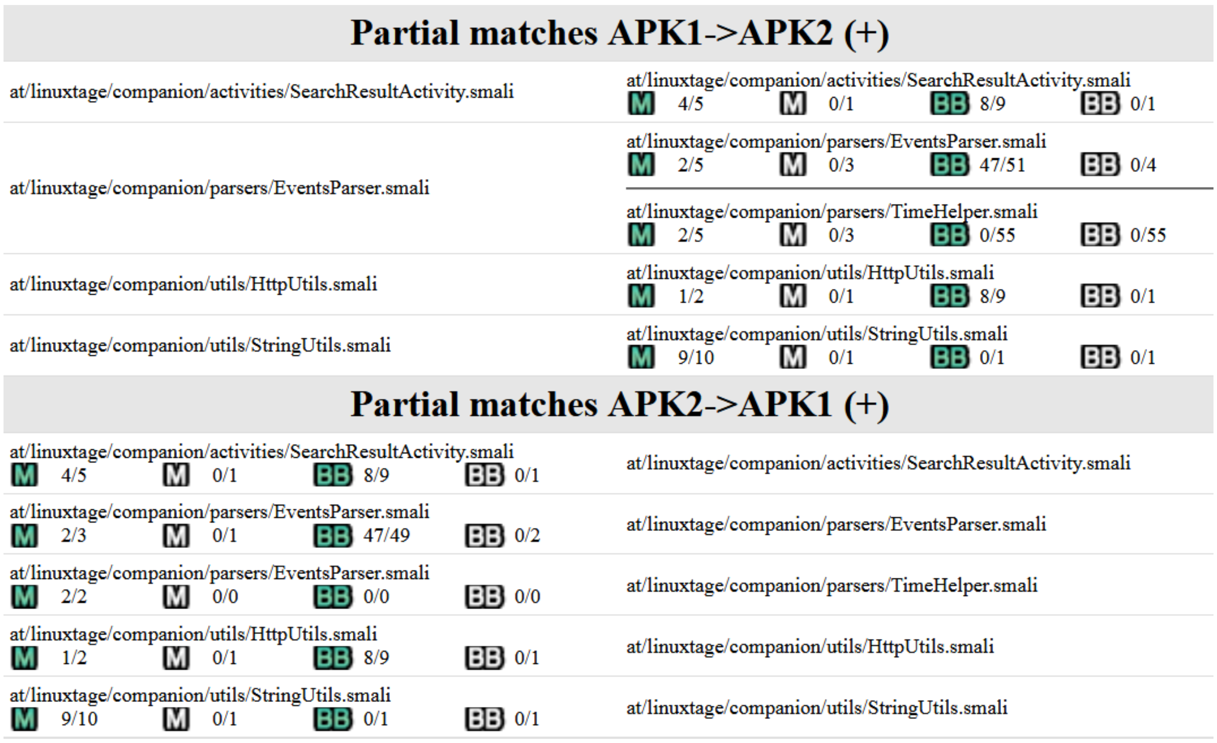
```
@@ diff: at/linuxtage/companion/utils/StringUtils.java <->
@@       at/linuxtage/companion/utils/StringUtils.java
   ...
   public static String stripHtml(@NonNull String paramString)
   {
     return trimEnd(Html.fromHtml(paramString)).toString();
   }

+  private static String toLower(@NonNull String paramString)
+  {
+    return paramString.toLowerCase(localeUS);
+   }

   public static String toSlug(@NonNull String paramString)
   {
-     return replaceNonAlphaGroups(trimNonAlpha(removeDiacritics(remove(paramString, '.')).
-       replace("ß", "ss")), '_').toLowerCase(Locale.US);
+     return toLower(
+       replaceNonAlphaGroups(trimNonAlpha(removeDiacritics(remove(paramString, '.')).
+         replace("ß", "ss")), '_'));
   }

   public static CharSequence trimEnd(@NonNull CharSequence paramCharSequence)
   ...
```

Listing 6.6: Modifications that have been detected in the *StringUtils* classes in APK1/APK2



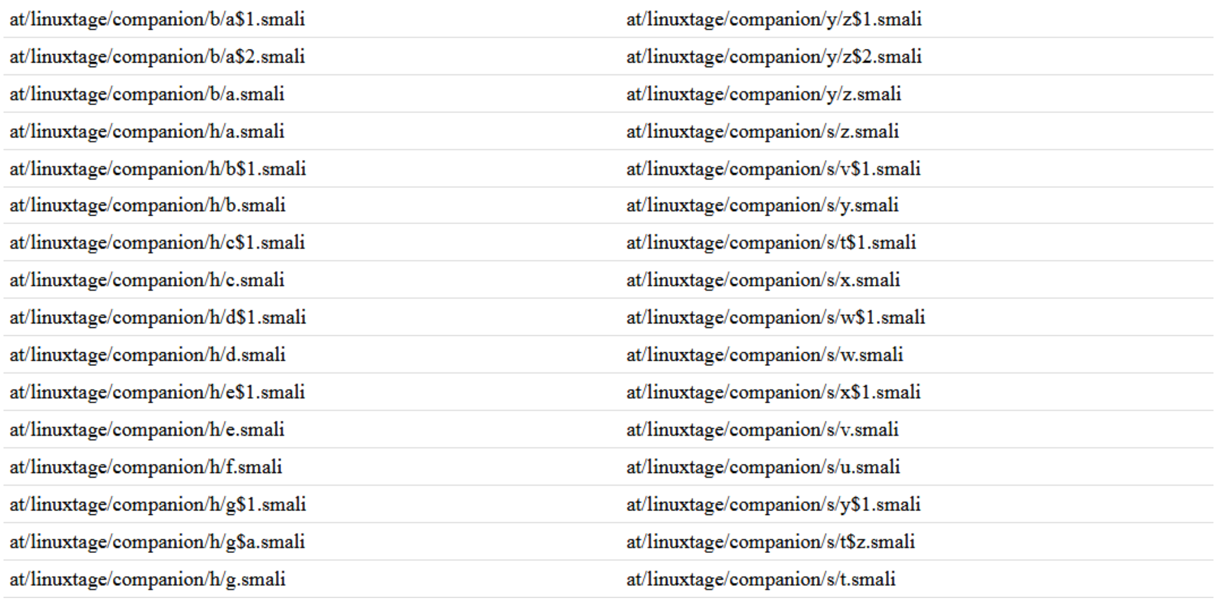
Partial matches APK1->APK2 (+)

at/linuxtage/companion/activities/SearchResultActivity.smali	at/linuxtage/companion/activities/SearchResultActivity.smali M 4/5 M 0/1 BB 8/9 BB 0/1
at/linuxtage/companion/parsers/EventsParser.smali	at/linuxtage/companion/parsers/EventsParser.smali M 2/5 M 0/3 BB 47/51 BB 0/4
	at/linuxtage/companion/parsers/TimeHelper.smali M 2/5 M 0/3 BB 0/55 BB 0/55
at/linuxtage/companion/utils/HttpUtils.smali	at/linuxtage/companion/utils/HttpUtils.smali M 1/2 M 0/1 BB 8/9 BB 0/1
at/linuxtage/companion/utils/StringUtils.smali	at/linuxtage/companion/utils/StringUtils.smali M 9/10 M 0/1 BB 0/1 BB 0/1

Partial matches APK2->APK1 (+)

at/linuxtage/companion/activities/SearchResultActivity.smali M 4/5 M 0/1 BB 8/9 BB 0/1	at/linuxtage/companion/activities/SearchResultActivity.smali
at/linuxtage/companion/parsers/EventsParser.smali M 2/3 M 0/1 BB 47/49 BB 0/2	at/linuxtage/companion/parsers/EventsParser.smali
at/linuxtage/companion/parsers/EventsParser.smali M 2/2 M 0/0 BB 0/0 BB 0/0	at/linuxtage/companion/parsers/TimeHelper.smali
at/linuxtage/companion/utils/HttpUtils.smali M 1/2 M 0/1 BB 8/9 BB 0/1	at/linuxtage/companion/utils/HttpUtils.smali
at/linuxtage/companion/utils/StringUtils.smali M 9/10 M 0/1 BB 0/1 BB 0/1	at/linuxtage/companion/utils/StringUtils.smali

Figure 6.8: List of classes that could be matched on method level.
The left side shows classes that could be found in APK1, on the right side we see the matching classes in APK2.



at/linuxtage/companion/b/a$1.smali	at/linuxtage/companion/y/z$1.smali
at/linuxtage/companion/b/a$2.smali	at/linuxtage/companion/y/z$2.smali
at/linuxtage/companion/b/a.smali	at/linuxtage/companion/y/z.smali
at/linuxtage/companion/h/a.smali	at/linuxtage/companion/s/z.smali
at/linuxtage/companion/h/b$1.smali	at/linuxtage/companion/s/v$1.smali
at/linuxtage/companion/h/b.smali	at/linuxtage/companion/s/y.smali
at/linuxtage/companion/h/c$1.smali	at/linuxtage/companion/s/t$1.smali
at/linuxtage/companion/h/c.smali	at/linuxtage/companion/s/x.smali
at/linuxtage/companion/h/d$1.smali	at/linuxtage/companion/s/w$1.smali
at/linuxtage/companion/h/d.smali	at/linuxtage/companion/s/w.smali
at/linuxtage/companion/h/e$1.smali	at/linuxtage/companion/s/x$1.smali
at/linuxtage/companion/h/e.smali	at/linuxtage/companion/s/v.smali
at/linuxtage/companion/h/f.smali	at/linuxtage/companion/s/u.smali
at/linuxtage/companion/h/g$1.smali	at/linuxtage/companion/s/y$1.smali
at/linuxtage/companion/h/g$a.smali	at/linuxtage/companion/s/t$z.smali
at/linuxtage/companion/h/g.smali	at/linuxtage/companion/s/t.smali

Figure 6.9: List of classes that could be matched on method level, even though the names are not equal because of code obfuscation.
On the left side, we see classes of APK3, the right side shows matching classes of APK4.

6.2 Case Study

In the previous section, we evaluated APKCompare by building different variants of an application and using our tool to check whether the modifications can be detected or not. Following this approach, we needed access to the source code of applications. In this section, however, we will use APKCompare to find similarities and differences of applications where the source code is not available. While we will use our tool to compare two versions of Android applications with each other in Section 6.2.1, we will manually edit an existing APK file in Section 6.2.2 and use APKCompare to compare the original APK file with the repackaged one.

6.2.1 Changelog Verification

In this section, we will evaluate APKCompare by comparing consecutive versions of Android applications in order to validate if the provided release notes are accurate or not. We will have a look at two popular applications, namely *Skype* and the password manager *1Password*. For both of them, the source code was not published, which means that only the changelog information and the results of APKCompare can be used for analysis. We could find security-relevant release notes for both of the applications and we will use APKCompare to verify if the changes that have been made actually match the statements in the changelogs.

6.2.1.1 Skype

In this setup, we chose the app versions 1.0.0.831 and 1.0.0.983 because the update information is very interesting, it says that a security issue was fixed, and also because the same versions have been used to evaluate the work of Desnos et al. [19] (see Section 3.1), which means that we can ascertain if APKCompare can also detect the changes that have been found by said team. Before having a look at the comparison result, we will describe the security issue in version 1.0.0.831, which is claimed to be fixed in the subsequent version.

As found out by *Android Police*[3], sensitive data like profile information (account balance, date of birth, email address and many more), information about contacts and even chat logs are saved in unencrypted form in the vulnerable version of *Skype*. Moreover, wrong access permissions have been chosen for these files, which allows other applications to read and write them. This is a significant security issue, but luckily it can be fixed easily by changing the file permissions of existing files and using appropriate permissions for new ones.

According to helloandroid.com, *Skype* announced the following text to introduce the updated version of the application:

> "After a period of developing and testing we have released a new version of the Skype for Android application onto the Android Market, containing a fix to the vulnerability reported to us. Please update to this version as soon as possible in order to help protect your information." [20]

To compare the two APK files with each other, we started APKCompare with default parameters. As it can be seen in Figure 6.10, the majority of resources and code of the applications was not modified. In detail, only 22 resource files are categorized as changed, while 1,558 are

[3]https://www.androidpolice.com/2011/04/14/exclusive-vulnerability-in-skype-for-android-is-exposing-your-name-phone-number-chat-logs-and-a-lot-more/

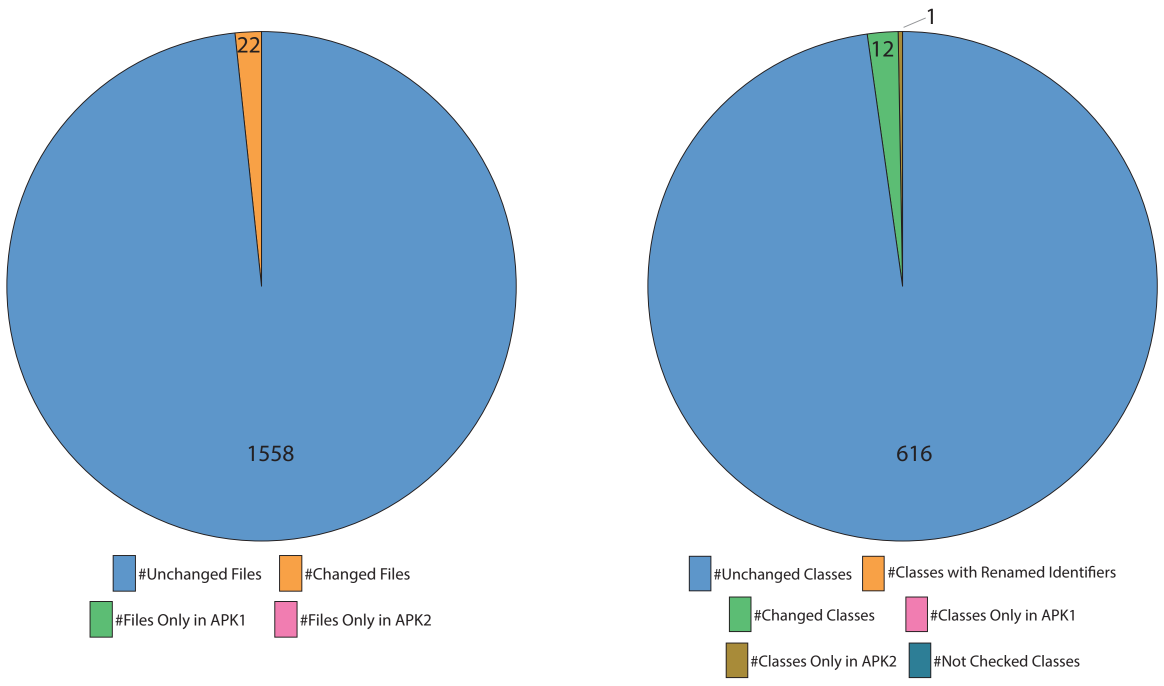


Figure 6.10: Overview of *Skype* comparison results.
On the left, an overview of the resource comparison results is presented. The right side shows the overview of the code comparison results.

unchanged, and 616 out of 629 classes are unmodified, 12 are categorized as changed, and a single class was only found in the newer version of the application.

A detailed look at the resources comparison result shows that the changed resources are negligible in this case. The majority of changes relates to changed strings in 17 languages, where every language is defined in a separate XML file, and the other changed resource files are also not of high relevance.

To investigate what has changed on the code level, we only have to analyze those classes that were categorized as changed or new ones by APKCompare. By doing this, we can find out that the changes which are relevant to the security issue have been made in the `com/skype/ipc/SkypeKitRunner` class. The most significant changes have been assembled in Listing 6.7, and can be summarized as follows:

As can be seen in the first area of the listing that shows fractions of the changes that have been detected in the `SkypeKitRunner` class, two new methods have been introduced in the updated version of *Skype*. These two methods, `fixPermissions(File[])` and `chmod(File, String)` are declared as private methods and are used to change access permissions of files that have been created with previous versions of the application. In the last fraction of Listing 6.7, we can see that also a call to the `fixPermissions` method was added. This call is made in the `run()` method of the class.

Two other important changes have also been made in the `run()` method of the `SkypeKit Runner` class, they are presented in the listing as well. The first mentioned change describes the modification of a method parameter. Instead of the value `3`, the value `0` is used as second parameter of the `openFileOutput(String, int)` method in the updated version of *Skype*. A look at the method reference[4] reveals that the value `0` describes the mode `MODE_PRIVATE`, while the value `3` is a combination of the modes `MODE_WORLD_READABLE` and `MODE_WORLD_WRITEABLE`.

A string which is used to define the access permissions of a file was also updated with the newer version. This modification can be seen in the listing as well. The value `777` (rwxrwxrwx) was replaced by `750` (rwxr-x—).

To conclude, we can say that APKCompare turns out to be a useful tool to compare these two given APK files with each other in order to verify the statement of the developers and that the security issue of version 1.0.0.831 was actually fixed in the subsequent version by changing access permissions.

[4]https://developer.android.com/reference/android/content/Context.html#openFileOutput(java.lang.String,%20int)

```
@@ diff: com/skype/ipc/SkypeKitRunner.smali <-> com/skype/ipc/SkypeKitRunner.smali
   ...

   .end method

+  .method private fixPermissions([Ljava/io/File;)V
+      .registers 7
+
+      array-length v0, p1
+
+      ...
+
+  .end method
+
+  .method private chmod(Ljava/io/File;Ljava/lang/String;)Z
+      .registers 7
+  ...

   ...

       const-string v6, "csf"

-      const/4 v7, 0x3
+      const/4 v7, 0x0

       invoke-virtual {v4, v6, v7}, Landroid/content/Context;->
         openFileOutput(Ljava/lang/String;I)Ljava/io/FileOutputStream;

   ...

      invoke-direct {v2}, Ljava/lang/StringBuilder;-><init>()V

-     const-string v4, "chmod 777 "
+     const-string v4, "chmod 750 "

      invoke-virtual {v2, v4}, Ljava/lang/StringBuilder;->
        append(Ljava/lang/String;)Ljava/lang/StringBuilder;

   ...

      move-result-object v1
+
+     move-object/from16 v3, p0
+
+     iget-object v3, v3, mContext:Landroid/content/Context;
+
+     move-object v2, v3
+
+     invoke-virtual {v2}, Landroid/content/Context;->getFilesDir()Ljava/io/File;
+
+     move-result-object v2
+
+     invoke-virtual {v2}, Ljava/io/File;->listFiles()[Ljava/io/File;
+
+     move-result-object v2
+
+     move-object/from16 v3, p0
+
+     move-object v18, v2
+
+     invoke-direct {v3, v18}, fixPermissions([Ljava/io/File;)V

      invoke-static {}, Ljava/lang/Runtime;->getRuntime()Ljava/lang/Runtime;

    ...
```

Listing 6.7: Relevant modifications of the *SkypeKitRunner* class that was updated in the *Skype* application. The code is presented in Smali representation.

6.2.1.2 1Password

1Password is a popular password manager application for Android. According to Google, this app has been installed more than 1,000,000 times[5]. The *1Password* application for Android exists in different versions, and for each version, release notes are available on the website of the developer team[6]. We studied the release notes and one of the versions, 6.4.1, particularly attracted our attention because security-relevant modifications are listed as reasons for the update:

> "In addition to several other improvements and fixes, this update addresses a number of issues with the 1Browser. In particular, we have updated 1Browser to use improved domain matching, default to HTTPS rather than HTTP, prevent loading of non-web URLs, and display informative dialogs when SSL errors occur. We want to extend a special thank you to Team SIK (www.team-sik.org) for disclosing these issues to us in a responsible manner." [21]

For three of the four separately addressed changes we could find the vulnerability reports of *Team SIK*[7] [8] [9] that describe problems and possible attack vectors of the application and that had a large impact on the update of the application obviously, as it can be read in the above-provided release notes.

In the following, we will address each of the four changes that are explicitly highlighted in the release notes of *1Password*'s version 6.4.1, and we will try to use the results provided by APKCompare to evaluate if the modifications have actually been made. To inspect and interpret the changes that have been made when updating from version *6.4 (build #58)* to *6.4.1 (build #59)*, we only have to look at those resource files and classes that are classified as modified or new by our tool. As we can see in Figure 6.11, which shows statistics about the comparison results, less than two percent of all classes have been modified and are therefore of special interest for us. Some of the 49 classes that have been classified as changed ones can be seen in Figure 6.12, which shows a screenshot taken from APKCompare's web interface.

- **Improved domain matching**

  In version 6.4 and also in prior versions as detected by *Team SIK*, the *1Password* application had a faulty behavior when inserting passwords into fields. In detail, the app did not consider subdomains when parsing URLs, which affected that credentials which were saved for a certain subdomain were also provided as credentials for other subdomains on the same domain. This is a major security issue, as potential attackers on the same domain could create web forms that save the credentials that actually belong to another subdomain.

  The error was made in the `AutologinActivity` class. In detail, the regular expression string that was used in the `showLogins` method, `([\\w\\d]*\\.)?(.*\\..*)`, is faulty. APKCompare correctly lists said class as a changed one, as it can be seen in Figure 6.12. The side by side comparison view of the `AutologinActivity` class visualizes that the regular expression is not used in the updated version anymore. Instead, the method `getLoginsForUrl` of the `Utils` class is called. By inspecting the `Utils` class, we can see that said method was newly added to the updated application. Internally, this method

[5]https://play.google.com/store/apps/details?id=com.agilebits.onepassword
[6]https://app-updates.agilebits.com/product_history/OPA4
[7]https://team-sik.org/sik-2016-038/
[8]https://team-sik.org/sik-2016-039/
[9]https://team-sik.org/sik-2016-041/

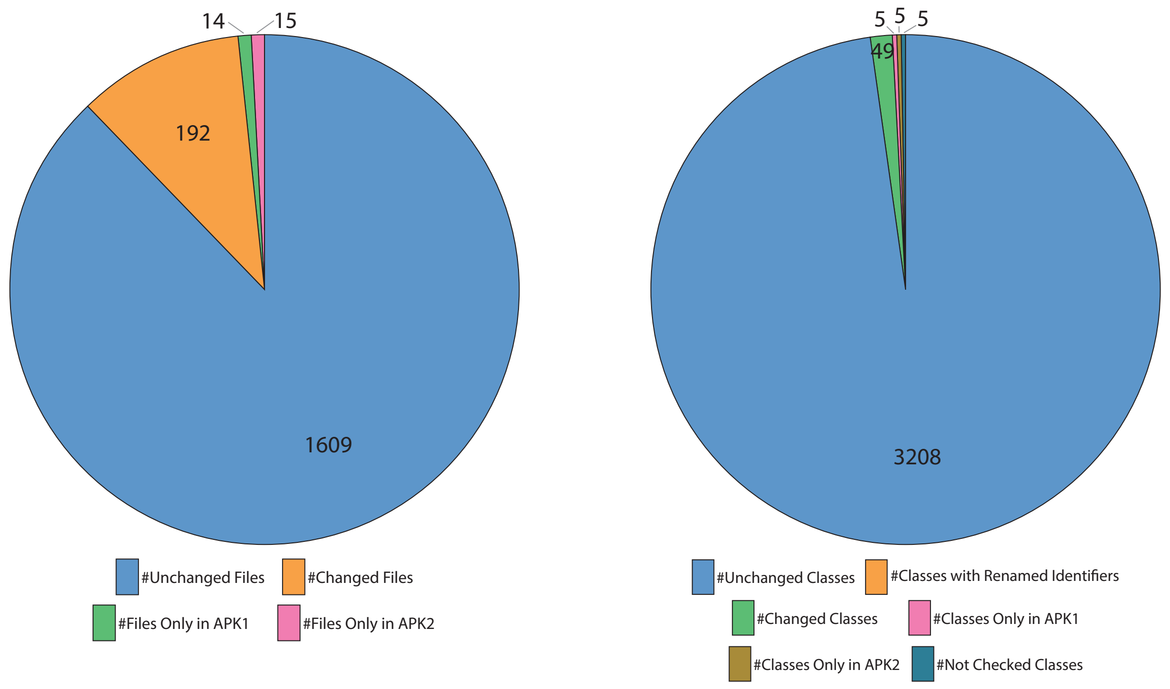
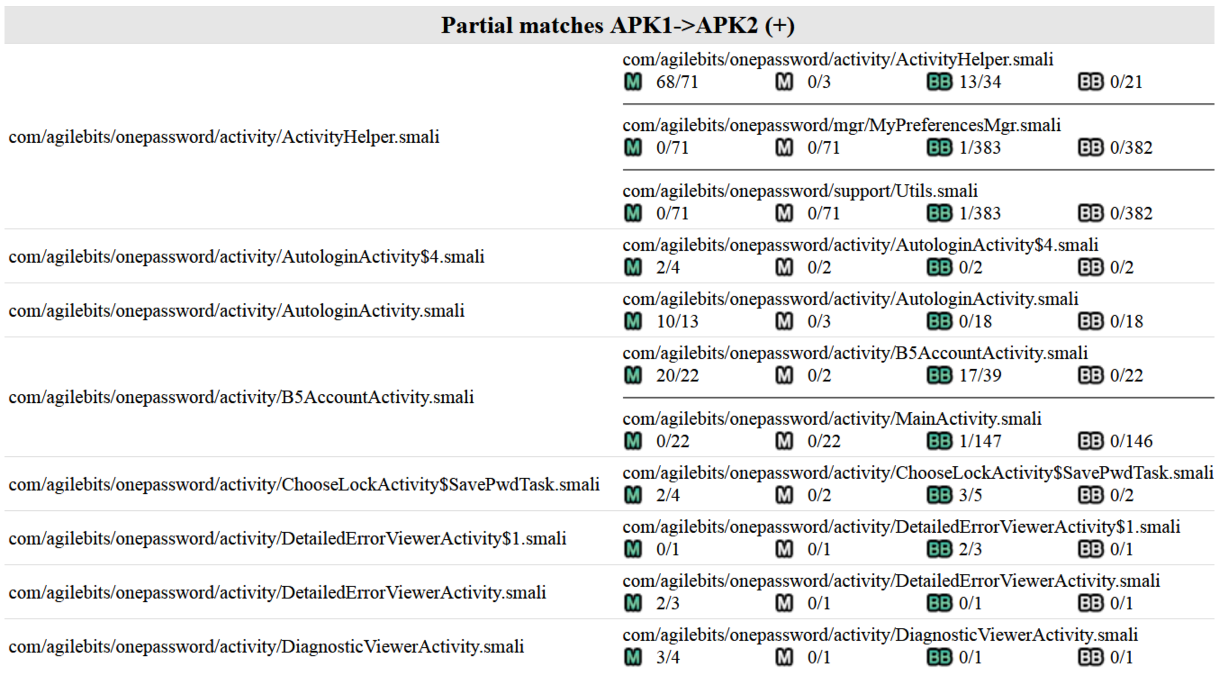


Figure 6.11: Overview of *1Password* comparison results.
On the left, an overview of the resource comparison results is presented. The right side shows the overview of the code comparison results.



Partial matches APK1->APK2 (+)					
com/agilebits/onepassword/activity/ActivityHelper.smali	com/agilebits/onepassword/activity/ActivityHelper.smali	M 68/71	M 0/3	BB 13/34	BB 0/21
	com/agilebits/onepassword/mgr/MyPreferencesMgr.smali	M 0/71	M 0/71	BB 1/383	BB 0/382
	com/agilebits/onepassword/support/Utils.smali	M 0/71	M 0/71	BB 1/383	BB 0/382
com/agilebits/onepassword/activity/AutologinActivity$4.smali	com/agilebits/onepassword/activity/AutologinActivity$4.smali	M 2/4	M 0/2	BB 0/2	BB 0/2
com/agilebits/onepassword/activity/AutologinActivity.smali	com/agilebits/onepassword/activity/AutologinActivity.smali	M 10/13	M 0/3	BB 0/18	BB 0/18
com/agilebits/onepassword/activity/B5AccountActivity.smali	com/agilebits/onepassword/activity/B5AccountActivity.smali	M 20/22	M 0/2	BB 17/39	BB 0/22
	com/agilebits/onepassword/activity/MainActivity.smali	M 0/22	M 0/22	BB 1/147	BB 0/146
com/agilebits/onepassword/activity/ChooseLockActivity$SavePwdTask.smali	com/agilebits/onepassword/activity/ChooseLockActivity$SavePwdTask.smali	M 2/4	M 0/2	BB 3/5	BB 0/2
com/agilebits/onepassword/activity/DetailedErrorViewerActivity$1.smali	com/agilebits/onepassword/activity/DetailedErrorViewerActivity$1.smali	M 0/1	M 0/1	BB 2/3	BB 0/1
com/agilebits/onepassword/activity/DetailedErrorViewerActivity.smali	com/agilebits/onepassword/activity/DetailedErrorViewerActivity.smali	M 2/3	M 0/1	BB 0/1	BB 0/1
com/agilebits/onepassword/activity/DiagnosticViewerActivity.smali	com/agilebits/onepassword/activity/DiagnosticViewerActivity.smali	M 3/4	M 0/1	BB 0/1	BB 0/1

Figure 6.12: Fraction of classes of the *1Password* application that could be matched on method level.
The left side shows classes that could be found in version 6.4, on the right side we see the matching classes of version 6.4.1.

calls another method, namely `registrableDomainForUrl` of the `PublicSuffix` class. With the provided side by side comparison view, we can quickly determine that also this method was updated and that the newly added method `parseURIFromUrl` of the `Utils` class is called internally. An inspection of the new code behavior shows us that the changes manage to fix the domain matching problem.

Because of reasons of clarity, we can not list all relevant code parts here, but some of the mentioned parts can be seen in Listings 6.8 and 6.9.

- **HTTPS rather than HTTP as default**

  Another issue that was claimed to be fixed in 6.4.1 relates to HTTP and HTTPS. In previous versions, the internal browser of the application used the HTTP scheme per default if no full URL was provided by the user. By inspecting the changed code, we could verify that the HTTPS scheme is actually used as the default scheme in the updated version, just like it is claimed by the developers. One of the relevant changes was made by adding the `paresURIFromURL` method. A tiny part of this method can be seen in Figure 6.9.

  Another change that relates to this topic can be investigated in Listing 6.10. We can see that the code which inserts the string *http://* to an empty field cannot be found in the updated version anymore. Moreover, we can see that the hint for URL fields was updated in the newer version. As Listing 6.12 shows, the edited hint string contains *https* instead of *http*.

- **Prevention of loading of non-web URLs**

  In older versions, users could read private data from the application folder by using URIs of the "file:///" scheme as URLs in the built-in web browser. Since version 6.4.1, this should not be possible anymore because only web URLs are allowed. By inspecting the changed code files that are detected by APKCompare, we can verify that this change was actually made. In the updated version, a dialog is displayed when non-web URLs are used as input. The code for this behavior can be found in Listing 6.11.

- **Informative dialogs when SSL errors occur**

  According to the release notes, more informative dialogs are shown when SSL errors occur in the new version. By inspecting the newly added class `CommonWebViewClient` and the newly added strings, which can be found in Listing 6.12, we can verify that also this modification was actually implemented by the developers.

Just as in the *Skype* comparison process, we started APKCompare with the default parameters and could quickly determine if the changelog information is trustworthy or not. We verified that all four modifications that have been highlighted in the release notes of *1Password 6.4.1 (build #59)* have actually been implemented.

```
@@ diff: com/agilebits/onepassword/support/PublicSuffix.java <->
@@       com/agilebits/onepassword/support/PublicSuffix.java
   ...

   public static String registrableDomainForUrl(String paramString)
   {
-    paramString = Utils.uriFromUrl(paramString);
+    paramString = Utils.parseURIFromUrl(paramString);
     if (paramString == null) {
       return null;
     }
     return registrableDomainForHost(paramString.getHost());
   }

   ...
```

Listing 6.8: Some of the relevant modifications that have been detected in the *PublicSuffix* class of the *1Password* application. The code is shown in Java representation.

```
@@ diff: com/agilebits/onepassword/support/Utils.java <->
@@       com/agilebits/onepassword/support/Utils.java
   ...

+  public static List<GenericItemBase> getLoginsForUrl(
+    List<GenericItemBase> paramList, String paramString)
+  {
+    paramString = PublicSuffix.registrableDomainForUrl(paramString);
+    ArrayList localArrayList = new ArrayList();
+    if ((paramList != null) && (!TextUtils.isEmpty(paramString)))
+    {
+      paramList = paramList.iterator();
+      while (paramList.hasNext())
+      {
+        GenericItemBase localGenericItemBase = (GenericItemBase)paramList.next();
+        if ((!TextUtils.isEmpty(mLocation)) &&
+          (paramString.equals(PublicSuffix.registrableDomainForUrl(mLocation)))) {
+          localArrayList.add(localGenericItemBase);
+        }
+      }
+    }
+    return localArrayList;
+  }

   ...

+  public static URI parseURIFromUrl(String paramString)
+  {
+    ...
+    return createURIFromUrlStr("https://" + paramString);
+  }

   ...
```

Listing 6.9: Modifications that have been detected in the *Utils* class of the *1Password* application. The code is shown in Java representation and presents two methods that have been added with the new version.

```
@@ diff: com/agilebits/onepassword/control/EditNodeUrl.java <->
@@       com/agilebits/onepassword/control/EditNodeUrl.java
   ...
   public EditNodeUrl(LinearLayout paramLinearLayout, ItemProperty paramItemProperty,
       GenericItem paramGenericItem)
   {
     super(paramLinearLayout, paramItemProperty);

     ...
     mDataView.setHint(2131231352); // = 0x7F080278
     // = value of the string resource "UrlHint" defined in values/strings.xml
     ...

-    mDataView.addTextChangedListener(new TextWatcher()
-    {
-      public void afterTextChanged(Editable paramAnonymousEditable)
-      {
-        if (paramAnonymousEditable.length() == 0) {
-          paramAnonymousEditable.insert(0, "http://");
-        }
-      }
-
-      public void beforeTextChanged(CharSequence paramAnonymousCharSequence,
-        int paramAnonymousInt1, int paramAnonymousInt2, int paramAnonymousInt3) {}
-
-      public void onTextChanged(CharSequence paramAnonymousCharSequence,
-        int paramAnonymousInt1, int paramAnonymousInt2, int paramAnonymousInt3) {}
-    });
     setOpenUrlEnabled(true);
   }
```

Listing 6.10: Relevant modifications that have been detected in the *EditNodeUrl* class of the *1Password* application. The code is shown in Java representation.

```
@@ diff: com/agilebits/onepassword/activity/AutologinActivity.java <->
@@       com/agilebits/onepassword/activity/AutologinActivity.java
   ...
   public void loadUrl(String paramString)
   {
-    paramString = Utils.uriFromUrl(paramString);
+    paramString = Utils.parseURIFromUrl(paramString);
     if (paramString != null) {
-      mWebView.loadUrl(paramString.toString());
+      mWebView.loadUrl(paramString.toASCIIString());
+      return;
     }
+    ActivityHelper.getAlertDialog(this, 2131231548, 2131231547).show();
   }
   ...
```

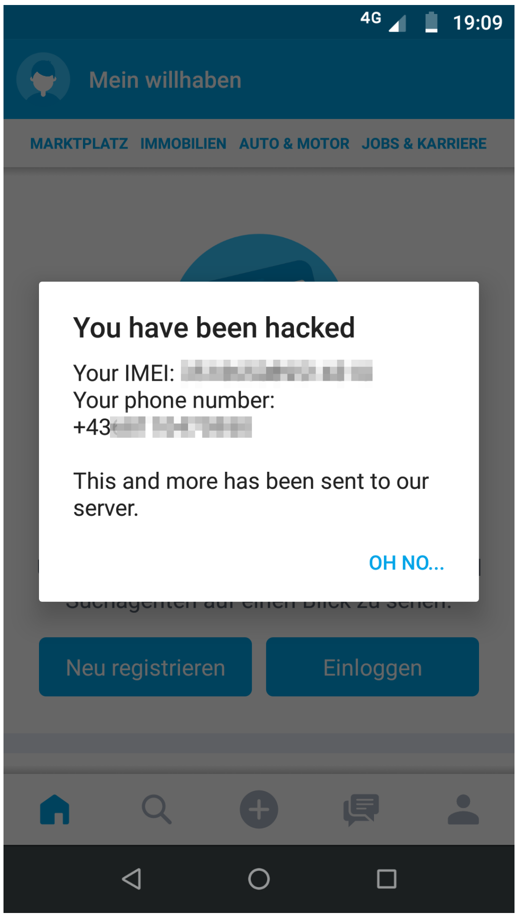
Listing 6.11: Modifications that have been detected in the *AutologinActivity* class of the *1Password* application. The listing shows code changes that cause the display of a dialog, among others.

```
@@ diff: values/strings.xml <->
@@       values/strings.xml
   ...
   <string name="UploadedTotalFilesMsg">Uploaded %1 files</string>
   <string name="UploadingFilesIntoKeychainMsg">Uploading files into vault</string>
-  <string name="UrlHint">http://www.example.com</string>
+  <string name="UrlHint">https://www.example.com</string>
   <string name="UseDefaultKeychainMsg">Select</string>
   <string name="UseFingerprint">Fingerprint Unlock</string>
   ...
   <string name="recommend_appUrl">market://details?id=com.agilebits.onepassword</string>
   <string name="security_pref_key">security</string>
+  <string name="ssl_date_invalid">The date of the certificate is invalid.</string>
+  <string name="ssl_error_msg">"%1$s couldn't be loaded because of an SSL error."</string>
+  <string name="ssl_error_title">Unable to load page</string>
+  <string name="ssl_expired">The certificate has expired.</string>
+  <string name="ssl_id_mismatch">The certificate hostname does not match.</string>
+  <string name="ssl_invalid">A generic error occurred.</string>
+  <string name="ssl_not_yet_valid">The certificate is not yet valid.</string>
+  <string name="ssl_untrusted">The certificate authority is not trusted.</string>
   <string name="sync_pref_key">sync</string>
   <string name="teams_pref_key">teams</string>
   ...
```

Listing 6.12: Relevant changed strings in the *1Password* application. The values can be found in in the *values/strings.xml* file.

6.2.2 Comparison of Repackaged Applications

There are many approaches to detect repackaged Android applications (see Section 3.2). This is justified by a large amount of repackaged applications, as it was shown by Potharaju et al. [12] and others. To determine if APKCompare can also detect specific changes of repackaged applications, we downloaded a widely spread Android application, namely *willhaben*, edited and repackaged it and compared the two versions with each other. The original application is available for free on the *Google Play* website and has been installed more than a million times according to *Google*[10].


You have been hacked

Your IMEI: [redacted]
Your phone number:
+43[redacted]

This and more has been sent to our server.

Figure 6.13: Screenshot of a repackaged Android application that contains malware

To manipulate the original application, we used the tool *Apktool* with the `decode` option, edited some of the resulting files and started *Apktool* with the `build` option to rebuild the application with the changes that have been made. In detail, only one XML file and one .smali file were edited, and one new class was added. However, these small changes have been enough to read the IMEI[11] as well as the phone number of the device, as Figure 6.13, which shows an original screenshot of the repackaged application running on an Android device, demonstrates. We will discuss the specific changes by having a look at the comparison results that are provided by APKCompare.

[10]https://play.google.com/store/apps/details?id=at.willhaben

[11]International Mobile Equipment Identity

For this comparison, the default parameters of APKCompare also resulted in meaningful outcomes, so no other comparisons with different parameters had to be started. To our surprise, 469 of the 2,084 resource files have been categorized as changed ones, despite only one file, namely `AndroidManifest.xml`, was modified on purpose.

The most relevant detected change in the `AndroidManifest.xml` file can be seen in Listing 6.13 and reveals that the repackaged application requests one permission more than the original one, namely `android.permission.READ_PHONE_STATE`.

Further investigations reveal that also another XML file, `res/values/public.xml`, is shown as a modified file correctly, as the positions of some values in the file have changed. They also illustrate that all remaining files that are categorized as changed ones are of the type PNG. It turns out that the PNG files are detected as changed ones because the hash values are not equal, although the images are identical in appearance. As the files that could be extracted from the repackaged application are bigger in size than those of the unchanged APK file, we assume that different compression techniques have been in use when adding the images to the APK files.

```
@@ diff: AndroidManifest.xml <-> AndroidManifest.xml
   <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
   <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
+  <uses-permission android:name="android.permission.READ_PHONE_STATE"/>
   <uses-permission android:name="android.permission.REORDER_TASKS"/>
   <uses-permission android:name="android.permission.CAMERA"/>
```

Listing 6.13: Changes that have been detected when comparing the *AndroidManifest.xml* files of the original and the repackaged application

By having a look at the changed resources, we could detect that one permission which is needed to read the state of the phone, for example the phone number, the status of ongoing calls, cellular network information and other data, was added to the modified application. In order to detect why this permission was added and what other modifications have been made, we have to look at the code changes that could be detected by APKCompare. It turns out that our tool does a good job at detecting code similarities and differences here as well. While 15,577 classes are categorized as unchanged, only one class is listed as a modified one (`FurbyActivity`), and another class, `MalwareHelper`, was detected as a newly added class. A detailed look at the presented side by side comparison shows that one method was added to the `FurbyActivity` class, and that exactly this method is invoked in the `onCreate()` method.

We present the code of the added method in Listing 6.14. It turns out that the code in this method is responsible for the dialog that can be seen in the above-presented screenshot, and a detailed study of the presented method discloses that the actual message that is shown comes from the `getPhoneInfo(Activity)` method, which is defined in the newly added class. For this class, we also present parts of the code; the described method can be found in Listing 6.15. As can be seen, a string is built in the newly introduced method, but no data is sent to a server, although this is stated in the message. The methods that are called by shown code also do not send data, but only return strings, which means that the repackaged application is fortunately not a malicious one and that APKCompare could be used to quickly investigate the differences between the original and the repackaged application.

```
@@ diff: at/willhaben/homescreen/FurbyActivity.smali <->
@@       at/willhaben/homescreen/FurbyActivity.smali
   ...
   .field private final l:Lat/willhaben/b/a;

   # direct methods
+  .method private showMalwareDialog()V
+      .registers 4
+
+      new-instance v0, Lat/willhaben/homescreen/MalwareHelper;
+
+      invoke-direct {v0}, Lat/willhaben/homescreen/MalwareHelper;-><init>()V
+
+      new-instance v1, Landroid/app/AlertDialog$Builder;
+
+      invoke-direct {v1, p0}, Landroid/app/AlertDialog$Builder;->
+        <init>(Landroid/content/Context;)V
+
+      const-string v2, "You have been hacked"
+
+      invoke-virtual {v1, v2}, Landroid/app/AlertDialog$Builder;->
+        setTitle(Ljava/lang/CharSequence;)Landroid/app/AlertDialog$Builder;
+
+      move-result-object v1
+
+      invoke-virtual {v0, p0}, Lat/willhaben/homescreen/MalwareHelper;->
+        getPhoneInfo(Landroid/app/Activity;)Ljava/lang/String;
+
+      move-result-object v0
+
+      invoke-virtual {v1, v0}, Landroid/app/AlertDialog$Builder;->
+        setMessage(Ljava/lang/CharSequence;)Landroid/app/AlertDialog$Builder;
+
+      move-result-object v0
+
+      const-string v1, "Oh no..."
+
+      const/4 v2, 0x0
+
+      invoke-virtual {v0, v1, v2}, Landroid/app/AlertDialog$Builder;->
+        setNegativeButton(Ljava/lang/CharSequence;
+        Landroid/content/DialogInterface$OnClickListener;)
+          Landroid/app/AlertDialog$Builder;
+
+      move-result-object v0
+
+      invoke-virtual {v0}, Landroid/app/AlertDialog$Builder;->
+        show()Landroid/app/AlertDialog;
+
+      return-void
+  .end method
+
   .method private final J()Lat/willhaben/store/o;
   .registers 4
   ...
```

Listing 6.14: Fraction of the *FurbyActivity* class that was found in the repackaged application. The code is presented in Smali representation. The green highlighted part was added.

```
@@ diff: (null) <->
@@         MalwareHelper.java
+  package at.willhaben.homescreen;
+
+  import android.app.Activity;
+  import android.content.Context;
+  import android.telephony.TelephonyManager;
+
+  public class MalwareHelper
+  {
+    private String getIMEI(Activity paramActivity)
+    {
+      ...
+    }
+
+    private String getPhoneNumber(Activity paramActivity)
+    {
+      ...
+    }
+
+    public String getPhoneInfo(Activity paramActivity)
+    {
+      StringBuilder localStringBuilder = new StringBuilder();
+      localStringBuilder.append("Your IMEI: ");
+      localStringBuilder.append(getIMEI(paramActivity));
+      localStringBuilder.append("\n");
+      localStringBuilder.append("Your phone number: ");
+      localStringBuilder.append(getPhoneNumber(paramActivity));
+      localStringBuilder.append("\n\n");
+      localStringBuilder.append("This and more has been sent to our server.");
+      return localStringBuilder.toString();
+    }
+  }
```

Listing 6.15: Fraction of the *MalwareHelper* class that was added to the repackaged application. The figure shows the whole *getPhoneInfo* method and the signatures of all other methods of this class. The code is shown in Java representation.

6.3 Limitations

While the above-listed evaluations, and many more which are not written down in this thesis, showed that APKCompare is a reliable tool to detect similarities and differences of two given APK files, we also detected some limitations that should not be neglected.

One of the limitations is due to the concept of hashing files and comparing the values to detect equal files. In some cases, small files resulted in the same hash values as their content was very generic, which resulted in the problem of wrongly matching files. For example, imagine that the files `/res/aa.xml` and `/res/bb.xml` are identical in content and exist in both given APK files. In this case, it might happen that the files are matched with the incongruous files of the other application each. While this is no error from a technical perspective as the file hashes are equal, a user would typically declare the files with identical paths as matches.

Also, images that are identical in appearance but different in the hash value would be typically identified as unchanged files by humans, but are listed as modified files by APKCompare. Again, this is correct from a technical point of view, but can be seen as a limitation in categorization as well.

Another limitation became apparent when comparing two seemingly similar applications. To our surprise, only a minority of classes were identified as matching ones by APKCompare, while many classes were listed as such that exist in only one of the APK files. It turned out that one of the applications contained numerous useless instructions, for example writing the same value to a register twice. These instructions do not have any practical effect on the behavior of the application, but lead to bad comparison results, as neither whole classes nor methods or basic blocks can be identified as similar ones as the useless instructions affect the hash values.

Moreover, we see a potential for improvement in the choosing process of reasonable parameters that are used when executing APKCompare. While the default parameters happen to result in reasonably good comparison results in most cases, there are some comparison scenarios where other parameters have to be chosen to gain good results, and sometimes multiple parameter combinations have been tested in order to find appropriate ones.

6.4 Summary of Evaluation Results

In this chapter, we evaluated APKCompare in different scenarios. As a first step, we tested how well our tool can be used to detect specific changes that have been made on purpose by building multiple versions of an application with known source code. Then, we tested APKCompare on two typical scenarios, namely the comparison of two consecutive versions of an application and the comparison of an unmodified and a repackaged version of the same app. In all the described scenarios, even when comparing two APK files that have been built with completely different obfuscation dictionaries, our tool was very reliable and managed to identify changes on resource and code level. Limitations that have been detected in the evaluation process have been collected in Section 6.3 and can be used as thought-provoking impulses for further improvements.

Chapter 7

Conclusion

The operating system Android is widely used and allows users to install arbitrary applications. Programs are typically distributed over the *Google Play Store* but can be downloaded from other sources as well. As applications are typically available in multiple versions, developers often provide notes about the reason for updates. However, verifying what has actually changed with the new version is a hard and time-consuming task that requires special foreknowledge. The comparison of two android applications is not only of relevance when looking for similarities and differences of two version of the same application, but also when analyzing repackaged applications, for example.

In this thesis, we presented the comparison tool APKCompare, which analyzes two given APK files and manages to detect differences and similarities between applications. The comparison results are presented in a well structured way, so that it is easier for users to analyze changes that have been detected and interpret the impact of the differences. APKCompare can be used to compare resource files of two given applications, but also to compare their code. Both tasks can be run independently, and both are based on the idea of hashing content and determining it as equal if the hash values are identical. Because of our particular code comparison approach that analyzes different parts of classes and determines the hash values of methods and basic blocks, APKCompare can detect even small changes that have been made. With a focus on naming-invariant comparison techniques, the tool is also resistant against basic code obfuscation techniques.

In the evaluation process, we could demonstrate that APKCompare is a useful tool to compare two Android applications with each other and present comparison results to the user. By comparing different types of APK files (ones that have been built ourselves in different variants, once downloaded files of applications in two versions, and an original and a repackaged app), we could show that APKCompare performed well when comparing resource files and code, even if obfuscation techniques had been in use.

With APKCompare, we implemented a fully functional tool to compare Android applications with each other. While we are already delighted about the outcome of our work, we have certain ideas that could still enhance our tool in functionality and usability:

- **Additional search functionality**
  Currently, the web interface allows collapsing the visibility of whole categories and filtering the list of matches. These functionalities allow users to get a quick overview of the comparison results and to navigate quickly through resource files and classes. However,

the web view currently does not allow to search for content in all files. Such functionality would not only be useful to list all files that contain certain parameter values, strings, or other values, but also to get an overview of all classes that call a certain method, for example.

- **Define multiple classes to exclude**
  APKCompare can be started with the `-excludeAndroidClasses true` option, which causes that all classes of the *android* package are not disassembled and therefore not considered in the code comparison steps. An additional option that provides the possibility to define a list of classes and packages that shall not be disassembled would allow potential better comparison results. Such an option could be used to exclude wrongly matched classes from the comparison process, for example, but also to exclude whole packages that are not of interest.

- **Removal of useless instructions**
  Smali instructions that do not affect the behavior of Android applications, for example storing a value in a register that is never used in the method, could be removed before comparison checks are performed. With this functionality, even better comparison results might be reached. Such functionality should be optional, as users might want to define if the original code or modified versions should be compared.

- **Exclude getters and setters from code comparison**
  Currently, options exist that define how many instructions methods and basic blocks must consist of in order to be relevant enough for comparison. Another similar idea is to determine getters and setters and give users the possibility to exclude these methods from the comparison.

- **Image comparison**
  Images that are equal in appearance but have different hash values are currently detected as changed files. Another category for such nearly equal images could be introduced, so that users can easily distinguish between actually changed image files and ones that are only classified as changed ones because of different hash values. Another idea regarding image comparison is to provide users with the ability to see the specific differences of images directly in the web interface. Javascript libraries like *pixelmatch*[1] or *Rembrandt.JS*[2] could be suitable for that approach.

[1]https://github.com/mapbox/pixelmatch
[2]https://github.com/imgly/rembrandt

Bibliography

[1] Android.com. *Configure Your Build | Android Studio*. 2018. `https://developer.android.com/studio/build/index.html`.

[2] Karim Yaghmour. *Embedded Android: Porting, Extending, and Customizing*. 1st. O'Reilly Media, Inc., 2013. ISBN 978-1-4493-0829-2.

[3] Johannes Feichtner. "CryptoSlice". Master's thesis. Graz University of Technology, May 2015. `https://diglib.tugraz.at/download.php?id=5891c7343eb38&location=browse`.

[4] Ben Gruver. *Smali*. 2014. `https://github.com/JesusFreke/smali`.

[5] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. "Statistical Deobfuscation of Android Applications". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: ACM, 2016, pages 343–355. ISBN 978-1-4503-4139-4. doi:10.1145/2976749.2978422. `http://doi.acm.org/10.1145/2976749.2978422`.

[6] Marc Stevens, Arjen K. Lenstra, and Benne De Weger. "Chosen-prefix Collisions for MD5 and Applications". In: *Int. J. Appl. Cryptol.* 2.4 (July 2012), pages 322–359. ISSN 1753-0563. doi:10.1504/IJACT.2012.048084. `http://dx.doi.org/10.1504/IJACT.2012.048084`.

[7] Anthony Desnos. "Android: Static Analysis Using Similarity Distance". In: *2012 45th Hawaii International Conference on System Sciences* (2012), pages 5394–5403. ISSN 1530-1605. doi:10.1109/HICSS.2012.114.

[8] Li Li, Tegawendé F Bissyandé, and Jacques Klein. "SimiDroid: Identifying and Explaining Similarities in Android Apps". In: *The 16th IEEE International Conference On Trust, Security And Privacy In Computing And Communications (TrustCom 2017)*. 2017.

[9] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. "Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot". In: *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*. SOAP '12. Beijing, China: ACM, 2012, pages 27–38. ISBN 978-1-4503-1490-9. doi:10.1145/2259051.2259056. `http://doi.acm.org/10.1145/2259051.2259056`.

[10] Kai Chen, Peng Liu, and Yingjun Zhang. "Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, pages 175–186. ISBN 978-1-4503-2756-5. doi:10.1145/2568225.2568286. `http://doi.acm.org/10.1145/2568225.2568286`.

[11] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. “WuKong: A Scalable and Accurate Two-phase Approach to Android App Clone Detection”. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ISSTA 2015. Baltimore, MD, USA: ACM, 2015, pages 71–82. ISBN 978-1-4503-3620-8. doi:10.1145/2771783.2771795. http://doi.acm.org/10.1145/2771783.2771795.

[12] Rahul Potharaju, Andrew Newell, Cristina Nita-Rotaru, and Xiangyu Zhang. “Plagiarizing Smartphone Applications: Attack Strategies and Defense Techniques”. In: *Engineering Secure Software and Systems*. Edited by Gilles Barthe, Benjamin Livshits, and Riccardo Scandariato. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pages 106–120. ISBN 978-3-642-28166-2.

[13] Hugo Gonzalez, Natalia Stakhanova, and Ali A. Ghorbani. “DroidKin: Lightweight Detection of Android Apps Similarity”. In: *International Conference on Security and Privacy in Communication Networks - 10th International ICST Conference, SecureComm 2014, Beijing, China, September 24-26, 2014, Revised Selected Papers, Part I*. 2014, pages 436–453. doi:10.1007/978-3-319-23829-6_30. https://doi.org/10.1007/978-3-319-23829-6_30.

[14] Christof Rabensteiner. “Android Library Identification”. Master’s thesis. 2017. http://diglib.tugraz.at/download.php?id=5988e795a35ec&location=browse.

[15] Michael Backes, Sven Bugiel, and Erik Derr. “Reliable Third-Party Library Detection in Android and its Security Applications”. In: *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS ’16)*. pub_id: 1066 Bibtex: derr:ccs:2016 URL date: None. Oct. 2016. https://publications.cispa.saarland/837/.

[16] Aisha Ali-Gombe, Irfan Ahmed, Golden G. Richard III, and Vassil Roussev. “OpSeq: Android Malware Fingerprinting”. In: *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*. PPREW-5. Los Angeles, CA, USA: ACM, 2015, 7:1–7:12. ISBN 978-1-4503-3642-0. doi:10.1145/2843859.2843860. http://doi.acm.org/10.1145/2843859.2843860.

[17] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. “Exploring the Influence of Identifier Names on Code Quality: An empirical study”. In: *14th European Conference on Software Maintenance and Reengineering*. Mar. 2010, pages 156–165. http://oro.open.ac.uk/19224/.

[18] Armstrong A. Takang, Penny Grubb, and Robert Macredie. “The effects of comments and identifier names on program comprehensibility: An experimental investigation”. In: *J. Prog. Lang.* 4 (Sept. 1996), pages 143–167.

[19] Anthony Desnos and Geoffroy Gueguen ESIEA. “Android : From Reversing to Decompilation”. In: 2011.

[20] helloandroid.com. *Skype security vulnerability fixed*. 2011. http://www.helloandroid.com/content/skype-security-vulnerability-fixed.

[21] agilebits.com. *1Password for Android Release Notes*. 2019. https://app-updates.agilebits.com/product_history/OPA4#v59.