Christof Rabensteiner BSc

Android Library Identification

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, February 2017

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date

Signature

Abstract

While Android developers integrate more and more libraries into their apps, certain libraries carry vulnerabilities or violate the user's privacy by forwarding sensitive information. Both tendencies create the need to identify libraries in apps to improve the risk assessment of apps and to separate app code from library code. This thesis presents ASTLI, a static program analysis tool that identifies third party libraries in Android apps. ASTLI learns and recognizes libraries by extracting and comparing features, which depend on abstract syntax trees (ASTs) and on signatures of methods. These features are designed to counter obfuscation, dead code removal and various code optimizations. We evaluate ASTLI with apps from an Open Source App Repository. Depending on the applied code transformations, between 96 and 97% (obfuscation, dead code removal), resp. 78% (optimizations) of our predictions are correct.

Keywords: Android, Apps, Third Party Libraries, Static Program Analysis, AST, Obfuscation, Reverse Engineering

Kurzfassung

Während App-Entwickler zunehmend die Funktionalität ihrer Anwendungen in Programmbibliotheken von Drittanbietern auslagern, weisen manche dieser Bibliotheken Sicherheitslücken auf oder verletzen die Privatssphäre der Benutzer, weil sie vertrauliche Daten weiterleiten. Diese Tendenzen erwecken das Bedürfnis, Bibliotheken in Apps zu erkennen um die Risikobewertung von Apps zu verbessern und um Apps von Bibliotheken zu trennen. Diese Arbeit präsentiert ASTLI, ein statisches Code-Analyse Werkzeug zur Bestimmung von Bibliotheken in Android Apps. Um Bibliotheken zu lernen und zu erkennen, extrahiert und vergleicht ASTLI Merkmale, welche auf abstrakten Syntaxbäumen und Signaturen von Methoden basieren. ASTLI kann dabei das Umbenennen von Debugsymbolen (*obfuscation*), die Entfernung von unerreichbarem Code und Code-Optimierungen bewältigen. Wir evaluieren ASTLI mit quelloffenen Apps und Bibliotheken. Abhängig von den angewandten Umformungstechniken liegt das Werkzeug bei 96% bis 97% (Umbennen, Entfernen von unerreichbarem Code), bzw. bei 78% (Optimierungen) der Vorhersagen richtig.

Keywords: Android, Apps, Bibliotheken, Statische Code-Analyse, Abstrakter Syntaxbaum, Obfuscation, Reverse Engineering

Contents

Contents v

List of Figures vii

List of Tables viii

1 Introduction 1

2 Android Ecosystem 5
- 2.1 Platform 5
- 2.2 Runtime 6
- 2.3 Build Process 8
- 2.4 Gradle 9
- 2.5 Proguard 9
- 2.6 File Format 11

3 Related Work 14
- 3.1 Code Based Plagiarism Detection 14
  - 3.1.1 App Repackaging 14
  - 3.1.2 Winnowing 15
  - 3.1.3 AST Distance 16
  - 3.1.4 Centroid 19
- 3.2 Library Detection 20
  - 3.2.1 Common Libraries 20
  - 3.2.2 LibScout 21

4 Approach 23
- 4.1 Requirements 23
- 4.2 Overcoming Obfuscation 24
  - 4.2.1 Features 24

4.2.2 Identifier Renaming . . . 25
4.2.3 Shrinking . . . 25
4.2.4 Optimizations . . . 26
4.3 Algorithm . . . 27
4.4 Extraction . . . 28
4.4.1 AST Vector . . . 29
4.4.2 Sanitized Signature . . . 31
4.4.3 Fingerprint and Package Hierarchy . . . 32
4.4.4 Example . . . 33
4.5 Matching . . . 35
4.5.1 Overview . . . 35
4.5.2 Fingerprint Particularity . . . 35
4.5.3 Inclusion . . . 36
4.5.4 Similarity Score . . . 39

5 Design 42
5.1 Components . . . 42
5.2 Extraction with baksmali . . . 44
5.3 Persistence . . . 46
5.3.1 Hyper SQL . . . 46
5.3.2 Active Objects . . . 47
5.4 Learning And Matching . . . 47

6 Evaluation 50
6.1 Overview . . . 50
6.2 Unit Testing . . . 51
6.2.1 Mock Objects . . . 52
6.2.2 Arrange, Act and Assert . . . 53
6.2.3 Code Coverage . . . 53
6.3 Quick Evaluation . . . 54
6.3.1 Design . . . 54
6.3.2 Result Verification . . . 55
6.4 FOSS Evaluation . . . 55
6.4.1 Apps . . . 56
6.4.2 Libraries . . . 56

6.4.3 Matching Configurations . . . 57
6.4.4 Gradle Setup . . . 57

7 Results **60**
7.1 HSQL Embedded vs. Server Mode . . . 60
7.2 Comparing Features . . . 63
7.3 Determining Package Particularity Threshold . . . 66
7.4 Determining Match Confidence Threshold . . . 67
7.5 Comparing Matcher . . . 71

8 Conclusion **77**

Bibliography **81**

List of Figures

2.1 Android Software Stack 6
2.2 Compilation from `.class` to `.dex`[1] 7
2.3 Excerpt of Build Process 8
2.4 Proguard Transformations 10
2.5 Bytecode structure 13

3.1 Example for *AST-coverage* based fingerprint extraction. Reproduced from Potharaju *et.al.*[2] 18

4.1 Venn Diagram of `Package X` in a Library and in Two Shrunken Apps . . . 26
4.2 High Level Description of the ASTLI algorithm; Top: **Learning Phase**; Bottom: **Matching Phase** 28
4.3 Extraction of AST from a Method Body 29
4.4 Conversion of an AST to an AST vector 30
4.5 Package Hierarchy Example 33
4.6 Example for Extraction Steps 34
4.7 Distribution of 120,000 Library Fingerprints 36
4.8 *Greedy* Class Inclusion Check 38
4.9 Example, where Greedy Package Signature Inclusion Check fails 38

5.1 ASTLI Components 43
5.2 Extraction Process Flow 45
5.3 Database Scheme 47

6.1 Test of Unit `A` 52
6.2 Implementation of Unit `A` 52
6.3 Test of Unit `A` with mocked Dependency `B` 53
6.4 Jacoco Code Coverage Report 54
6.5 Example for App Folder Structure 58

7.1 Timing Diagram of both HSQL modes 61
7.2 Runtime in Seconds (left), in Relation (right) 62
7.3 Confusion Matrix based on AST Vectors (left) and Sanitized Signatures (right) 64
7.4 Confusion Matrix with both Features combined 65
7.5 Influence of t_{pp} on Accuracy and Keep Ratio 67
7.6 Comparing ROC-Curves of Different Build Types 70
7.7 Confidence Histograms; From Left to Right: Regular, Shrunken, Obfuscated 70
7.8 Confidence Histograms. Left: Shrunken and Obfuscated; Right: Shrunken, Obfs. and Optimized 71
7.9 Multiclass Metrics; $n \ldots$ amount of matches, $l \ldots$ amount of library packages 73
7.10 Precision (left), Recall (right) 74
7.11 F1 (left), Accuracy (right) 75
7.12 Runtime 75

List of Tables

2.1 Structure of `apk` and `aar` files compared 11

4.1 Example Result 24
4.2 Mapping of Primitive and Object Types to Characters 32

6.1 Evaluation Strategies compared 51
6.2 Build Types Compared 57
6.3 Configurations for ASTLI Matching Algorithm 57

7.1 Example of how ASTLI's Results are Mapped to Binary Classification Results 68
7.2 All Metrics of Hybrid Vs Similarity Matcher 73

Acknowledgements

I thank Johannes Feichtner for his rapid and valuable feedback, his advice on how to approach the problem, and his continuous encouragement. Without him, this thesis would not exist. I also thank Prof. Roderik Bloem for his guide[1] on how to write a master thesis, which helped me to put my work into the right shape. I thank Prof. Keith Andrews for his LaTeX template[2], which i used to write this thesis, and Peter Teufl, for piquing my curiosity in the topic of mobile security. I also thank my colleagues at IAIK, the students from the coworking space, and my friends for their support. My special thanks go to Erika, Alois, Gabriel, and Manfred.

[1] https://www.iaik.tugraz.at/content/about_iaik/people/bloem_roderick/how_to_write_a_thesis.txt, accessed on 2017-02-03

[2] http://ftp.iicm.tugraz.at/pub/keith/thesis/thesis.zip, accessed on 2017-02-03

1 Introduction

The mobile computing landscape changed drastically within the last decade. Smartphones introduced the upheaval and other smart devices such as tablets, wearables, and smart TVs followed. By now, these devices are deeply integrated in our everyday lives: We use them to communicate, socialize, navigate, purchase, play games and more — they are a portal to the digital world. In order to fulfill these duties, smartphones and friends are equipped with a multitude of sensors, computational power, memory, and connectivity. The technology that drives mobile devices becomes increasingly complex and the complexity makes it easy to overlook weak spots. At the same time these devices receive, process and forward a significant amount of personal information about their owner. The combination of increasing complexity and increasing value make mobile devices an appealing target for attackers.

A common attack surface for mobile devices are apps: An attacker can craft a malicious app and trick users into installing it. Bad engineering practices and overlooked flaws in apps can harm users just as well. It is generally hard to tell what exactly an app does and in which jeopardy it puts the user, once it runs on a device. For this reason, security experts inspect the code of apps and try to comprehend its behavior, which is known as *reverse engineering*.

This thesis provides a tool that aids security experts when analyzing and reverse engineering apps. The tool analyzes apps written for Google's Operating System *Android*. Android is a significant platform because it runs on a large share of mobile devices: 300 million Android devices (84% of the global market) were sold in the first quarter of 2016[1]. Furthermore, Android runs on tablets, watches, TVs, and even car media centers[2]. Its versatility and its prevalence make Android a target for malware distributors, who leverage its popularity to reach a large user base. This thesis focuses on threats that come from specific components in Android apps: *third party libraries*.

[1] `https://www.gartner.com/newsroom/id/3323017`, accessed on 2016-07-12
[2] `https://www.android.com/auto/`, accessed on 2017-02-02

The code of an Android app can be divided into code that was written by the app developer (*app code*) and code that was not. The latter comes in the form of libraries (thus *library code*). Libraries from third party suppliers are a cornerstone of the mobile app ecosystem because they fill gaps which cannot be filled by app developers themselves; UI components, networking capabilities, social– and ad network integration are some of the features third party libraries offer. Despite their helpfulness, relying on libraries also entails certain risks: If outdated, the library can leave apps exposed to vulnerabilities. The Apache Cordova library, for instance, suffered from an exposure, which allowed an attacker to alter the apps behavior by sending malicious intents[3]. Another risk are attackers, who inject tampered libraries into apps. The dependency chain attack[4], for instance, tricks a developers build system into pulling malicious libraries. Furthermore, libraries can violate the users privacy by collecting and disclosing personal information. By analyzing 100,000 apps, Grace *et al.*[3] revealed unsettling peculiarities: Certain ad libraries track users, collect and forward their location, their call logs and their browser bookmarks or even execute untrustworthy code from remote servers. These cases should give an example of the risks both developers and users face when third party libraries come into play.

Problem Statement This thesis introduces *ASTLI* (**A**bstract **S**yntax **T**ree based **Li**brary **I**dentification), a tool which analyzes Android applications and identifies third party libraries. Identifying libraries is useful in a variety of situations: In the context of IT security, it is useful to know included libraries because it allows the analyst to infer knowledge of an app from its libraries. It is also convenient to separate app code from library code because it allows the analyst to narrow down the subject of his analysis by excluding the library code and focusing on the app code. Identifying libraries can also be helpful from a legal perspective: Many software companies quarrel with *License Contamination*[5]. This contamination happens when developers integrate Open Source Software (OSS) into commercial products; The OSS license may not approve commercial use, which leads to copyright infringements. A tool that automatically detects OSS libraries could prevent such contamination.

Identifying libraries in apps is challenging because the original library code can be subject to a variety of transformations. One transformation is *code obfuscation*: It removes all debug symbols and identifiers from the code, which prevents identifying libraries based

[3] `https://cordova.apache.org/announcements/2015/05/26/android-402.html`, accessed on 2016-03-03

[4] `http://gary-rowe.com/agilestack/2013/07/03/preventing-dependency-chain-attacks-in-maven/`, accessed on 2015-12-09

[5] `http://www.zdnet.com/article/preventing-open-source-software-contamination/`, accessed on 2017-01-31

on class- , package- or variable names. Another transformation is *shrinking*: This step identifies dead code in libraries and removes it. Shrinking impedes library detection because it can remove a large chunk of evidence. Yet another set of transformations are *code optimizations*, which aim to improve the runtime efficiency of the code. All these transformation alter the library code and make it difficult to trace back the code to its origin.

This thesis examines the transformation techniques of the byte code obfuscator Proguard[6]. Proguard is integrated in the Android SDK, it can be used free of charge, it is easy to activate and developers have good reasons to use it: Obfuscation prevents reverse engineering and shrinking reduces the size of the application. These circumstances motivate the assumption that Proguard's transformations are used frequently.

Approach Our approach is based on ground truth of libraries: ASTLI learns libraries and, when given an application, it can detect the learned libraries. When learning a library, ASTLI extracts certain features from the library. These features are designed in a way that code transformation techniques have little to no effect on them. One feature is based on the Abstract Syntax Tree of a method (thus the tool's name) and the other feature is based on a simplified version of the methods signature. Both features combined form a method's *fingerprint*. All fingerprints of one class represent the class and all classes of a package describe the package. When identifying libraries in apps, these descriptions serve to estimate the similarity between packages. This thesis proposes multiple strategies for measuring the similarity between packages and compares them to each other.

Research Questions Our research centers around the following questions:

- Can we identify libraries in applications?
- How do different code transformation techniques influence the identification?
- How well do our features identify code segments? Are they invariant to transformation techniques?

To answer these questions, we design an evaluation framework, evaluate ASTLI and discuss its results. As data source for the evaluation, we crawl an app repository with free and open source software, download the source code and build the apps with different

[6] `https://www.guardsquare.com/en/proguard`, accessed on 2017-01-31

code transformations. $96 - 97\%$ of our predictions are correct, even if the code has been obfuscated or shrunken. When optimizations are in place, 78% of our predictions are correct.

Outline This thesis is structured as follows: Chapter 2 gives an overview of the Android Platform from a developer's perspective. It explains the design of the Android Runtime and its difference to the Java Runtime. Afterwards it examines the build process of an Android app. The chapter emphasizes the build tools Proguard and Gradle, because they affect many of ASTLI's design decisions. Eventually it describes both the format of Java bytecode and the file format of Android apps and third party libraries.

After having established the basics, Chapter 3 reviews different approaches within the realm of plagiarism detection. It analyzes, which attempts have been made to detect plagiarism in Android market places, how those schemes are related to our scheme and where they differ. It further examines two library detection schemes and explains the differences to our scheme.

Chapter 4 analyzes the requirements for ASTLI. It introduces the features for the code detection and discusses their invariance against code transformation techniques. Based on these features, the chapter then presents the feature extraction- and the library detection algorithm.

With the approach explained, Chapter 5 gives an overview of the design of ASTLI and examines its individual components. It describes how we leveraged the disassembler `baksmali` to extract features from apps and libs, how ASTLI stores these features persistently and how we decomposed the library detection problem into subtasks.

In Chapter 6 we describe the evaluation framework, which consists of the three strategies. The chapter explains how we apply *unit testing* and how we test ASTLI with real apps and libraries. It further states the source for our sample data and gives details on how to run the evaluation.

The outcome of the evaluation can be found in Chapter 7. Each section of this chapter describes a particular problem domain, poses associated research questions and answers them by presenting and discussing the evaluation results.

Chapter 8 proposes ideas and concepts for future work and concludes this thesis.

2 Android Ecosystem

This chapter examines the Android platform and its components. Section 2.1 looks at the origin and the design of the Android operating system in order to establish the domain in which this thesis operates. Section 2.2 gives an overview of both the design of runtime environment and the code that is being used to run apps on Android. Section 2.3 takes a look at the build tools bundled in the development kit and how they interact with each other in the compilation process. We take a closer look at the tool Gradle (Section 2.4) and Proguard (Section 2.5). These tools are of special interest, as it is crucial to understand their functionality in order to motivate design decisions in Chapter 4. Finally, Section 2.6 explains how apps and library archives are structured.

2.1 Platform

Android is an open source operating system employed in mobile devices such as smart phones, tablets, wearables and recently even in cars. The platform is being developed by the Open Handset Alliance with Google as the driving force behind the project. First introduced in 2008, Android managed to take over the smartphone market within a few years reaching a market share of 65% in June 2016[1]. The support the platform enjoys is not limited to hardware: There is a vast supply of apps for Android coming from a range of market places. In June 2016 Google's official market place *Google Play* offered 2.2 million apps [2], followed by the Apple App Store (2 million) and leaving the remaining contenders far behind.

Figure 2.1[3] illustrates the different layers of the Android stack. At the bottom, we find a **Linux Kernel**, which provides basic operating system features like file system access, process management, *etc.* The employed kernel features several adaptions, which serve the purpose to suit the mobile environment. Those adaptions are also referred to

[1] https://www.netmarketshare.com/operating-system-market-share.aspx?qprid=8&qpcustomd=1, accessed on 2016-07-18

[2] http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/; accessed on 2016-07-18

[3] adapted from https://source.android.com/source/index.html

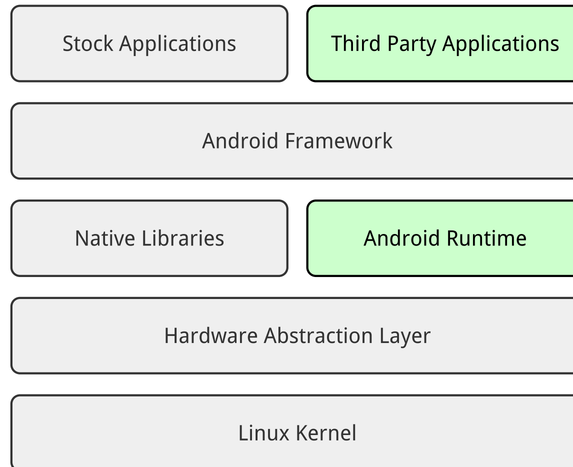

Figure 2.1: Android Software Stack

as *Androidism*[4], and some of them have been merged back into the Linux kernel[4]. The **hardware abstraction layer** provides standardized interfaces to access hardware components such as cameras, Bluetooth connectivity or speakers. On top of that we find a set of **native libraries**, which provide functions like media playback, 2D and 3D graphics and a browser engine. On the same abstraction level we find the **Android Runtime**, which is composed of an implementation of Java's core libraries and a virtual machine that runs a Java bytecode derivative named **Dalvik EXecutable (dex)**. Since this component influences the design of Android as a platform for third party software, Section 2.2 will shed more light on it. The **Application Framework** Layer offers a set of high level building blocks for applications, including but not limited to user interface components, access to resources, locations and others. Finally, we have both **stock apps**, which are being shipped with the device and **third party apps**, which can be installed and uninstalled at the users liking. The system presented in this thesis will analyze the second class of applications and their libraries.

2.2 Runtime

The Java compiler produces *byte code*, which is platform independent. A platform dependent virtual machine then interprets or optimizes the code at runtime. On PCs and servers,

[4] `https://kernelnewbies.org/Linux_3.3#head-b733d694037e0b34ad47e1b5d38ebc4d1bd1d89f`, accessed on 2016-07-18

the **Java Virtual Machine (JVM)** usually performs this task. However, mobile devices lack in computational power an memory and are hence not fit to run a JVM. In order to deal with constraints, such as slow CPUs, little RAM, no swap space and being battery powered, Android deploys its own virtual machine named **Android Runtime (ART)**[4, p. 62].

The following list describes major differences between the ART and the JVM:

- JVM gets served `.class` files, whereas ART requires the entire application code to be compiled into a single `.dex`. Figure 2.2 illustrates how the build tool `dx` merges different code segments. Doing so facilitates the reuse of symbols from one big constant pool, which can cut code size in half.

- ART is a register-based VM, whereas the JVM is stack-based. In register machines, instructions need to reference operands and destination explicitly. In stack machines, operands are being pushed on a stack beforehand and the instruction operates on the top of stack implicitly. A direct consequence is that instructions of register machines are longer and more complicated because of explicit referencing, but the code itself becomes more compact because of increased semantic density and also faster in execution because stack maintenance can be omitted[5]. `dx` takes care of the necessary register allocation.

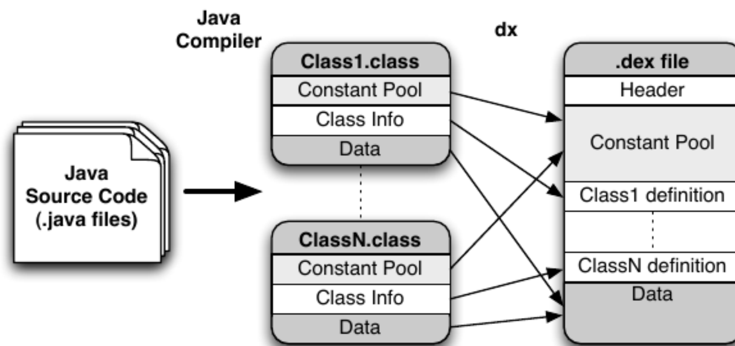

Figure 2.2: Compilation from `.class` to `.dex`[1]

Note that the name *Dalvik Executable* stems from the discontinued VM named *Dalvik*, which was replaced by ART in Android 4.4. Although the internals of the VM were redesigned from scratch, the code format `.dex` has remained unchanged.

[5] `https://sites.google.com/site/io/dalvik-vm-internals/2008-05-29-Presentation-Of-Dalvik-VM-Internals.pdf`, accessed on 2016-07-28

2.3 Build Process

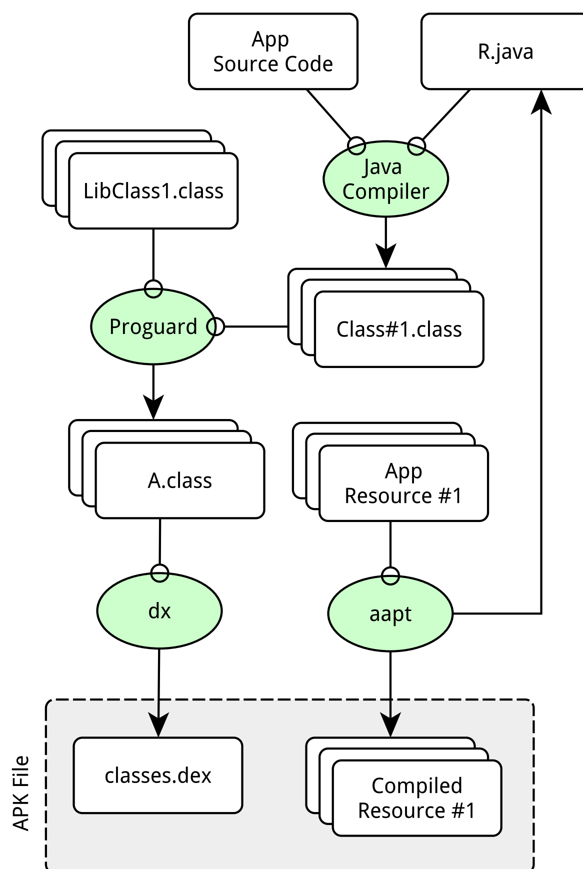

Figure 2.3: Excerpt of Build Process

In order to build software for Android one can use the set of tools that come with the official Android SDK[6]. Figure 2.3 depicts how some of these tools interact with each other while building an app. The build process starts with the compiler `javac`, which compiles Java source code to bytecode. Whereas most of the source is written by hand, a special file that goes by the name `R.java` is being generated automatically by the **Android Asset Packaging Tool (aapt)**. `R.java` provides references to resources over static members, which allows to check statically whether or not a resource has been referenced correctly. In the next step, both the compiled application code and library code run through the obfuscation tool named **Proguard**. The level of obfuscation performed in this step depends

[6]Available through `https://developer.android.com/studio/index.html#downloads`

on how the developer configured the build script. Section 2.5 describes the transformations in more detail. Afterwards **dx** transforms the class files into `.dex` and merges them into a single file named `classes.dex`. More details on what happens in this step can be found in Section 2.6. In the final step `aapt` compiles resource files and packages them with the code into an `apk` archive such that the app is ready for installation or distribution. The entire build process is orchestrated by **Gradle**, which is explained in the following section.

2.4 Gradle

Gradle is a model driven build tool that that uses the language `groovie` to describe build configurations. The tool has been introduced in 2007 with the idea of filling the gap between *Ant* and *Maven*. Ant is configurable but lacks in conventions, which makes build descriptions verbose. Maven on the other hand offers a strong model, but little freedom of deviation within that model[5]. Both tools rely on `XML` for describing the build, which can become hard to read for longer documents due to its verbosity. Gradle provides useful defaults (”convention over configuration”), but also allows to redefine the model, if needed. Gradle also handles dependency management, meaning that one can reference depending libraries from within the build script and Gradle will take care of their availability. The Android SDK comes with Gradle and a plugin for building Android Apps and Android Libraries[7]. This plugin also lets one configure how Proguard should transform the code. The build script itself is named `build.gradle` and found in the root directory of a project or in a first level subfolder (”module”). In Chapter 7 we will see how to adapt these scripts in order to build apks with different obfuscation settings.

2.5 Proguard

Proguard is an obfuscator for Java bytecode, which gets shipped with the Android SDK and is seamlessly integrated into the build process of apps. In order to employ Proguard, a developer only needs to activate the option `minifyEnabled` within projects build script. This option activates a set of default transformations which cover the most common use cases. Figure 2.4 depicts, which transformation Proguard can apply and in which order these transformations will be performed.

The following list describes the transformations in detail:

[7] `https://google.github.io/android-gradle-dsl/current/`

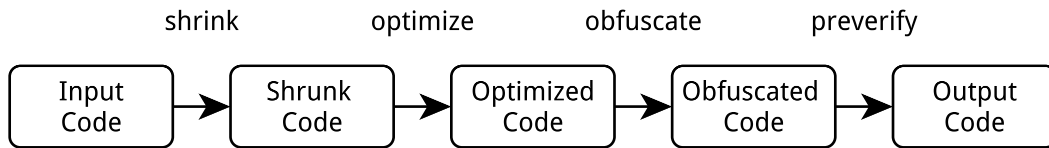

Figure 2.4: Proguard Transformations

Shrinking Hereby, Proguard identifies unreachable code using control flow analysis and strips it from the final code set. The goal of this feature is to reduce the code to the bare minimum, which has several benefits: The app becomes easier to ship and consumes less storage on the users device[8]. The smaller code size also leads to shorter startup times when launching an app, which improves the user experience. After identifying dead code one can also drop unreferenced resources, which reduces the final archive size even further. When `minifyEnabled` is activated, Proguard will shrink the code without any further configurations. Therefor, this transformation is expected to be found frequently in released apps.

Optimization In this step, Proguard performs transformations with the aim to improve efficiency and reduce the runtime, including but not limited to techniques as *inlining*, *constant folding*, *constant value propagation* and a multitude of *peephole optimizations*[9]. This step includes replacing, rearranging, adding and removing code instructions without altering its semantics. Note that in the Java stack, optimizations are usually not performed by the Java compiler, but by the runtime environment at runtime, which also holds for the Android platform. Because some optimizations done by Proguard happen to be incompatible with Androids Runtime[10], these transformation are turned off per default and need to be activated manually.

Obfuscation is a general term for measures that impede code from being reverse engineered. There are a variety of ways to perform this task with a varying degree of sophistication. Proguards employs a rather naive approach and obfuscates by removing debug symbols such names of variables, classes, methods, arguments and packages. The original name is hereby replaced with a shorter, meaningless sequence

[8] see `http://proguard.sourceforge.net/results.html` for examples

[9] see `http://proguard.sourceforge.net/manual/optimizations.html` for a complete list

[10] `https://sites.google.com/a/android.com/tools/recent/proguardimprovements`, accessed on 2016-07-25; `https://stackoverflow.com/questions/35321742/android-proguard-most-aggressive-optimizations`, accessed on 2016-07-25

of characters, which results in package names as `a.a.a.a`. Doing so achieves two goals: It disguises code semantics and reduces the code size. This transformation is included with `minifyEnabled`. Note that in this thesis we use the term *identifier renaming* when referring to this transformation.

Preverification In this step, Proguard adds information that helps the class loader to ensure that the code does not perform any obvious malicious actions. However, the Android Runtime does not utilize this information[11]. For that reason we will neglect this feature when performing our analysis.

2.6 File Format

This section examines the different file formats we will encounter throughout this thesis. It explains the structure of apps and libraries, followed by a review of the bytecode format within those archives. These explanations help to understand where ASTLI operates and motivate design decisions in Chapter 4.

App.apk	Lib.aar
AndroidManifest.xml	AndroidManifest.xml
classes.dex	classes.jar
res/	res/
assets/	assets/
lib/	jni/
resources.arsc	lint.jar
META-INF	proguard.txt
	libs/
	R.txt

Table 2.1: Structure of `apk` and `aar` files compared

[11] http://proguard.sourceforge.net/manual/examples.html#androidactivity, accessed on 2016-07-25

Apps

Android Apps are distributed using the **Android Application Package (apk)** format, which is a `zip` archive following the structure[6] presented in Table 2.1 on the left side. The file `AndroidManifest.xml` contains package name, a range of comptabile API Versions, entry points and other metadata. The actual code of the app is stored in the file named `classes.dex`. All classes from the app and the depending libraries have been prepared for the Runtime and merged into it. The folders `assets/` and `res/` hold resource files for the application. Those folders differ in the fact that `res`-files are managed to a certain degree by the build tools in terms of referencing and localizing them, where `asset`-files are managed by the developer. The file `resources.arsc` contains resources too, but only precompiled strings and binary `XML`. The folder `lib` contains platform dependent native code and `META-INF` holds archive related meta data and optional code signatures.

Libraries

Third party libraries come in two different, yet related file formats named `jar` and `aar`. The former abbreviation stands for **Java Archive** and can be seen as the universal format for distributing code in the Java ecosystem. It contains byte code in the form of `.class` files, which are located in subdirectories determined by the respective package. In conventional Java applications, depending libraries need to be available *on the classpath* during runtime such that the class loader can fetch classes on demand. In the case of Android apps, the library code gets merged with the application code at compile time because there is no option to make the library code separately available on the device. If a library does not rely on any resources, then it can be delivered as a `jar`. Some libraries however do rely on resources such as visual components. The format **Android Archive Library (aar)** was introduced to cover this usecase by bundling library code, resources and meta data in one `zip` archive. `aar` and `apk` files exhibit a similar structure, which can be seen in Table 2.1. The entries above the dashed line are present in both formats and fulfill the same purpose, whereas the entries below the line have format specific tasks. Instead of `classes.dex`, the library holds a `classes.jar` file. Here, the code has not been compiled to `.dex` because it would be much harder to merge it with the app code during compilation that way, which has to happen eventually before the code gets onto a device. Native code is located in the folder named `jni/`. Proguard related settings for a library can be found in the optional file named `proguard.txt`. The folder `libs/` holds dependencies, `R.txt` contains the output

of `aapt` and `lint.jar` has custom lint filters.

Bytecode

Java bytecode is organized in a strict hierarchy of packages and classes. A class bundles state and behavior in form of variables and methods. Related classes are in turn bundled in a package. Figure 2.5 illustrates an example hierarchy: It shows the classes of Package `tld.company.project.pckgA` and their methods. We also notice that packages can have subpackages (e.g. `subPckgB` is a sub package of `pckgA`). The fact that bytecode contains groups of packages and classes helps ASTLI with the detection of similar packages.

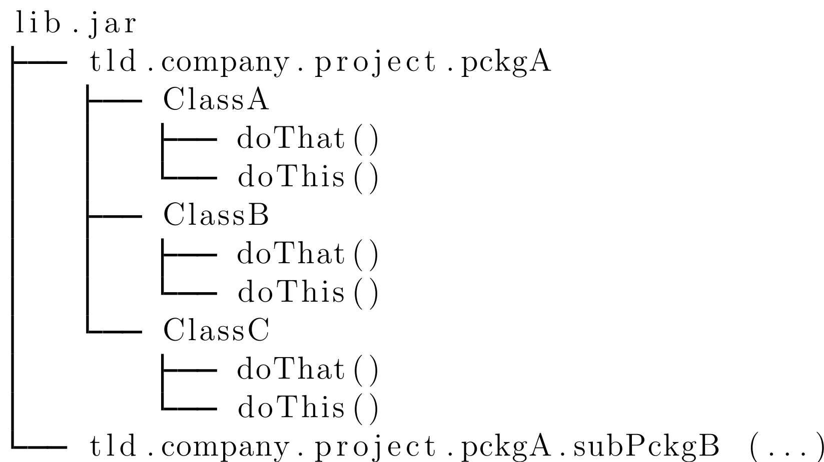

Figure 2.5: Bytecode structure

Note that *Android Application Packages* and *bytecode packages* are different concepts despite sharing the term *package*. In order to keep the terminology unambiguous, we use the term *package* when referring to *bytecode package* in this thesis.

Conclusion

This chapter discussed fundamental components of the Android Ecosystem. It showed how the Android software stack is build and on which layer of the stack the work of this thesis is located. We further examined the design of the Android Runtime and the build process of an Android app. In order to understand design decisions, we investigated what features the obfuscation tool Proguard offers. Eventually we presented the archive format of both apps and libraries and the structure of the java bytecode within.

3 Related Work

This chapter inspects approaches of different fields that are related to the work in this thesis. Section 3.1 covers the field of code based plagiarism detection. It explains which problem plagiarism detection solves and how it is related to library detection. Section 3.2 takes a look at Android library detection approaches.

3.1 Code Based Plagiarism Detection

Detecting plagiarized Android apps is a well investigated field of research because it aims to find *repackaged apps*. This section explains what app repackaging is and points out reasons for being such common attack vector, followed by a in-depth review of three plagiarism detection approaches.

3.1.1 App Repackaging

App repackaging is the act of obtaining a legitimate application package, integrating malicious code and redistributing it over alternative channels. Attackers do this to spy on users, steal their sensitive information (location data, login credentials) or to impersonate them. Another intent for repackaging an app is to add or replace advertisements such that an attacker can harvest ad revenue. The design of the Android software stack facilitates repackaging, whereas platforms such as iOS are less prone. Many Android apps can be reverse engineered with little cost and proficiency. The application format `.dex` can be lead back to bytecode using `d2j-dex2jar` and decompiled with a Java decompiler[1]. Alternatively one can transform the code into the assembly language `smali`, perform modifications and reassemble the app. Another promotive factor lies in the openness of the platform. Android users can install apps from untrusted sources, which is known as *sideloading*. Although sideloading is disabled per default, it is both easy and common to bypass

[1] JD-GUI, `http://jd.benow.ca/` or Procyon, `https://bitbucket.org/mstrobel/procyon`

this measure. There is a wide range of third party market places and APK repositories to choose apps from. Some of those markets offer legitimate apps, while others focus on pirated content. Repackaged apps are common because users are willing to sideload and because there is a vast supply.

Code based plagiarism detection can be used to identify apps that have been repackaged, because both the original app and the copycat share a large amount of code. The same holds for identifying library code: When both library and app share segments of code, it is likely that the app relies on the library. Thus, plagiarism detection offers a range of ideas that are worth exploring for the sake of library detection. The following sections elaborate on different plagiarism detection approaches.

3.1.2 Winnowing

In [7] Aiken *et.al.* present the plagiarism detection algorithm *winnowing.* They further evaluate the service *Measure of Software Similarity* (MOSS), which applies *winnowing* for detecting plagiarism in programming classes. The algorithm slices a text file into overlapping chunks, hashes them and selects a fraction of them as document representatives. The following listing explains each step in detail and applies it to this example document:

```
The quick brown fox jumps over the lazy dog
```

1. Remove noise like whitespace characters and canonicalize the document.

   ```
   thequickbrownfoxjumpsoverthelazydog
   ```

2. Slice the document into continuous substrings of length k (k-*grams*). The example uses $k = 5$.

   ```
   thequ hequi equic quick uickb ickbr ... helaz elazy lazyd azydo zydog
   ```

3. Hash each k-gram using a collision resistant hash function.

   ```
   7e ef b9 28 39 20 16 ... 0c 2f af c8 14 76 11 21
   ```

4. For each w sized group of consecutive hashes (*window*), add the smallest hash value and its offset to a set with unique elements. In the example, w is set to 4.

1	2	3	4	5	6	7	
7e	ef	b9	**28**	39	20	16 ...	add (28, 4)
7e	ef	b9	**28**	39	20	16 ...	dont add (28, 4) again
7e	ef	b9	28	39	**20**	16 ...	add (20, 6)
7e	ef	b9	28	39	20	**16** ...	add (16, 7)
⋮							

5. This set represents the document (*fingerprint*)

   `[(28, 4), (20, 6), (16, 7), ... ]`

In order to see if two documents share a substring longer than the noise threshold k, the algorithm computes both their fingerprint and checks if they share hash values. The approach is based on the following hypothesis: If two fingerprints have the same hash value, the k-grams are likely to coincide as well, because the hash function is collision resistant. Choosing a hash from a local window allows to discard other hashes and to keep the fingerprint small.

Gap Analysis

Winnowing is designed to deal with documents in text format. In order to use *winnowing* for matching libraries, one could adjust the algorithm to work with binary input. However, defining a set of *noise characters* to be removed in Step 1 is not possible for binary files in general, and may also not be possible for `.dex`-files in particular. Alternatively, one could decompile or disassemble APKs with `baksmali` and replace variable and method identifiers with a placeholder. This preprocessing step would allow *winnowing* to operate on the resulting text files. Another benefit of *winnowing* lies in its agnostic attitude towards the language of a document: The entire algorithm makes no assumptions on structure or semantics. This enables detection for documents in any language, whether it be programming or natural languages. Eventually the algorithm was not applied for library detection because taking advantage of bytecode characteristics was expected to yield better recognition rates.

3.1.3 AST Distance

In [2] Potharaju *et.al.* investigate how attackers use social engineering techniques and app repackaging in order to distribute malware across Android market places. Within

their study, they propose three approaches for detecting repackaged apps. One of these approaches goes by the name *AST-Distance* and inspired many design decisions of this thesis. The algorithm relies on `.dex` format for input files and computes app fingerprints based on features extracted from Abstract Syntax Trees (ASTs). The following listing describes how fingerprints are being extracted:

1. Transform `.dex` file into a custom assembly language.
2. Remove all artifacts except for the following:
   - For each method signature keep the number of arguments.
   - For each `invoke-direct` and `invoke-virtual` instruction in the method body, keep the instruction.
   - Replace variable identifiers with the placeholder `local` for local variables or `param` otherwise.
3. Construct an Abstract Syntax Tree of the remaining instructions within the method body.
4. Generate fingerprints out of the ASTs by counting features and populating them in a fixed sized feature vector. This feature vector is composed of *horizontal* and *vertical features*. A *horizontal feature* is the occurrence of two leaf notes with the same parent nodes in the AST, whereas a *vertical feature* is a directed path within the tree of arbitrary length.
5. Compute the fingerprint of an app by summing up over all method fingerprints.

Figure 3.1 illustrates how an example method gets transformed into a fingerprint. In order to tell whether an app is a plagiarism of another app, *AST-Distance* uses the euclidean distance between both app fingerprints and reports a match if the distance is below a certain threshold. The algorithm leans on the hypothesis that two apps are similar if their fingerprints are located within a small neighborhood.

Gap Analyis

ASTLI adopts concepts of *AST-distance* because the latter approach yields a high detection rate (0.5% false positives) and because both problems seemed related enough to reduce one to the other. One of the adopted concepts lies in the transformation from ASTs to feature

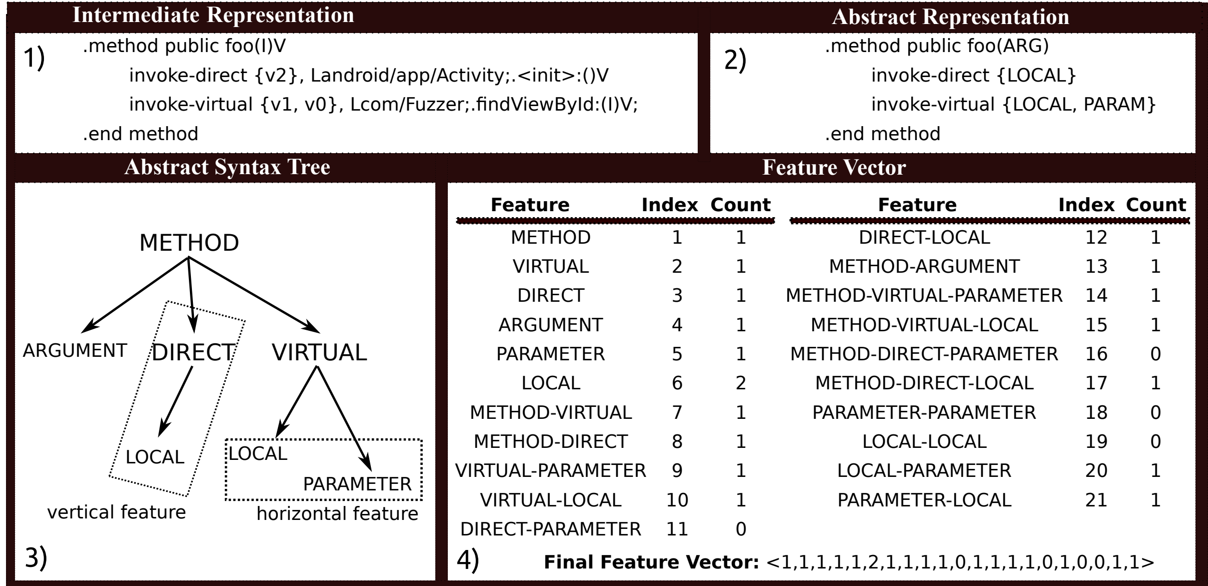

Figure 3.1: Example for *AST-coverage* based fingerprint extraction. Reproduced from Potharaju *et.al.*[2]

vectors, which makes comparing methods cheaper than detecting graph isomorphism or computing the tree editing distance between ASTs[8]. However, *AST-Distance* suffers from the following shortcomings:

- It is questionable if AST based fingerprints actually capture the full extend of an app, since most of the instruction are being removed in Step 2.
- Some dimensions in the *AST-distance* vector depend on each other, which makes them a burden without adding information. One example for dependence are all vertical features that start with `method` and are longer than one node (e.g. `method-virtual`). Adding `method` as the root node in a vertical path is redundant because of `.dex`'s fixed instruction syntax and because there cannot be a root node apart from `method`. Also the features `local-param` and `param-local` depend on each other since the algorithm ignores the order in which they occur. In total, only 12 out of 21 dimensions do not depend directly on each other, which wastes 43% of each vector.
- In Step 5 *AST-distance* creates an app fingerprint by summing over all its method fingerprints. We can adopt this idea for library identification by summing over smaller app components such as packages or classes. However, this idea requires each component to be "complete" when creating the fingerprint, which means that all methods

of a class should be present. Unfortunately, the requirement is not met when considering that libraries may contain dead code and that Proguard removes such code during compile time. *Shrinking* causes large chunks of library code to be removed, which renders the component fingerprints useless because it breaks the fingerprint hypothesis.

3.1.4 Centroid

Chen *et.al* introduce a different approach for app clone detection in [9], which leverages a methods control flow graph for the generation of feature vectors. The authors employ their method to analyze apps and detect plagiarism across third party markets. The following listing describes the fingerprint extraction process in detail:

1. Transform `.dex` to `smali` using `baksmali`.
2. Construct a control flow graph (CFG) for each method. Nodes of the graph represent instruction blocks, edges represent flow dependencies between the blocks.
3. Create a *3D-CFG* by converting each node of the CFG to a vector $\langle x, y, z \rangle \in \mathbb{N}$, where x denotes the block index, y the amount of outgoing edges and z the loop depth of the block.
4. Add two weights w_1 and w_2 to every node with $w_1 = s$ and $w_2 = s + s_i$. s represents the amount of all statements per block and s_i represents the amount of `invoke`-statements per block.
5. Calculate the center of mass (*centroid*) of a *3D-CFG* for both weights.
6. Create a fingerprint by joining the two *centroids*.

Similar to *AST-Distance*, *Centroid* avoids the graph isomorphism problem by conveying the graph into a vector. The algorithm compares methods by computing and normalizing their Chebyshev distance. They further introduce a quasimetric to compute the similarity between two apps based on its centroids. When looking up a centroid in the database, the algorithm does not compare it to all centroids, but to centroids within a certain range after having them sorted upon storage.

Gap Analysis

Centroid can be adopted for library detection. Having features per method is more expensive than app level features, but it suits some library identification related use cases (e.g. *shrinking*) better. We also adopted the idea of sorting feature vectors upon storage and limiting the amount of comparisons to a certain range, which helped us to keep the lookup below quadratic growth. Eventually we dismissed this feature extraction process because we expected better results with the AST based extraction.

Conclusion

This section explained what app repackaging is and which impact it has on users and developers. It also depicted the relation between code based plagiarism detection and library detection and how plagiarism detection can be used to find repackaged apps. Eventually it discussed three approaches that detect plagiarism and analyzed how these approaches can be adapted in order to perform library detection.

3.2 Library Detection

This section will inspect two approaches that deal with third party library detection in Android apps, but have different requirements and approaches to the problem than this thesis.

3.2.1 Common Libraries

In [10] Li *et.al.* analyze apps from different markets in order to populate a list of commonly employed libraries. They publish this list to improve research on application clone detection. Li *et.al.* deduce library popularity by inferring library packages from apps. In the first step, their algorithm analyzes a set of apps and counts how often a package name occurs. After that, the algorithm refines the list by filtering out packages that are unlikely to belong to a common library. An example for such packages are those which occur less than 10 times or packages with obfuscated names. Eventually, the algorithm compares apps with a shared package pairwise. This step establishes how similar the apps themselves and the shared package of both apps are. If both apps and shared packages have similar code segments, the apps are classified as clones, which implies that the package contains app

code. If the packages match, but the apps do not, then the package is classified as a library package.

Gap Analysis

Maintaining a library database by hand can be both time-consuming and prone to incompleteness. The automated inference eliminates these costs and risks, but also takes its toll: Counting the package occurrence relies on the package names being unchanged, which does not hold for apps where identifiers are renamed. The system therefor skips apps that are believed to be obfuscated, whereas the approach presented in this thesis deals with obfuscation on various levels. The approach of Li *et.al.* also relies on heuristics such as package naming conventions. This can lead to false positives when library developer disregard these naming conventions.

3.2.2 LibScout

In [11] Backes *et.al.* describe the tool *LibScout*, which detects third party libraries in obfuscated apps. This tool leverages both the package hierarchy structure and method signatures to build library profiles. The algorithm transforms method signatures into obfuscation invariant *fuzzy descriptors* by removing identifiers and class types. These descriptors are then hashed and fed into a Merkle tree, which represents the package hierarchy.

Gap Analysis

Compared to the approach in Section 3.2.1, *LibScout* is based on ground truth. Libraries need to be collected and labeled manually, which is likely to result in a incomplete data set. However, learning libraries explicitly enables *LibScout* to recognize libraries and their exact version in obfuscated apps. With that ability the authors analyze 5000 popular apps on *Google Play* and find libraries with known exposures. They further investigate how frequent developers update libraries in their apps.

Both the work of Backes *et.al.* and the work done in this thesis exhibit similar premises and requirements. The difference lies in how the problem has been tackled: *LibScout* uses method signatures and package hierarchies for comparison. *ASTLI* uses those features too, but adds features extracted from the code implementation. This improves recognition rates in the case of *shrunken* apps: Whereas *LibScout* cannot handle libraries where more than 40% of the original code has been removed, *ASTLI* can match libraries as long as a

certain amount of code is available. However, *LibScout* is expected to scale better because its approach of comparing packages is less complex than the one used by *ASTLI*.

Conclusion

This section discussed two approaches that share the goal of detecting libraries, but reach that goal within a different scope. The former approach infers libraries from packages, which scales better but excludes obfuscated apps and risks false positives. The latter approach requires libraries to be learned beforehand, which is time consuming and prone to incompleteness but enables detection in obfuscated apps.

4 Approach

In this chapter we introduce our solution to the library detection problem. Section 4.1 lists requirements and limitations of the approach. In Section 4.2 we introduce the features that we use to describe code segments and discuss their invariance against common obfuscation techniques. Section 4.3 gives an overview on the design of the detection algorithm and shows how the different steps relate to each other. Eventually we examine the major steps in the algorithm, which are the extraction step in Section 4.4 and the matching step in Section 4.5.

4.1 Requirements

We design a static analysis tool that is able to detect libraries in application archives. We can split the primary functionality of the tool into two steps: In the **learning phase** the tool learns given libraries. In the **matching phase**, the tool analyzes a given application archive and tries to match its packages with the packages of learned libraries. The following list describes the requirements for the toolkit:

1. If an app includes a library, the tool shall identify both the **name** and the **version** of the library.
2. The tool shall handle application archives, even if they have been **obfuscated**. The tool shall deal with common obfuscation techniques, especially techniques that are employed by Proguard.
3. After analyzing an app, the tool shall report which **package** from the app belongs to which library. Table 4.1 shows an example of how the result from the app analysis looks like. The table lists all app packages, to which library package the app package was mapped, how *confident* the match is (between 0 and 100%) and how much *evidence* was present (`cl` represents the amount of classes in the package, `m` represents the amount of methods in all classes of the package). Row 1 in the table states that

the app package `a.b.pckgA` was mapped to the package `org.lib.pckg1` from the library `libA`. ASTLI is 100% confident that the match is correct and `a.b.pckgA` contained 7 classes and 12 methods, which were used to make the decision.

App Package	Lib Package	Lib & Version	Confidence	Evidence
`a.b.pckgA`	`org.lib.pckg1`	`libA 1.1`	100%	7 `cl`, 12 `m`
`a.b.pckgB`	`org.lib.pckg2`	`libB 1.0`	93%	1 `cl`, 1 `m`
...	...	...	...	...
`a.b.pckgC`	no match			

Table 4.1: Example Result

We also set the following limitations of the tool:

- We focus on apps compiled for the Android platform.
- We will not consider libraries written in **native code** because we cannot employ the proposed feature extraction techniques in native code without further adaptions.
- We can only detect libraries that have been learned before. Other libraries will not be detected by the toolkit.

4.2 Overcoming Obfuscation

One can detect libraries in unobfuscated apps by leveraging names of packages, classes and methods. Obfuscated apps however lack in those names, so we require different features to identify libraries. This section introduces features used for extraction and identification of libraries and discusses their invariance with regard to obfuscation. The analysis focuses on techniques that are employed by the obfuscation tool Proguard (see Section 2.5) and motivates design decisions that have been made to overcome these obstacles.

4.2.1 Features

We rely on features that exhibit the following properties:

1. **Identification** The feature should allow us to identify a code segment. The feature shall change for different sections of code, but remain the same for similar sections of code.

2. **Obfuscation Invariance** The feature should be invariant to transformations performed by an obfuscator.

With theses properties in mind, we base our approach on two features named *AST vectors* and *sanitized signatures*. AST vectors are vectors obtained by extracting structural dependencies of a methods abstract syntax tree. We adopted this feature from the *AST-Distance* approach described in Section 3.1.3. Sanitized signatures result from removing all identifiers from a method signature. These identifiers include the name of the method and the class identifier in all parameter types and in the return type. Section 4.4.2 sheds more light on how ASTLI sanitizes signatures. We combine an AST vector and a sanitized signature to a *fingerprint*. Analogous to the bytecode structure (see Section 2.6), fingerprints are grouped in *package hierarchies*. Section 4.4.3 explains this data structure in detail.

The following sections argue why our chosen features fulfill the stated properties with respect to Proguard's obfuscation techniques.

4.2.2 Identifier Renaming

In this transformation Proguard replaces the original debug symbols with meaningless character sequences. If a developer activates this transformation during compilation of an app, the package names in the application archive will not disclose hints on included libraries. This means that if we want to find libraries in apps with replaced identifiers, we cannot simply rely on comparing package names. AST vectors and sanitized signatures do not contain identifiers, which makes them invariant to identifier renaming.

4.2.3 Shrinking

In this step Proguard removes dead code from the application archive. What code is dead and what is not depends entirely on the app that includes the code. In the learning phase we cannot tell which parts of a library will be removed during app compilation. In preliminary tests we identified cases where more than 90% of the library packages were removed.

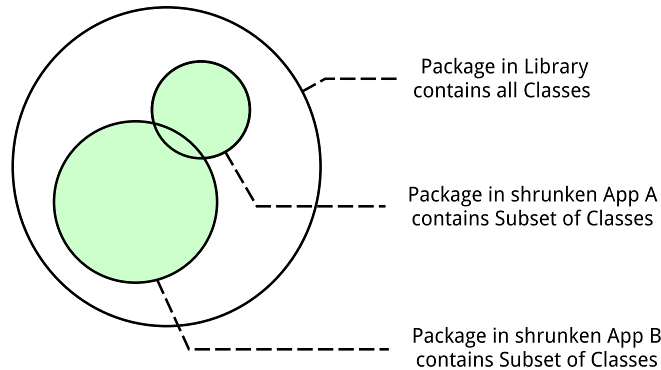

Figure 4.1: Venn Diagram of `Package X` in a Library and in Two Shrunken Apps

Shrinking does not only decide if an entire package gets in- or excluded; it can also remove unused classes in packages and unused methods in classes. Figure 4.1 illustrates this by an example. The big circle represents a `Package X`. When learning the library, all the classes of `Package X` are available. When analyzing `App A`, which has been shrunken at compile time, only a subset of the classes of `Package X` is present. `App B` has also been shrunken at compile time, but since `App B` needs different classes of `Package X` than `App A`, the included subsets differ.

Since all methods in a class and all classes in a package can be subject to dead code elimination, we need to adapt the feature extraction process. *AST-distance* (see Section 3.1.3), the foundation of our approach, computes vectors per app by summing up over its method vectors. When we consider that methods or classes can be removed between learning and matching phase, then the sum of method vectors fails to represent a class, and the sum of class vectors fails to represent a package. This motivates the decision to collect features **per method** in the learning phase and to use them in the matching phase. The downside of storing features per method is that it complicates the matching process both computationally and storage wise, compared to a lower level of detail.

4.2.4 Optimizations

Code optimizations involve rearranging, replacing, adding and removing code segments. Some of the transformations can affect our features in theory, but we can show in our evaluation that those transformations have a minor impact on the detection rates. It is also important to distinguish which feature is affected by the transformation and how

much. A slight change in the AST vector does not necessarily inhibit a correct mapping, since the similarity between AST vectors is based on their distance. However, a sanitized signature that has been altered cannot be led back to the original method, since we check for strict equality when comparing signatures.

Proguard Version 5.3 offers 29 optimizations [1]. Some of these optimizations are disabled by default because they are known to cause trouble on certain versions of Android's Runtime; for that reason we do not expect to encounter them. The following optimizations are known to affect our features:

Inlining Hereby Proguard replaces a method invocation with the body of the invoked method. This usually affects short or unique methods, but also tail recursive methods can be inlined. Inlining alters the AST vector, however if it is a short method that is being inlined, we can argue that the alteration is limited as well.

Code Merging With this transformation Proguard identifies duplicated code fragments and merges them by modifying branch targets. Merging affects the AST vector because it reduces the nodes in the abstract syntax tree.

Method Parameter Removal This transformation causes Proguard to identify unused parameters in methods and remove them from the signature. If a method undergoes this transformation, the sanitized signature will be affected and we will not be able to match it with the corresponding method.

Note that our design focuses on dealing with renamed identifiers and shrunken apps. We are less concerned with handling different optimization techniques because applying them requires considerable testing efforts for app developers; their inconvenience makes optimizations less frequent and therefore less relevant in our opinion.

4.3 Algorithm

This section introduces a high level description of the library detection and discusses how the individual steps of our detection approach interact with each other.

Figure 4.2 describes the steps of ASTLI on a high level. The upper side explains how the learning phase is structured, whereas the lower side depicts the matching phase. The following listing gives a short overview of each step:

[1] `http://proguard.sourceforge.net/manual/optimizations.html` accessed on 2016-10-10

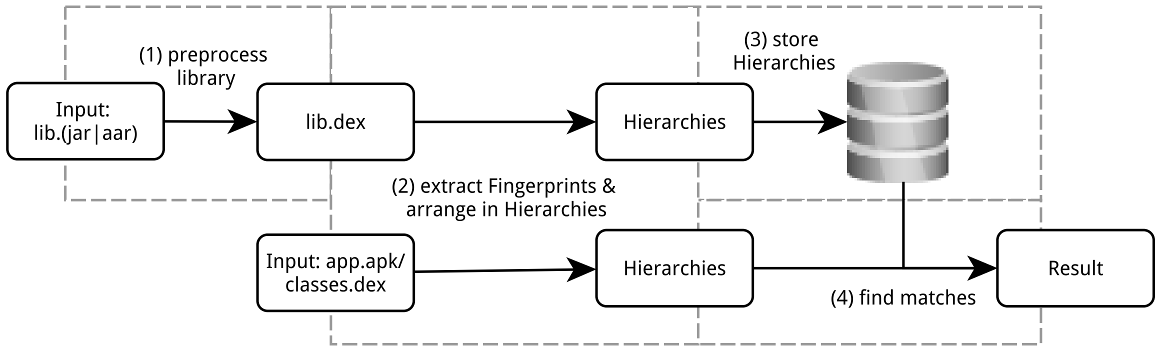

Figure 4.2: High Level Description of the ASTLI algorithm; Top: **Learning Phase**; Bottom: **Matching Phase**

1. When learning a library, ASTLI converts the library into the `.dex` format. This conversion is delegated to the build tool `dx` included in the Android SDK. The conversion to `.dex` allows ASTLI to use the same extraction method for both application archives and libraries.

2. In this step ASTLI generates fingerprints and package hierarchies from a `.dex` file. Both phases share this step. Section 4.4 elaborates on this part of the approach in more detail.

3. After extracting fingerprints and arranging them in package hierarchies, ASTLI stores the results in a database.

4. In the final step of the matching phase ASTLI fetches package hierarchies from the database and compares them with package hierarchies from the application. Section 4.5 explains the details of this step.

4.4 Extraction

This section explains how ASTLI extracts features from `.dex`-files. We start with the extraction of AST vectors in Section 4.4.1 and sanitized signatures in Section 4.4.2. We extract these features from every method in a `.dex`-file and assemble them to fingerprints. Section 4.4.3 explains how and why fingerprints are structured in package hierarchies. We conclude this section with a thorough example, which shows how a fingerprint emerges from a method.

4.4.1 AST Vector

ASTLI generates an AST vector by building an abstract syntax tree and conveying this tree to a vector. AST vectors have the edge over ASTs when we want to compare methods, because they allow us to express the similarity of two methods by computing their distance. Listing 4.3 describes how ASTLI builds the AST out of a method body. The following listing explains each step in detail:

```
Input   : Method Body
Output: Abstract Syntax Tree
1  AST ← createRootNode();
2  foreach instruction ∈ method body do
3  |   keep instructions with opcode in {INVOKE_DIRECT, INVOKE_VIRTUAL};
4  |   instructionNode ← createNode(instruction.opcode);
5  |   foreach parameter ∈ instruction do
6  |   |   parameterNode ← createNode(parameter.type);
7  |   |   instructionNode.addChild(parameterNode);
8  |   end
9  |   AST.addChild(instructionNode);
10 end
11 return AST
```

Listing 4.3: Extraction of AST from a Method Body

- In Line 1 we generate the root node of our tree.
- In Line 2 we iterate over all instructions within the method body
- We consider instructions of type `INVOKE_DIRECT` and `INVOKE_VIRTUAL`[2]; therefor we skip over other instructions in Line 3. We chose these types because they are the most common invocation types according to *Potharaju et.al.*[2]. Alternatively, we could keep track of all invocation types (including `_SUPER`, `_STATIC` and `_INTERFACE`), which would yield a more detailed vector.
- In Line 4 we create a node for the current invocation. Its type depends on the type of the invocation.
- In Line 5-8 we go through the parameters of the current invocation. For each parameter we create a node. The type of the node depends on whether the parameter

[2] Dalvik opcodes explained: https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html; accessed on 2017-02-02

is a *local* variable or a *parameter* of the method (By *parameter* we mean if the parameter passed to the method in the invocation statement is a parameter of the method we are currently conveying to an AST; For clarification, consult the example in Section 4.4.4). We add this node as a child of the invocation node.

- In Line 9 we add the instruction node as a child of the tree root.

ASTLI converts the tree into a vector by counting both *horizontal* and *vertical features*. We borrow the definition from *AST-Distance* in Section 3.1.3: A *horizontal feature* is a pair of leaf nodes with the same parent node, whereas a *vertical feature* is a directed path of arbitrary length, which starts at the root node of the tree. Each dimension in our AST vector resembles the amount of occurrences of a particular horizontal or vertical feature. The following enumeration explain how ASTLI counts the occurrences in Listing 4.4:

```
   Input  : Abstract Syntax Tree
   Output: Abstract Syntax Tree Vector
 1 vector = createVector();
 2 //count horizontal features;
 3 foreach invokactionNode ∈ AST.getChildren() do
 4 |   #locals ← |{c ∈ invokactionNode.getChildren() | c.type = local}|;
 5 |   #params ← |{c ∈ invokactionNode.getChildren() | c.type = param}|;
 6 |   vector[local_local] ← (#locals choose 2);
 7 |   vector[param_param] ← (#params choose 2);
 8 end
 9 //count vertical features;
10 foreach lvl1Node ∈ AST.getChildren() do
11 |   increment(vector[lvl1Node]);
12 |   foreach lvl2Node ∈ lvl1Node.getChildren() do
13 |   |   increment(vector[lvl2Node]);
14 |   |   increment(vector[lvl1Node, lvl2Node]);
15 |   end
16 end
17 return vector
```

Listing 4.4: Conversion of an AST to an AST vector

- Line 1 initializes our vector.

- Lines 3 to 8 count the horizontal features by going through all first level nodes of the AST and determining the amount of leaf pairs for each node. Line 4 and Line 5

count the amount of *local variables* and *parameters* of the current invocation node. Line 6 and 7 compute the amount of pairs of type `local-local` and `param-param`. Determining the amount of pairs is equivalent to the *handshake problem*[3], so we can compute it using the binomial coefficient over 2.

- Line 10 to 16 count the vertical features by iterating over all first level nodes of the AST again. Line 11 increments the occurrence count of the current node by 1. In Line 12 we iterate over the children of the current node and increment the occurrences of both paths, be it either level 2 only or a conjunction of level 1 and level 2.

Note that our conversion neglects some dimensions from the original scheme in [2] for the following reasons:

method This vertical feature is always 1 for each AST. Keeping track of it makes sense in conjunction with grouping vectors by summing them up. Since we do not group vectors in this way, we removed this dimension.

method-* Here we have a dependency to each path that is composed of the same nodes but does not start with `method` (e.g. `method-direct-local` and `direct-local`). Since `method` is the only possible root node for our AST, each path starts with a `method` node. Knowing that our path started with `method` does not add information, therefore it can be omitted.

argument was removed because the amount of arguments is included in the sanitized signature of the method.

local-param was removed because it depends on the amount of `locals` and `params`. Since we keep track of both, we can omit mixed pairs.

4.4.2 Sanitized Signature

Sanitized signatures contain information from a method signature that are invariant to obfuscation techniques. In order to sanitize the signature from features prone to obfuscation, ASTLI removes method identifiers, parameter names and modifiers from the original signature. It further replaces parameter types and the return type with a single letter code. Table 4.2 illustrates how ASTLI maps different types to characters. For primitive types,

[3] `http://mathworld.wolfram.com/HandshakeProblem.html`, accessed on 2016-10-23

ASTLI adopts the mapping from `smali`[4]. Since object types can be subject to identifier renaming, they are mapped to a obfuscation invariant token according to the following rules:

Primitive				Object	
boolean	Z	int	I	current class	T
byte	B	long	J	class in same package	O
short	S	float	F	external class	E
character	C	double	D		

Table 4.2: Mapping of Primitive and Object Types to Characters

- If the type matches with the type of the class we are currently processing, we assign the letter `T`.
- If the type belongs to the same package as the current class, we assign the letter `O`.
- Otherwise we assign the letter `E` for external.

4.4.3 Fingerprint and Package Hierarchy

After having extracted both AST vector and sanitized signature, ASTLI combines both features to a fingerprint. These fingerprints can already be used for the matching process, but just by themselves they do not provide enough information to map packages unambiguously. An example for ambiguous methods are *getters* and *setters*: They have a similar structure and thus a similar fingerprint, but they do not necessarily belong to the same class.

We can overcome ambiguity by capturing the entire structure of a package. Two unrelated methods might share the same fingerprint, but two unrelated classes are unlikely to share all of their fingerprints. The same logic holds for packages: We can conclude that two packages are related if their classes share the same fingerprints. To draw this conclusion we require a data structure that groups fingerprints in classes and classes in packages. We refer to this data structure as package hierarchy. Figure 4.5 illustrates an example of a package hierarchy: `Package X` is composed of classes and each class is a group of fingerprints.

[4] `https://github.com/JesusFreke/smali/wiki/TypesMethodsAndFields`, accessed on 2016-10-26

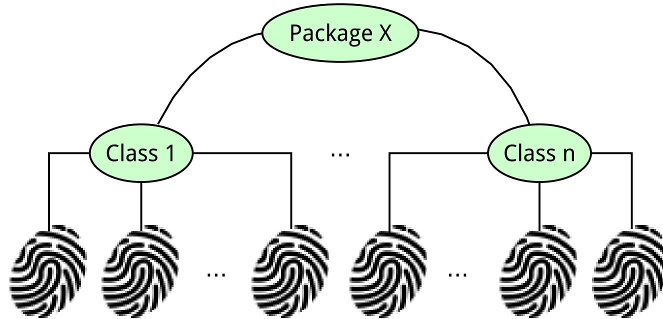

Figure 4.5: Package Hierarchy Example

4.4.4 Example

Figure 4.6 contains a complete example of all extraction steps. We convert the method `doSomething` from the class `ClassX` located in the package `hello`. Note that we use `smali` to represent the method and its implementation. The original Java source code equivalent should help to gain a better understanding of the `.dex` format, but is not involved in the extraction process. The following listing explains what happens in each step:

1. In this step we create an AST out of the methods implementation. We add one root node `method` and process the instruction `invoke-direct {v0, v1, p1}`. For this instruction we create the node `inv-direct`. The parameters `v0, v1` of the `invoke-direct` statement are local parameters. `v0` is an instance of `ClassY`, which is always the first parameter of a instance method invocation, whereas `v1` is an `Integer` that contains the value of `ClassX.field1`. `p1` is the parameter `number`. For `v0, v1` and `p1` we generate two `local` nodes and one `param` node and add them to the `inv-direct` node as children.

2. In this step we convert the tree to an AST vector by counting both vertical and horizontal features. The vertical features are paths consisting of 1 or 2 nodes. We count the following paths of 1 length: `DRC`:1, because of the `inv-direct` node; `LOC`:2, `PAR`:1 because of the respective child nodes; `VRT`:0 since there is no `invoke-virtual` node in the tree. Paths of length 2 are: `DRC-LOC`:2 and `DRC-PAR`:1. Both `INV-LOC` and `INV-PAR` remain 0. We finally generate the horizontal features by counting the pairs of `local` and `param` nodes. There is 1 `local` pair, thus `LOC-LOC`:1 and no `param` pair.

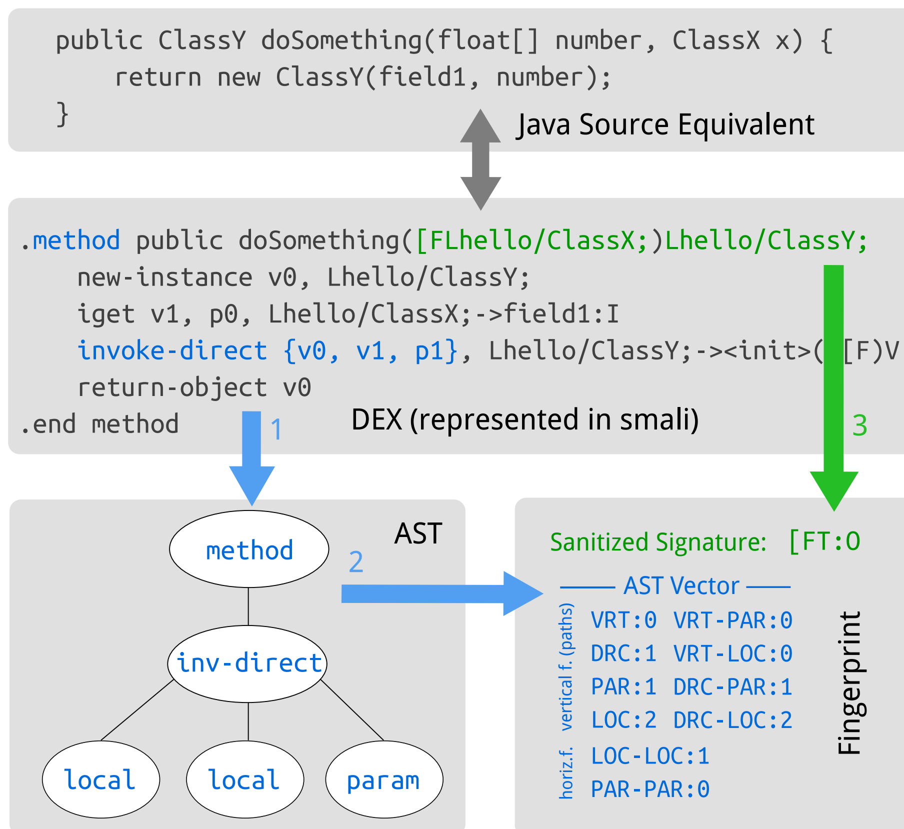

Figure 4.6: Example for Extraction Steps

3. In this step we generate the sanitized signature. We are interested in the parameters `[F`, which represents the parameter `float[] number`, and `hello/ClassX`[5], followed by the return type `hello/ClassY`. Parameters and return type are colored green in the `.dex`-box of the figure. We leave `[F` as is and replace the parameter `hello/ClassX` with the character `T` because the type matches the class we are currently processing. We add a colon `:` to divide parameter types from return type and replace the return type `hello/ClassY` with the character `O` because the type is located in the `hello` package. This generates the sanitized signature `[FT:O`.

The fingerprint on the bottom right box of the figure is composed of sanitized signature and AST vector.

[5] In this description we omitted the the prefix 'L' and suffix ';' of `smali` object types to improve readability.

4.5 Matching

This section explains how ASTLI matches packages from an application archive to packages from learned libraries. Section 4.5.1 gives an overview of the entire matching process. Section 4.5.2 introduces the concept of fingerprint particularity and explains how it helps ASTLI to find better matching candidates. Section 4.5.3 defines the inclusion relation $\subseteq$ between packages and Section 4.5.4 explains how ASTLI measures similarity between two package hierarchies.

4.5.1 Overview

In the matching process, we are given a set of package hierarchies P_a which we extracted from an application archive. For each package hierarchy $p_a \in P_a$ we do the following:

1. We sort all fingerprints in p_a by **particularity** in descending order such that we can choose a set of particular fingerprints. Section 4.5.2 explains what a particular fingerprint is. We refer to particular fingerprint as *needles*, which should portray the metaphor of finding a needle in a haystack.

2. For each needle, we query the database (*haystack*) for fingerprints that have the exact same AST vector and sanitized signature. We collect the package hierarchies of similar fingerprints and store them in the candidate set P_l.

3. For each candidate $p_l \in P_l$, we check if $p_a \subseteq p_l$, which means that the application package is **included** in the library package. Section 4.5.3 defines this relation and explains how to compute it.

4. If $p_a \subseteq p_l$, we compute the **similarity** $s(p_a, p_l)$, which depends on the AST vectors in p_a and p_l. Otherwise, we set $s(p_a, p_l)$ to 0. Section 4.5.4 elaborates on the definition and computation of the similarity score.

5. We sort package candidates by similarity score in descending order and yield the package with the highest score that meets a minimum threshold as a match.

4.5.2 Fingerprint Particularity

Some fingerprints are more likely to match with unrelated fingerprints than others. When we populate a set of candidates, we are better of with rare, particular fingerprints than

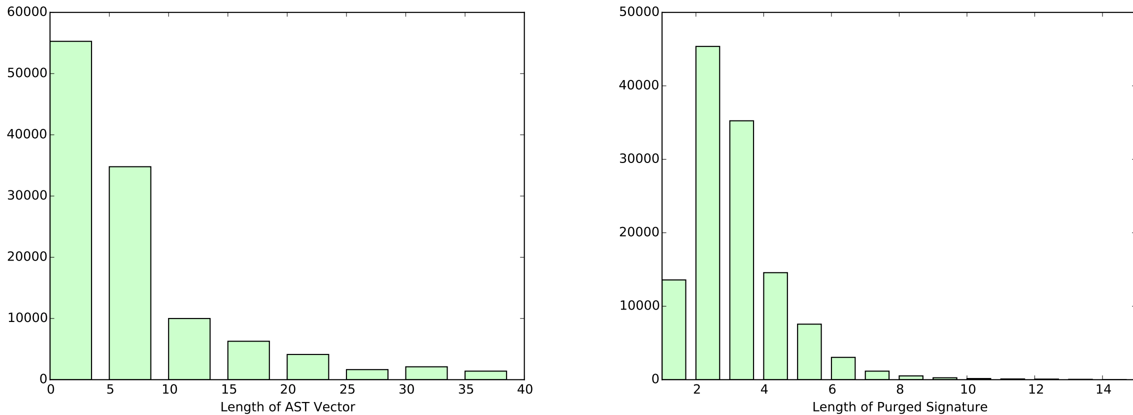

Figure 4.7: Distribution of 120,000 Library Fingerprints

with frequent ones, because a rare fingerprint will yield less *false positive* candidates than a frequent fingerprint. We can estimate particularity by observing how fingerprints are distributed. Both histograms in Figure 4.7 depict the distribution of 120,000 fingerprints: The left histogram shows the distribution over the length of AST vectors, whereas the right histogram shows the distribution over the length of sanitized signatures. From these distributions we observe that a fingerprint with a long signature or a long AST vector is more particular, because it occurs less frequent. This helps us when we want to find appropriate candidates for a package hierarchy. If the fingerprint that we look up is rare, then we can confine the set of candidates and thus save time.

We approximate the particularity of a fingerprint with the *particularity score*. Let $m = (s, v)$ be a method fingerprint with a sanitized signature s and an AST vector v. Then the particularity score of m is defined as

$$\text{score}(m) := w_s \cdot \text{length}(s) + w_v \cdot \|v\|_1, \tag{4.1}$$

whereas $\text{length}(s)$ returns the amount of character of s and $\|v\|_1$ denotes the Manhattan distance of v. We weight both dimensions with w_s and w_v in order to rectify the distributions.

4.5.3 Inclusion

The inclusion relation $\subseteq$ expresses if a package hierarchy p is included in a package hierarchy p'. Inclusion depends on the sanitized signatures in both package hierarchies. We use

inclusion instead of equivalence in order to handle the loss of code when an obfuscator removes dead code from an app. Therefor, inclusion is reflexive and transitive, but not symmetric. We define inclusion as follows:

$$p \subseteq p' \Leftrightarrow \exists f_c : p \mapsto p',\ f_c\text{... injective.} \tag{4.2}$$

In other words: Package p is included in Package p' if and only if there exists an injective mapping f_c for all classes in p to the classes in p'. We require injectivity because we expect each library class to end up as one application class at most.

We add further requirements for our class mapping f_c: Let $c \in p$ be a class in the package p and $c' \in p'$. Then

$$f_c(c) = c' \Rightarrow c \subseteq c'. \tag{4.3}$$

If we map an application class c to a library class c', then the application class is included in the library class. The inclusion relation between classes is defined analogous to the inclusion relation between packages:

$$c \subseteq c' \Leftrightarrow \exists f_m : c \mapsto c',\ f_m\text{... injective.} \tag{4.4}$$

However, we can only map a fingerprint $m \in c$ to a fingerprint $m' \in c'$ if their sanitized signatures are equal, or:

$$f_m(m) = m' \Rightarrow \text{Signature of } m \text{ and } m' \text{ are equal.} \tag{4.5}$$

Now that we defined inclusion for both packages and classes, we can describe how we determine inclusion.

Determine Class Inclusion

Let $c = \{s_1, ..., s_n\}$ be a class consisting of sanitized signatures s_i (for the sake of simplicity, we ignore AST vectors and fingerprints for now). Then we can determine if $c \subseteq c'$ in a *greedy* manner as described in Listing 4.8. The idea behind this approach is to find all signatures from c in c'. If we find one, we delete it from the c' set such that we do not pick the given signature in c' twice. If we found all signatures from c in c', we know that there is an injective mapping f_m and thus $c \subseteq c'$.

Input : Class c, Class c'
Output: True if $c \subseteq c'$

```
1 foreach s_i ∈ c do
2   if s_i ∈ c' then
3     remove s_i from c'
4   else
5     return False;
6   end
7 end
8 return True;
```

Listing 4.8: *Greedy* Class Inclusion Check

Determine Package Inclusion

In order to determine if package p is included in package p', we need to find an injective mapping f_c for all classes in p to classes in p'. This task is less straightforward than the mapping f_m for methods because of the lacking symmetry in the class inclusion relation. Figure 4.9 illustrates an example, where the greedy approach from Listing 4.8 fails.

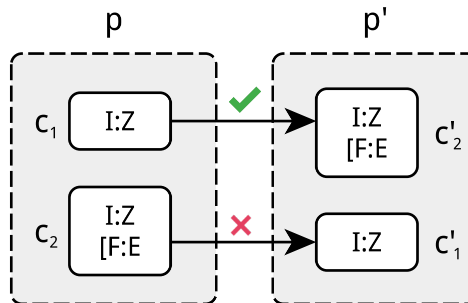

Figure 4.9: Example, where Greedy Package Signature Inclusion Check fails

We are given the packages p and p' and we can tell that $p \subseteq p'$, because there exists an injective mapping f_c with $f_c(c_1) = c'_1$ and $f_c(c_2) = c'_2$. However, the greedy approach fails because class c_1 can also be assigned to c'_2, since $c_1 \subseteq c'_2$ holds. If we assign c_1 to c'_2, we end up with c_2 being unassigned, because $c_2 \not\subseteq c'_1$.

In the case where a valid assignment is not possible, we could backtrack by amending some assignments until we explore all possibilities. However, we opted to reduce the problem such that we can solve it with the **Hungarian Algorithm**[12]. Given a set of

workers, a set of tasks and a cost matrix, this algorithm assigns workers to tasks such that the overall costs are minimized. Instead of workers and tasks, we use classes of p and p'. We construct our cost matrix M_s as follows:

$$M_s \in \{0,1\}^{|p| \times |p'|},\ M_s[i,j] = \begin{cases} 0 & \text{if } c_i \subseteq c'_j \\ 1 & \text{otherwise} \end{cases} \tag{4.6}$$

If we apply the Hungarian Algorithm on M_s, we end up with an assignment f_c. Since the algorithm minimizes the cost of f_c, it prefers assignments that cost 0 over the ones that cost 1. Eventually, we compute the overall cost of f_c with

$$\text{cost}(M_s, f_c) := \sum_{i=1}^{|p|} M_s[i, f_c(i)] \tag{4.7}$$

and can argue that if

$$\text{cost}(M_s, f_c) = 0 \Leftrightarrow \exists f_c : p \mapsto p',\ f_c \text{... injective.} \Leftrightarrow p \subseteq p' \tag{4.8}$$

4.5.4 Similarity Score

The similarity score helps us to determine how similar two packages hierarchies p and p' are. The score depends on the similarity of the AST vectors in the respective packages. Packages that are similar yield a higher score than packages that are not. This section explains how we compute similarity between packages, classes and AST vectors.

Package Similarity

We compute the similarity score $\text{s}(p, p')$ by leveraging the Hungarian Algorithm, because it helps us to find the mapping between classes with maximum similarity. We fill the cost matrix S with the similarity score of the respective classes $s(c, c')$, which is explained in the following section.

$$S \in \mathbb{R}^{|p| \times |p'|},\ S[i,j] = \begin{cases} s(c, c') & \text{if } c_i \subseteq c'_j \\ 0 & \text{otherwise} \end{cases} \tag{4.9}$$

Since the Hungarian Algorithm minimizes the costs in the cost matrix, we apply it on an inverted matrix S_{inverted}, where we negate each entry and shift it by the maximum:

$$S_{\text{inverted}} = (\max(S) - S[i,j])_{ij} \tag{4.10}$$

After the Hungarian Algorithm generated a mapping f_c, we can compute the similarity with $\text{cost}(S, f_c)$ as defined in Definition 4.7.

Class Similarity

Let $c = \{m_1, ..., m_n\}$ be a class consisting of a list of fingerprints m_i where each fingerprint consist of a sanitized signature s_i and an AST vector v_i. Then the class similarity $s(c, c')$ can be determined with the Hungarian Algorithm once again. First, we generate the cost matrix T:

$$T \in \mathbb{R}^{|c| \times |c'|},\ T[i,j] = \begin{cases} s(v_i, v'_j) & \text{if } s_i = s'_j \\ 0 & \text{otherwise} \end{cases} \tag{4.11}$$

We invert T as in Equation 4.10 and let the Hungarian Algorithm find the best assignment. We use the cost function $\text{cost}(T, f_c)$ in Definition 4.7 to determine the similarity score $s(c, c')$. The next section explains how we measure similarity between the two AST vectors v_i and v'_j.

AST vector Similarity

We express the similarity between two AST vectors v and v' with the following formula:

$$s(v, v') = \max(0, \|v\|_1 - \|v - v'\|_1) \tag{4.12}$$

We use the Manhattan distance in order to determine distance and length of vectors because we adopted this metric from [8], who introduced the feature extraction process. This formula fulfills the following requirements:

- We want the similarity to be 0 if the vectors are too far apart. The threshold where similarity becomes 0 is reached if the difference between to vectors is greater than the vector itself.
- We do not accept negative values for similarity, because we want to avoid the situation where a mismatch of vectors worsens the overall score of an assignment. We ensure this requirement by taking the maximum between the difference and 0.

- We require maximum similarity when both vectors are equal. In that case $\|v - v'\|_1$ becomes 0, such that $s(v, v') = \|v\|_1$.
- If $\|v_1\|_1 > \|v_2\|_1$, we require $s(v_1, v_1) > s(v_2, v_2)$ because we want larger and therefor more particular vectors to have more influence on the assignment cost.

Conclusion

This chapter explained how we designed the solution of the library detection problem. First we introduced and motivated both a set of requirements and limitations. We explained which features we extract and discussed their theoretical performance with regards to different obfuscation techniques. After that we introduced the algorithm and its different phases. We went into detail on the learning phase, where the extraction of feature takes place, and on the matching phase, where the same features are used to map code segments.

5 Design

This chapter explains the design of ASTLI. It first gives a general overview in Section 5.1 by explaining how responsibilities are separated into components and how components interact with each other. The successive sections examine these components in more detail. Section 5.2 describes the feature extraction process and its tie to the `.dex` disassembler `baksmali`. Section 5.3 explains how learned libraries are stored to and retrieved from disk. We conclude this chapter with an in dept review of how learning and matching was implemented.

5.1 Components

In the design process we identified responsibilities and assigned them to components. Each component has its distinct responsibility which requires it to operate on a certain level of abstraction. In order to prevent mixing different abstraction levels, we partitioned the tool and reduced dependencies between components to the minimum. Figure 5.1 depicts the components in the form of a package diagram. The following listing describes each component *bottom up* by starting from the component with the fewest outgoing dependencies.

baksmali/* contains a fork of the `.dex` file disassembler `baksmali`. We leverage the code of this project and adapt its program flow. The altered version parses a `.dex` file and returns handles that enable the inspection of dexed classes. We use these handles for the feature extraction. Note that this component is actually composed of multiple packages that share the prefix `org.jf.baksmali`, but has been portrayed as one single component in Figure 5.1 for the sake of clarity.

extraction handles the feature extraction process. It delegates `.dex` file parsing to `baksmali` and iterates over classes and methods to extract ASTs, fingerprints and package hierarchies. Section 5.2 describes this component in more detail.

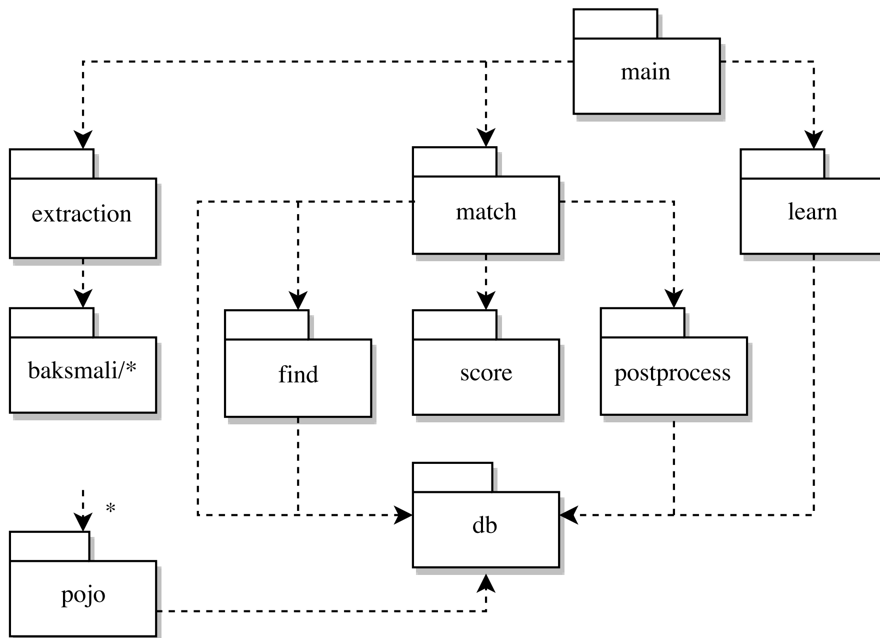

Figure 5.1: ASTLI Components

db deals with persistent storage of package hierarchies. It offers an API to store and retrieve hierarchies. This component relies on the object relational mapping framework `Active Objects` and on the in-memory database `HyperSQL`. Section 5.3 explains which classes enable persistent storage and how the database scheme is designed.

pojo contains data structures that represent fingerprints, package hierarchies and vectors. The name is an acronym for *Plain Old Java Object*, which hints the lack of dependencies to external libraries. All components in ASTLI[1] rely on `pojo` because they use `pojo` classes to communicate with each other.

learn receives extracted package hierarchies and stores them to the database.

match takes extracted package hierarchies and matches them with package hierarchies from the database. This component handles the matching process and delegates individual sub tasks to respective components `find`, `score` and `postprocess`. Section 5.4 goes into more details on how the responsibilities are divided.

[1] except `baksmali`

find Given a package hierarchy, this component finds package hierarchies from the persistent storage and offers them as candidates to the matching algorithm.

score determines the similarity score between two package hierarchies.

postprocess contains classes that handle results of the matching algorithm. Post processing task involve verifying the classification results, extracting scores for further analysis and formatting results for review.

main serves as entry point for the application. It handles the command line arguments and bootstraps the algorithms. `main` extracts package hierarchies from a given input file via the `extraction` component and passes them on to either the matching or the learning algorithm.

5.2 Extraction with baksmali

Figure 5.2 breaks the feature extraction process down into single steps. It shows, which transformations the input file undergoes and in which class these transformations occur. The white rectangles with rounded edges indicate the input/output format of a step, whereas the wrapping boxes represent classes, components and external libraries. The following listing explains each step:

Feature Extractor acts as the entry point for external components through its public API and manages the entire extraction process. It receives a file of type jar or `.dex` and returns a list of package hierarchies, which serve as input for both the learning and matching algorithm. The class delegates individual subtasks to the following classes or components and passes on their intermediary results.

dx is part of the Android SDK[2] and bundles all `class`-files in a `jar`-file to a single `.dex` file. This preprocessing step is exclusive to the library learning phase because Android Apps are already dexed. By converting libraries to `.dex`, ASTLI can apply the same feature extraction procedures in both learning and matching phase.

baksmali parses the `.dex` file and returns a list of *class definitions*, which represent classes of the `.dex` file. The feature extractor passes the class definitions on to the package hierarchy generator, which orchestrates the actual extraction.

[2] located in `<SDK>/build-tools/<VERSION>/lib/dx.jar`

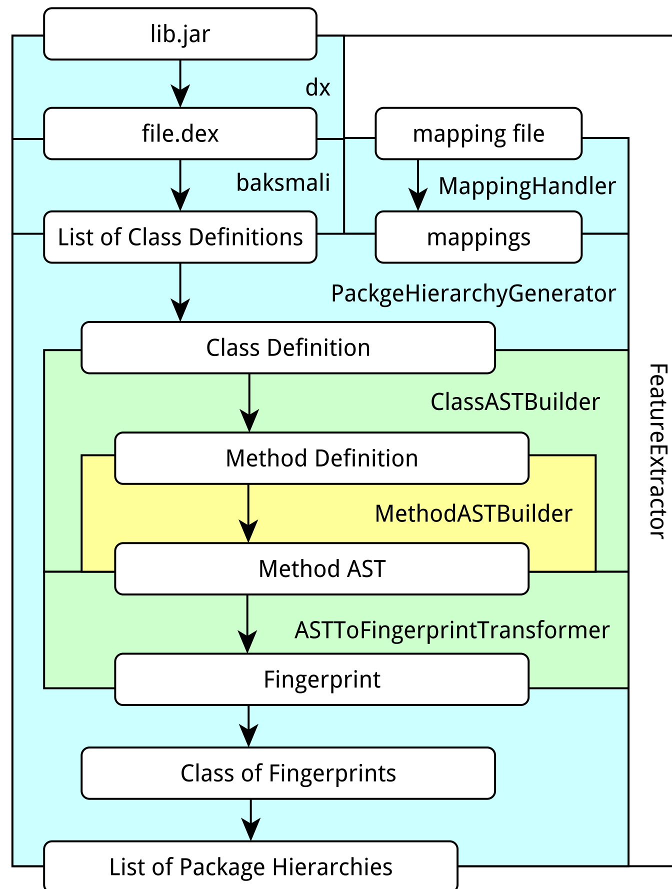

Figure 5.2: Extraction Process Flow

Mapping Handler Before performing the extraction, the feature extractor lets the mapping handler parse a so called *mapping file*. This mapping file is a byproduct of compiling Android apps with Proguard and maps obfuscated class, package and method names to their original name. The mapping is useful when evaluating the matching algorithm, because it enables verification of matches by comparing package names. This step is optional and depends on whether or not a mapping file is available.

Package Hierarchy Generator iterates though class definitions and delegates extraction of ASTs and Fingerprints to dedicated classes. If the generator receives a mapping, it will use the mapping to translate names of classes, packages and types. Furthermore, the package hierarchy generator will assemble fingerprints into corresponding package hierarchies. The package hierarchy generator does not receive bare class definitions

but class definitions wrapped in Class AST Builders.

Class / Method AST Builders handle the transformation of code instructions into ASTs. The class AST builder iterates over all methods of a class definition, whereas the method AST Builder creates ASTs (Listing 4.3) and sanitized signatures (Section 4.4.2).

AST To Fingerprint Transformer counts horizontal and vertical features in an AST and assembles those feature into an AST vector as described in Listing 4.4.

5.3 Persistence

ASTLI offers to learn features of libraries and to match those features in apps. The two phases do not necessarily run in direct succession. Therefor the features from the learning phase cannot be kept in memory for the matching phase. Even if learning and matching run consecutively, learning the same libraries repeatedly from scratch is much more time-consuming than learning libraries once and storing the extracted features persistently. In order access learned features during the matching phase, ASTLI stores results in a database. The database is managed by Hyper SQL. Furthermore, ASTLI relies on the Object Relational mapping framework Active Objects which simplifies database related input and output operations.

5.3.1 Hyper SQL

Hyper SQL[3] is a lightweight relational database management system, which can operate in multiple modes. One mode is the server mode, in which Hyper SQL runs in an independent process. Another mode is the embedded mode, where Hyper SQL runs in the same process as the application that connects to it.

During the development process we used Hyper SQL in embedded mode for the sake of simplicity. In production, a user might chose to use Hyper SQL in server mode. If a user analyzes multiple apps in a row, the server mode prevents needles startup times in between runs, which are caused by loading the database into memory.

[3] `http://hsqldb.org/`

5.3.2 Active Objects

Active Objects[4] is a framework that maps objects to entities in a database scheme. Instead of converting objects to entities and vice versa, the developer creates and annotates interfaces. Active Objects then generates the corresponding database scheme and handles the conversions.

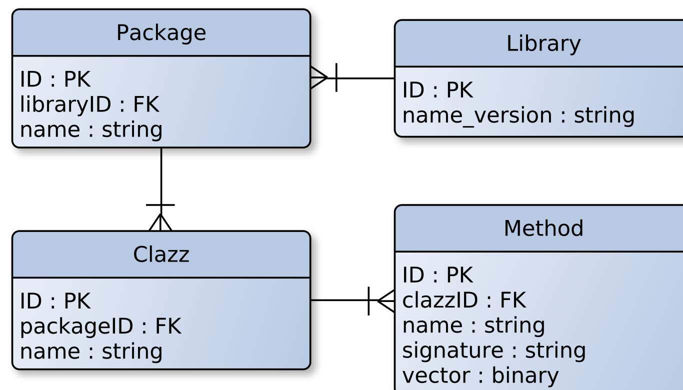

Figure 5.3: Database Scheme

Figure 5.3 depicts the database scheme used for storing and retrieving components. The scheme contains four entities: *libraries*, *packages*, *clazzes*[5] and *methods*. The entire scheme follows a tree like structure. A library can have many packages, but a package only belongs to one library. The same 1 to n relation holds for the other entities and their parents. ASTLI stores extracted features in the method table under the columns *signature* and *vector*.

5.4 Learning And Matching

This section discusses the design of `learn` and `match` components and their sub components `find`, `score` and `postprocess`. Learning and matching are on opposing ends of complexity: While the `learn` component merely forwards extracted features to a database, `match` has to deal with a variety of decisions and parameters that influence its results, including questions as:

- Which packages make suitable candidates for package comparison?

[4] https://developer.atlassian.com/docs/atlassian-platform-common-components/active-objects

[5] We use the identifier *clazz* to refer to classes in order to prevent naming collisions with `java.lang.Class`

- How should packages be compared and the score be estimated?
- How similar should packages be? At which threshold should a match be accepted or rejected?

In order to answer these questions we designed a framework which allows us to deploy ASTLI with different matching strategies and parameters. Apart from how we perform matching, we also distinguish how we deal with the matching results. For these situations we implemented two modes of operation:

Production Mode This mode is intended for end users and follows the requirements stated in Section 4.1. In this mode, ASTLI prepares and prints results in a human readable format without further analysis.

Evaluation Mode In this mode results undergo further analysis. We question the correctness of the result by checking if a package has been matched correctly or if a package should have been recognized because it is in the database. This mode collects data for analysis and makes it available in an exportable format such as `csv` or `JSON`. The evaluation mode also provides logging of intermediate results for debugging purposes.

Both the ability for fine tuning and the different modes of operation affect the matching process. This increased complexity manifests itself in the architecture: The matching phase has been divided into multiple sub components, whereas the learning phase is composed of one single class. The following listing explains how we broke down the matching process into smaller tasks:

match This package consists of three classes. The class `MatchingAlgorithm` bootstraps the matching process. As already mentioned, there are multiple strategies to perform the matching which can be tweaked with parameters. The user decides upon strategy and parameters and `MatchingAlgorithm` interprets the user's directives and assembles the matching process accordingly. The class `MatchingProcess` directs the process and forwards intermediate results between sub tasks. The third class `SetupLogger` logs strategies and parameters before running the matching process. By doing so the class attaches information on the setup to the matching results, which renders the results more traceable.

find Given a package from an app, this component finds and offers appropriate package candidates from the database. We implemented two approaches of finding and offering candidates. The class `NeedleFinder` implements the idea described in the listing of Section 4.5.1, Step 1 and 2: First, it extracts the most *particular* fingerprints (*needles)* from the application package. Then it looks in the database (*haystack*) for similar fingerprints and offers the corresponding packages as candidates. The second approach `NameWithFallbackFinder` looks up packages by their name. If it does not find packages with the same name in the database, it falls back to the needle-haystack approach. `NameWithFallbackFinder` is intended for *in production* use: If the app is not obfuscated, we can leverage clear names to improve the detection accuracy.

score contains all classes that establish similarity between two packages. We implemented three approaches for comparing package hierarchies: `InclusionChecker` implements the signature based inclusion check described in Section 4.5.3. `SimilarityMatcher` implements the score computation described in Section 4.5.4. The third implementation `HybridMatcher` combines both approaches by using `InclusionChecker` as precondition for the `SimilarityMatcher`: We compute similarity score only if package inclusion $\subseteq$ holds. The `score` component further hosts an implementation of the Hungarian Algorithm[6], which aids all assignment related decisions in this component.

postprocess Contains all implementations that decide what do do with matching results, depending on the chosen mode of operation. In evaluation mode, ASTLI prints results and statistics in `.csv` format such that it can be exported and analyzed with an external framework. In production mode, we prepare the match result and print it a human readable table format.

Conclusion

This chapter examined how ASTLI was designed. It gave an overview over the architecture of the toolkit and described both purpose and dependencies of each component. We emphasized three aspects of the architecture: feature extraction, persistence storage of features and feature matching. We discussed how ASTLI benefits from other tools as `baksmali`, `dx`, `Active Objects` and `Hyper SQL` and we introduced two modes of operation named *production mode* and *evaluation mode.*

[6] imported from Kevin Stern, `https://github.com/KevinStern/software-and-algorithms`

6 Evaluation

Throughout this thesis we introduced concepts and ideas to solve certain aspects of library identification. We discussed theoretic benefits and made assumptions in order to justify our design decisions. In this chapter we introduce the evaluation framework, which helps us to verify our claims. Section 6.1 gives an overview on the evaluation strategies and motivates their design. Section 6.2, 6.3, and 6.4 go into more detail on each strategy.

6.1 Overview

The primary target of the evaluation is to find out how well ASLTI works and where its weaknesses lie. In order to gain comprehensive results and to minimize the probability of missing weak spots, we evaluate a large set of randomly chosen apps and libraries. Only with a large enough sample set we can compensate for bias introduced by outliers. Another target of the evaluation is to have feedback on the sanity of the code base. When implementing a new strategy, tweaking parameters or fixing a bug, it is often more important to receive feedback quickly at the cost of comprehensiveness. We compromise between comprehensiveness and time saving by introducing the three evaluation strategies. Table 6.1 compares key characteristics of the strategies with each other and the following listing summarizes each strategy:

Unit Testing This strategy gives immediate feedback on the codebase's state by checking if its components, the so-called *units*, fulfill the developers expectations. Unit testing is fast and its results shed light on the codebase' sanity. Section 6.2 explains how we applied this concept.

Quick Evaluation This strategy analyzes a homemade application and matches a small set of libraries. The goal of the quick evaluation is to give a first impression on accuracy and performance of the matching and to determine if all components collaborate as expected. Section 6.3 goes into more detail on this strategy.

FOSS Evaluation This strategy analyzes a comprehensive set of open source apps and matches a large set of libraries. The results of this strategy back our claims on accuracy and performance. Due to the size of the sample set, this evaluation takes the longest to perform. Section 6.4 explains, how we selected apps and libraries and how we designed the evaluation.

	Unit	Quick	FOSS
Framework	JUnit	JUnit	Gradle
App Sample Size	-	1 App	52 Apps
App Source	-	written	FOSS
Library Sample Size	-	2 libs	97 libs
Runtime	few seconds	$<$ 1 minute	$\sim$ 30 minutes
Result Verification	assertions	manually	externally
Dependencies	mocked	real	real
Evaluation Results in Section	-	7.1 and 7.2	7.3, 7.4, and 7.5

Table 6.1: Evaluation Strategies compared

6.2 Unit Testing

Unit Testing is a technique that verifies automatically if components (units) are fit for use. Thereby, the developer constructs a test case that runs the component and checks if it behaved in an expected way. Having a comprehensive test suite gives the developer immediate feedback on the sanity of the codebase and therefore confidence of his work: He can implement changes and by running the test suite, he can check if the changes introduced unexpected behavior. We use *JUnit*[1] to write and run our test suite. Each test case in this suite contains assertions on the behavior of ASTLI's components. The following sections explain patterns and concepts which we used when designing and implementing the test suite.

[1] http://junit.org/

6.2.1 Mock Objects

A unit test should check the behavior of exactly one unit. This assumption does not hold if the unit depends on other components. In such cases, the unit runs code from its dependencies and therefore relies on their correctness. Consider the example of a unit test (Figure 6.1) and a unit (Figure 6.2):

```
class UnitTestA {

  void testDoThis() {
    A a = new A(new B());

    assert(a.doThis().equals(
            "expected"));
  }

}
```

Listing 6.1: Test of Unit `A`

```
class A {

  void A(B b) {
    this.b = b;
  }

  String doThis() {
    String tmp = b.doThat();
    String result = trim(tmp)
    return result;
  }

}
```

Listing 6.2: Implementation of Unit `A`

In this example we test Unit `A` and make sure that Method `A.doThis()` behaves as expected. Now Unit `A` depends on `B` because `A.doThis()` invokes `B.doThat()` in Listing 6.2, Line 8. This implies that the outcome of `A.doThis()` relies on both the correctness of `A.doThis()` **and** `B.doThat()`. If `A.doThis()` does not return the expected result, we cannot be sure if the error lies in `A.doThis()` or `B.doThat()`. However, the test case of `A.doThis()` shall fail *if and only if* there is an error in `A.doThis()`. In other words: `B.doThat()` shall not influence the test case because `B` is not being tested.

A **Mock object** limits the scope of a test to the respective unit. It offers the same interface as the dependency but acts differently, because its behavior can be overridden. By doing so, we can assume that the mock follows our expectations in a particular situation, whereas the real dependency might deviate from our expectation. Mocking allows us to uncouple a unit from its dependencies and thereby exclude dependencies as causes of defect. Listing 6.3 improves the test case of Listing 6.1 by replacing `B` with a mock object.

We create and configure mocks with the *Mockito*[2] framework and we use this technique

[2] `http://site.mockito.org/`

```
class UnitTestA {

  void testDoThis() {
    B bmock = mock(B.class);
    when(bmock.doThat()).thenReturn(" expected ");
    A a = new A(bmock);
    assert(a.doThis().equals("expected"));
  }

}
```

Listing 6.3: Test of Unit `A` with mocked Dependency `B`

whenever creating a mock and adapting it to the situation is less complex than providing an actual implementation.

6.2.2 Arrange, Act and Assert

Each unit test follows the *Arrange*, *Act* and *Assert* pattern (*AAA*), which divides the test into three sections: In the *Arrange* section we set up all components and make sure that all preconditions for the test subject are met. This involves creating a particular instance of a scenario by instantiating the subject and creating and injecting mock objects. In the *Act* section we kick off the test by invoking the method/s that are being tested. Eventually, we verify the test case in the *Assert* section by comparing the results with our expectations.

6.2.3 Code Coverage

All in all we implemented a test suite composed of 86 test cases. We opted to write unit tests for units that require a certain degree of complexity and are therefore more error-prone. Units that perform auxiliary tasks, such as user interface handling, managing control flow and reporting, are covered by the strategy described in the Section 6.3. Our unit test suite covers 62% of all instructions and 66% of all branches. Figure 6.4 gives a detailed overview on the coverage status of each component.

Element	Missed Instructions	Cov.	Missed Branches	Cov.
astli.postprocess		24%		53%
astli.extraction		81%		74%
astli.match		0%		0%
astli.main		0%		0%
astli.score		80%		74%
astli.pojo		81%		65%
astli.db		69%		90%
astli.learn		0%		0%
astli.find		89%		n/a
Total	2,807 of 7,297	62%	182 of 532	66%

Figure 6.4: Jacoco Code Coverage Report

6.3 Quick Evaluation

The *quick evaluation* (QE) strategy gives feedback on how well components interact with each other. This strategy shares similarities with the concept of *integration testing* because both combine multiple units with external resources and test them as a whole. QE tests fill the gap between unit tests and FOSS evaluation, because they are much faster than the latter but cover the entire software stack as opposed to the former. The following sections motivate the design of this strategy and explain how matching results are being verified.

6.3.1 Design

The QE strategy does not only tell if the tool succeeds to detect libraries, but also how well the detection works. Its results give an intuition on how changes of the code base or parameters affect the tool's accuracy and performance. This aids the developer in making design decisions as it debunks poor ideas early on.

In order to provide rapid feedback, the QE strategy focuses on short run times at the cost of the result's significance. The detection rates these tests yield should be taken with a grain of salt because they stem from a small sample set. This set consists of one sample app, two libraries and a total amount of 150 learned packages. We wrote the sample app ourselves for the following reasons:

- When reviewing test results, we benefit from knowing how the app is designed and which libraries are in place. The best way to have this knowledge is to design the app ourselves.

- We need `.apks` build with different compilation and obfuscation settings. If we build the app ourselves, we can control all parameters of the build process.
- We wrote and built an app to get familiar with the development and build process of Android apps. This helped us to make assumptions on how apps are being built and delivered, which influenced the design of ASTLI.

QE tests also help to verify the sanity of parts that are not covered by unit tests. For this, QE tests use the same entry and exit points as would, which includes most of the auxiliary units implicitly.

6.3.2 Result Verification

During QE tests, ASTLI generates `.csv` files, which contain matches between application packages and library packages. The correctness of the matches is verified by comparing the package name. When testing obfuscated apps, matches cannot be verified as is because the names have been altered at build time. For this reason we implemented a parser for Proguard *mapping files* which we briefly introduced in Section 5.2. The parser extracts a dictionary from the mapping file which helps to restore the original names of classes, methods and packages.

The QE tests neither impose expectations on the detection rate nor on the run time; instead, the results require manual review. Automated verification has not been implemented because we are more interested in *how* results are generated and less in the actual results. The output of a QE test consists of Table 4.1 and of multiple log files. Each log file keeps track of events and intermediary results with a varying degree of detail. Fine grained logging allows us to skim over results and to investigate the reasons for certain conclusions.

6.4 FOSS Evaluation

The *Free and Open Source Software (FOSS)* Evaluation matches a comprehensive set of apps with learned libraries. The name refers to the Open Source licencing of analyzed apps. We chose Open Source apps because we can build them with custom obfuscation settings and we can extract information on included libraries. By knowning the used libraries we can verify the correctness of ASTLI's matches. Based on the assumption that FOSS apps exhibit the same bytecode composition and library inclusion as other apps, we can

generalize our findings on this set such that it holds for Android applications in general. Section 6.4.1 describes how we chose and populated the set of apps and how we prepared them for the evaluation. Section 6.4.2 explains which libraries we used for the evaluation. Section 6.4.3 describes how the matching process can be configured. Section 6.4.4 describes how we used Gradle to create a command line utility to start ASTLI and to generate the evaluation input.

6.4.1 Apps

To populate a large set of apps, we chose to crawl the *F-Droid* Repository[3]. This repository contains aproximately 1000 FOSS licensed Android apps. We were not interested in the `.apk` files from the repositories because we need to build custom `.apk` files from the source code of the app. We generated our apps by scripting the following tasks:

1. Crawl the *F-Droid* Repository and if the app description contains a *GitHub* URL, use `git clone` to check out the repository.

2. For each repository, check if the repository was built with Gradle. We require Gradle builds because we know how to extend the build script such that we can automatically build the app with different obfuscation settings.

3. For each Gradle app we filter the repositories where the command `./gradle build` would finish without error message. We focus on apps that can be build this way.

4. For each buildable app we append custom build types to the build script and build the app with them. Table 6.2 explains which build types we added and which transformations they perform.

5. Eventually we harvest the resulting archives and place them in a separate directory.

After these filters 52 applications remained, which formed our final app set.

6.4.2 Libraries

We chose to alter the build script of our collection of easily buildable apps such that we can harvest depending libraries of each app. For this we wrote a Gradle task that would download and copy the dependencies to a separate folder. By populating our library

[3]`http://f-droid.org`, accessed on 2017-01-27

Build Type Identifier	`1reg`	`2obf`	`3shr`	`4os`	`5opt`
Shrinking			✓	✓	✓
Obfuscation		✓		✓	✓
Optimization					✓
# Packages	711	711	463	486	447

Table 6.2: Build Types Compared

data base with dependencies from our app set, we make sure to provide enough positive matches. In the end we collected 97 libraries, which consist of 678 packages, 12.158 classes and 121.742 methods.

6.4.3 Matching Configurations

The FOSS evaluation tackles a variety of problems and each problem domain requires its own configuration of the matching algorithm. Table 6.3 shows all configurations and list their exact parameters. The threshold for matching confidence t_{mc} tells, how confident a match needs to be in order to be accepted. The threshold for package particularity t_{pp} decides, how particular a package needs to be in order to be analyzed. An exact description of the parameters and a motivation for the chosen values can be found in the respective section in Chapter 7.

Configuration ID	Used In	Matcher	Finder	t_{mc}	t_{pp}
`default`	Production	Similarity	NameWithFallback	.5	80
`tpp`	Section 7.3	Similarity	Needle	.5	0-200
`tmc`	Section 7.4	Similarity	Needle	none	80
`simmat`	Section 7.5	Similarity	Needle	.5	80
`hybmat`	Section 7.5	Hybrid	Needle	.5	80

Table 6.3: Configurations for ASTLI Matching Algorithm

6.4.4 Gradle Setup

Gradle did not only help us to populate a set of apps and libs, but also offered a command line interface to learn libraries and analyze apps from the FOSS dataset. For this we implemented the following tasks:

fossStore-<LibName>.jar learns the library `LibName.jar` from folder `libs/`[4].

fossStore learns all libraries in the subfolder `libs/`.

match-<AppName>-<BuildType>-<ConfigurationID> matches the app `AppName` build with `BuildType` and uses the matching parameters from `ConfigurationID`. The folder that contains the apps needs to follow this structure: Each app has its own folder in `apks/`. The folder contains different build types. If the build type is obfuscated, then the Proguard mapping file shall have be extended with the suffix `.txt`. Figure 6.5 states an example of the folder structure.

saveResultOf-<AppName>-<BuildType>-<ConfigurationID> runs the former `match` task with the same parameters and archives the results.

fossEvaluate-<AppName> runs the matching process with all build types of the app `AppName` with all existing configurations and archives the results.

fossEvaluate runs the entire evaluation with all apps, all build types and all configurations.

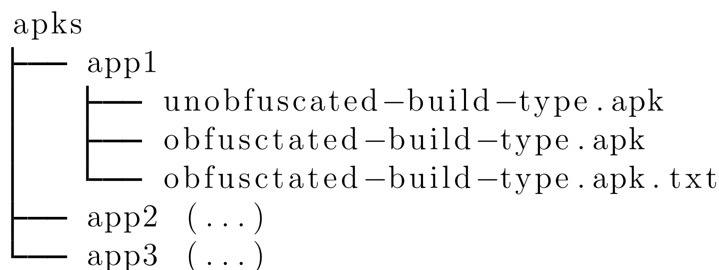

Figure 6.5: Example for App Folder Structure

Gradle scans the two folders `apks/` and `libs/` to create these tasks dynamically. The command `gradle tasks -all` lists all available tasks. We used *IPython*[5] to process ASTLI's output files, to compute metrics and to create the plots used in Chapter 7.

Conclusion

This chapter presented the evaluation framework which consists of three evaluation strategies: Unit Tests offer immediate feedback on the code base sanity. The FOSS evaluation

[4] This folder is located in `<astli-root>/src/integrationTest/resources/fossEvaluation`

[5] `https://ipython.org`, accessed on 2017-01-27

assesses ASTLI's performance by leveraging an extensive data set. The quick evaluation closes the gap between the former strategies, because it operates on real apps and libraries but takes little time to run. With the framework established, the following chapter introduces a set of questions regarding performance and classification quality of ASTLI and uses the framework to answer them.

7 Results

This chapter presents the outcome of the evaluation. Each section is structured as follows: We introduce a problem domain and pose one or more research questions. We explain how we prepared and executed the evaluation, including the chosen sample sources, the algorithm parameters and data analysis methods. We conclude each section with the outcome and discuss, motivate, and question the results.

In Section 7.1 we find out which HSQL mode is faster. In Section 7.2 we revisit *AST vectors* and *purged signatures* and analyze how well they identify code segments. In Section 7.3 and Section 7.4 we try to find reasonable parameters for the matching algorithm. In Section 7.5 we compare two package similarity measurements and use multiclass metrics to analyze how well ASTLI performs.

7.1 HSQL Embedded vs. Server Mode

HSQL can operate in *server mode* and in *embedded mode* (see Section 5.3.1): When running in the former mode, the HSQL DBMS runs in its own process and communicates with ASTLI over the network. When in embedded mode, HSQL and ASTLI run in the same process. Depending on the scenario, one mode might perform better than the other. In this section we introduce a likely scenario and explain how the chosen HSQL mode influences the scenario. Afterwards we introduce the evaluation setup and what we want to find out. We conclude this section with the evaluation results and its discussion.

Scenario

Consider the following scenarios:

- A user has a large set of libraries that he or she wants to feed into ASTLI's database.
- A user has a large set of apps that he or she wants to analyze.

In both cases ASTLI runs multiple times consecutively and each run requires a startup and teardown phase.

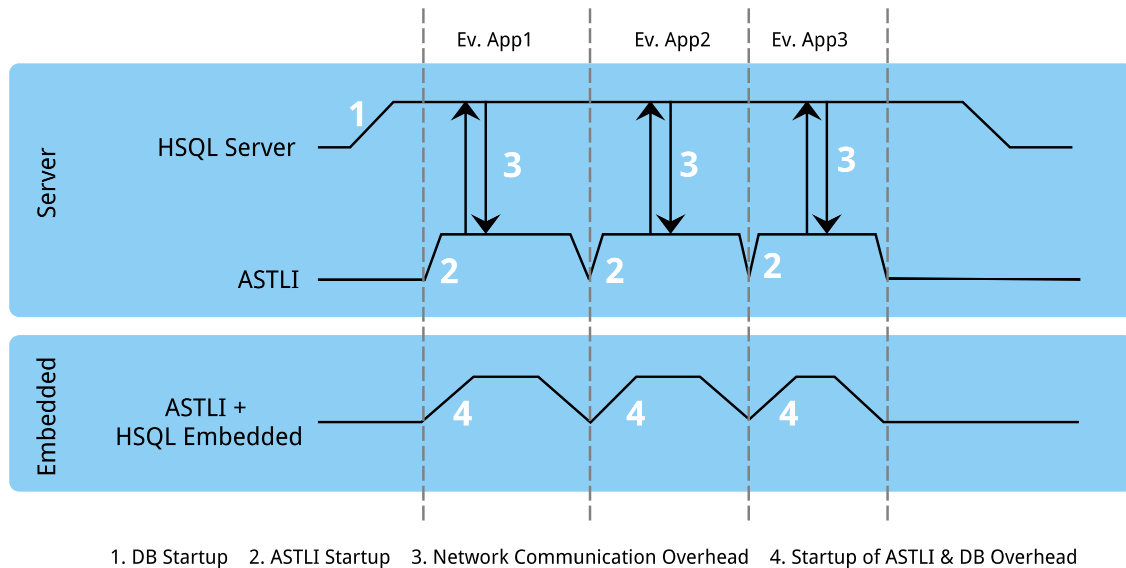

Figure 7.1: Timing Diagram of both HSQL modes

Figure 7.1 depicts a timing diagram and how the HSQL mode influences the runtime performance of the given scenario: In the upper part HSQL runs in *server mode*. The HSQL server process takes some time to start up (notice the flat angle of the rising edge in **1**). This startup time depends on the size of the database. Every time when the actual app analysis starts, ASTLI has a quick startup phase (steep angle of the rising edge in **2**). ASTLI communicates with the server process to fetch learned libraries (**3**). The lower part of Figure 7.1 depicts the timing of ASTLI in *embedded mode*. The startup phase of ASTLI is longer because the database needs its time to get ready. On the other hand, the actual runtime of ASTLI is shorter because the communication with the database does not run over the network stack.

Evaluation Setup

Given the two modes and their influence on the scenario, we want to find out which mode performs better:

- What has more impact on the runtime: network communication overhead or repeatedly prolonged startup overhead?
- Which mode of operation prevails in which situation?

In order to answer these questions we measure and compare the runtime of ASTLI in the following cases:

1. Learn a randomly chosen library from the FOSS library set (Section 6.4.2).
2. Learn all 97 libraries from the FOSS library set consecutively.
3. Analyze a randomly chosen app from the FOSS app set with the default matching parameters.

Evaluation Results

Figure 7.2 compares both modes in absolute numbers and depicts the relative difference between modes. In all three cases, the server mode takes much longer than the embedded mode. The additional startup and teardown time of the embedded mode carry no weight compared to the network communication overhead. We can conclude that the overhead of the network communication excels the startup overhead in any case. For this reason we operate HSQL in *embedded mode* for the evaluation and do not recommend using the *server mode* in production.

Scenario	Embedded	Server
Learn 1 Lib	8s	31s
Learn 97 Libs	544s	8447s
Match 1 App	10s	210s

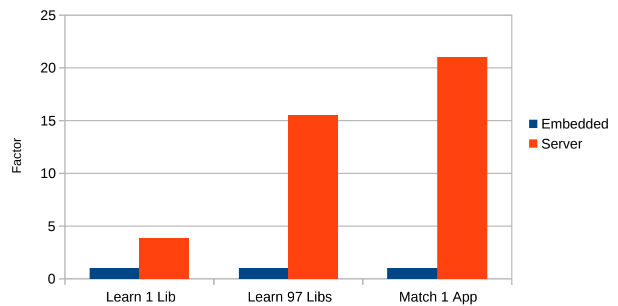

Figure 7.2: Runtime in Seconds (left), in Relation (right)

7.2 Comparing Features

In Section 4.2.1 we specified that our features shall identify code segments such that similar code segments share features, whereas unrelated segments have distinctive features. We introduced *AST vectors* and *sanitized signatures* and argued why these features fulfill the identity property. In this section we use our homemade application to show to which degree the identity property holds. This section is structured as follows: We introduce the evaluation questions and describe the evaluation setup. We conclude the section with results and its discussion.

Questions

We want to know how well the chosen features and the feature comparison algorithm identifies packages. We break this question down as follows:

- How well do *AST vectors* identify code segments?
- How well do *sanitized signatures* identify code segments?
- How well do *both features combined* identify code segments?

In order to answer these questions we deploy the confusion matrix $M = (m_{ij})$: It depicts how well we assign labels to Android application packages. It further visualizes the odds of labeling packages incorrectly. Each row resembles a label and each column resembles an actual package. The color of the cell m_{ij} indicates, how confident ASTLI is that the `.apk` package i belongs to the library package j. We can draw conclusions on the matching quality from the structure of M: If its main diagonal is confident and the rest is not, the features identify code segments without confusion.

Evaluation Setup

We leverage the packages from the homemade application of the QE Strategy[1] to build the confusion matrix. With 150 packages the sample set is large enough to have a variety of different packages and small enough to visualize its confusion matrix properly. With this set, we analyze our obfuscated, but not shrunken or optimized sample application. We compute the similarity score between each package from the application with each package from our learned libraries.

[1]See Section 6.3

Evaluation Results

This section analyzes three confusion matrices based on AST vectors, sanitized signatures, and both features combined.

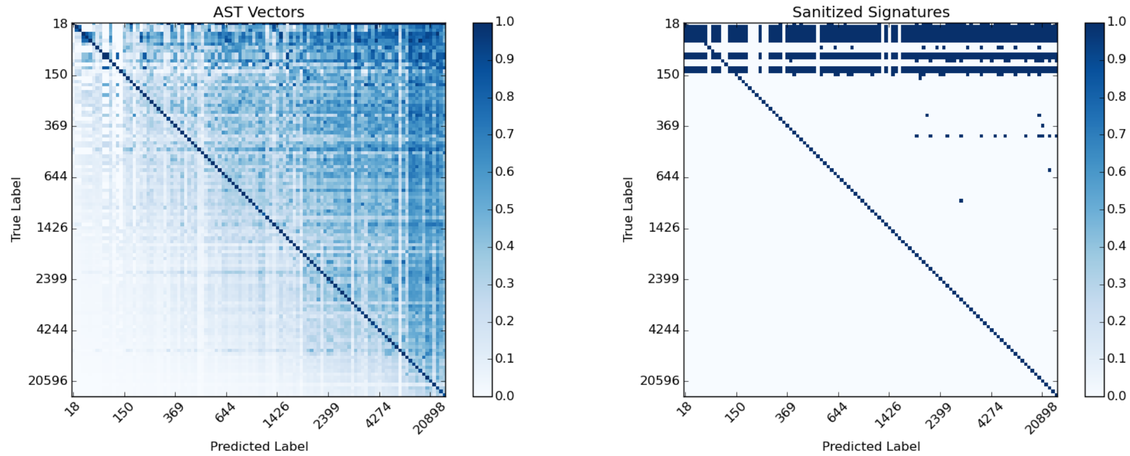

Figure 7.3: Confusion Matrix based on AST Vectors (left) and Sanitized Signatures (right)

Figure 7.3 shows the confusion matrix when establishing similarity using AST vectors on the left and sanitized signatures on the right. The x and y-axis are sorted by package particularity[2]. Packages with small particularity are located on top / left, whereas packages on bottom / right have high particularity. On the left matrix we notice that AST vectors similarity is confident on the main diagonal. However, the AST vector based similarity is prone to confusion. In the upper right part of the matrix we find that many packages have been labeled incorrectly with high confidence. We can explain this observation by the fact that a small application package can be easily mapped to a large library package. The other way around does not hold: large application packages are not confused with small library packages. We conclude that AST vector based similarity is a good start but does not provide enough accuracy.

The confusion matrix on the right of Figure 7.3 is based on sanitized signatures. This similarity measure only yields two values: It is either absolutely confident if a package is included in another package[3], or not confident otherwise. Compared to AST vector similarity, we notice that there is less confusion in the upper right part of the matrix, where less particular app packages are compared to more particular library packages. However, a lot of confusion arises with small application packages. The reason for this confusion is

[2] The concept of package particularity is explained in Section 4.5.2

[3] See Section 4.5.3

that large packages are more likely to contain all signatures of small packages. We conclude that sanitized signatures yield promising results if the package particularity is greater than 150.

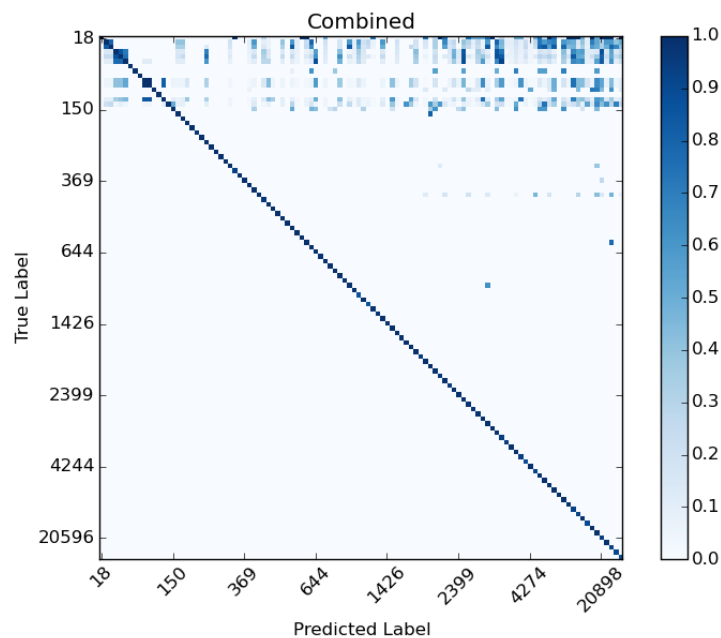

Figure 7.4: Confusion Matrix with both Features combined

The matrix in Figure 7.4 combines both features, which means: We only allow mapping of methods if their signature is equal. By combining both features, we can settle most of the confusion from the previous attempts. We notice that there is some confusion left in low particularity packages in the top rows of the matrix. The small clusters around the main diagonal in the top left area come from the fact that these packages are semantically related because they implement the same interfaces. We conclude that combining AST vectors and sanitized signatures enables accurate package matching and eliminates most of the confusion.

Conclusion

This section backs the claims we made about our features in Section 4.2.1. We showed that both AST vectors and sanitized signature based similarity perform good on their own, but that we can eliminate most of the confusion when we combine both features. We also saw that low package particularity is still prone to confusion, which is what we deal with in the next section.

7.3 Determining Package Particularity Threshold

Section 7.2 showed that confusion is more common below a certain package particularity. In order to prevent confusion and improve accuracy, we introduce the threshold for minimum package particularity (t_{pp}). Before we process an application package, we check if the package is particular enough. If not, we simply ignore the package because we cannot rely on its matches. The higher we choose t_{pp}, the more accurate the results become. However, with a high t_{pp} we ignore more packages and thus gain less insights from the analysis. We measure this influence with the *keep ratio*:

$$\text{keep ratio} = \frac{|\text{Analyzed Packages of App}|}{|\text{Packages of App}|} \tag{7.1}$$

In this section we try to find a reasonable value for t_{pp} such that we compromise between accuracy and keep ratio. We pose the following question: *How much particularity does a package need such that we get reliable results?*

Evaluation Setup

In order to find a good value for t_{pp} we leverage the FOSS sample apps and libraries from Section 6.4. We analyze all apps in all build types and use the `tpp` configuration. For t_{pp} we try values between 0 and 200. After we get the results from ASLTI, we build a confusion matrix and derive accuracy and keep ratio.

Evaluation Results

Figure 7.5 shows how t_{pp} influences accuracy and keep ratio. For small values of t_{pp} the keep ratio stays near 1, which means that we analyze almost all packages. The accuracy in this area is at .7, which means that 30% of all matches are incorrect. The higher t_{pp} becomes, the more packages we drop and the more accurate our results become. At $t_{\text{pp}} = 75$ the accuracy reached .9 and stagnates from there on, whereas the keep ratio keeps on declining. This observation matches our observation from Figure 7.3: Confusion arises below a certain package particularity, but decreases above a certain particularity. We conclude that we consider any value for t_{pp} below 80 as reasonable, depending on which goal (high accuracy versus high keep ratio) has more importance. We dissuade from values above 80 for t_{pp} because they do not improve accuracy. Our final recommendation for t_{pp} is 80 because it yields the highest accuracy without dropping too many packages.

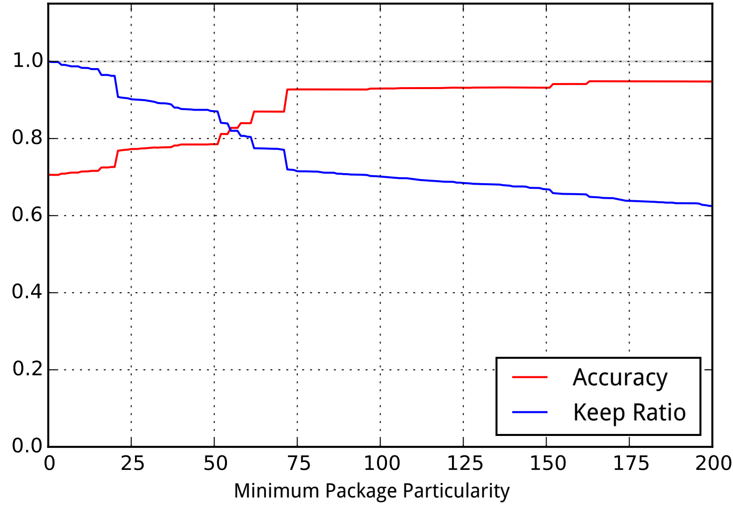

Figure 7.5: Influence of t_{pp} on Accuracy and Keep Ratio

7.4 Determining Match Confidence Threshold

In this section we try to find a reasonable value for the *match confidence threshold* t_{mc}, which is a parameter of the matching algorithm. We first introduce how we measure *confidence* in a match and how we decide on accepting or rejecting a match. We then explain how we designed the evaluation and discuss the evaluation results.

Match Confidence

ASTLI expresses the similarity between an application package p_a and a library package p_l with the similarity score $s(p_a, p_l)$ described in Section 4.5.4. This score depends on both how similar and on how particular packages are: More particular packages yield a higher score, whereas less particular packages yield a lower score. This imbalance is problematic when ASTLI decides on whether accepting or rejecting a match, because a constant threshold for all packages favors more particular packages over less particular ones with no regard to the actual similarity. We counteract to the imbalance by deriving the *confidence* of a match from the package similarity as follows:

$$\mathrm{confidence}(p_a, p_l) = \frac{s(p_a, p_l)}{s(p_a, p_a)} \tag{7.2}$$

The confidence between p_a and p_l is $\in [0, 1]$ because $0 \leq s(p_a, p_l) < s(p_a, p_a)$.

Match Confidence Threshold

We introduce the *match confidence threshold* t_{mc} which decides if a match is to be accepted or rejected, based on its confidence. The question we pose is: *How confident does a match need to be such that we can accept it?* If we require little confidence for accepting a match, we are likely to find more matches overall but we also report more incorrect matches. If we require high confidence, our matching results are more trustworthy but at the same time we are more likely to miss out on matches.

Evaluation Setup

In order to find the best value for t_{mc} we leverage our FOSS sample set from Section 6.4. We analyze all apps in all build types and use the `tmc` configuration. We remodel our multiclass problem into a binary classification problem with the One-Vs-All approach. This binary classifier tells whether a package is known (*positive*, *+*) or unknown to the system (*negative*, -). We transform each match into the new problem domain as follows:

- Replace learned library packages with *positive*.
- Replace others packages, such as not learned library packages or application packages, with *negative*.

Table 7.4 gives an example of this transformation. On the left we see a list of matches that ASTLI produced, on the right we see how the matches have been mapped to the binary classification. The following listing explains what happens in each line:

	Multiclass		**Binary Class**		
#	Actual	Predicted	Actual	Predicted	Confidence
1	`libA.pckgA`	`libA.pckgA`	+	+	100%
2	`libA.pckgB`	`libB.pckgC`	+	+	60%
3	`libA.pckgC`	`<no match>`	+	-	0%
4	`appB.pckgA`	`<no match>`	-	-	0%
5	`appB.pckgB`	`libA.pckgC`	-	+	20%

Table 7.1: Example of how ASTLI's Results are Mapped to Binary Classification Results

- Line 1 contains a correct match with maximum confidence (true positive).
- Line 2 contains an incorrect match, because `libA.pckgB` is not `libB.pckgC`. The binary classifier interprets it as true positive though because both packages belong to the learned libraries. This example should illustrate that this binary classifier only distinguishes between learned and not learned libraries, not that libraries have been identified correctly. Therefor the metrics retrieved from the binary classifier do not represent the multi classifier.
- In Line 3 ASTLI failed to recognize `libA.pckgC` (false negative).
- In Line 4 ASTLI correctly rejected to match `appB.pckgA` (true negative).
- In Line 5 ASTLI failed to reject `appB.pckgB` (false positive).

With our match results transformed into binary classifications, we build *receiver operational characteristics* (ROC) curves, which illustrate the performance of a binary classifier and reveal how the accept threshold for confidence influences both *T+* and *F+* rate. The ROC curves shed light on the separability of known and unknown packages and aid the search of a reasonable threshold value for t_{mc}. We do this for different build types to see how well we can distinguish known from unknown packages if certain code transformations are in place.

Evaluation Results

Figure 7.6 compares different ROC-curves of the binary classifier. The classifier separates known from unknown packages with high accuracy in almost all build types. The *area under the ROC-curves* (AUC) for these build types is above 99.5%. The only build type where our classifier performs suboptimal is the one with obfuscation, shrinking and optimizations activated. In this build type the AUC is 87.7%, which is still acceptable.

Figure 7.7 and Figure 7.8 shows how known and unknown package matches are distributed over their confidence. The red bar indicates the occurrence of unknown packages, the green bar the occurrence of known packages. Note that the y-axis is scaled logarithmically because matches with $\mathit{confidence} = 1$ and $\mathit{confidence} = 0$ tend to dominate the histogram. We notice that *regular* and *shrunken* app packages can be separated perfectly at $t_{\mathrm{mc}} = 0.8$. Separability in the confidence histogram of *obfuscated* app packages (Figure 7.7, right) is still good because the distributions barely overlap. The same holds for

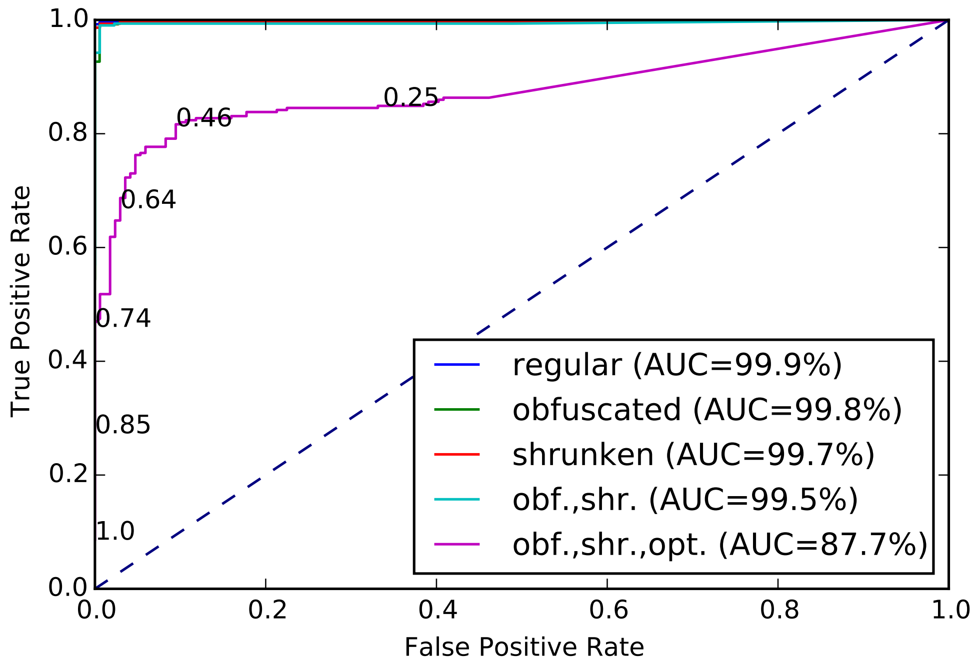

Figure 7.6: Comparing ROC-Curves of Different Build Types

obfuscated and shrunken app packages in Figure 7.8, left. As we learned in Figure 7.6, app packages with all possible transformations applied are the hardest to separate. Judging from the corresponding ROC-Curve, the best value for t_{mc} is around 0.5 because it optimizes accuracy. If either true- or false positive rate are significantly more important than the other, one could consider a value for t_{mc} that would favor either of them. However the influence of t_{mc} on true- or false positive rate is rather limited.

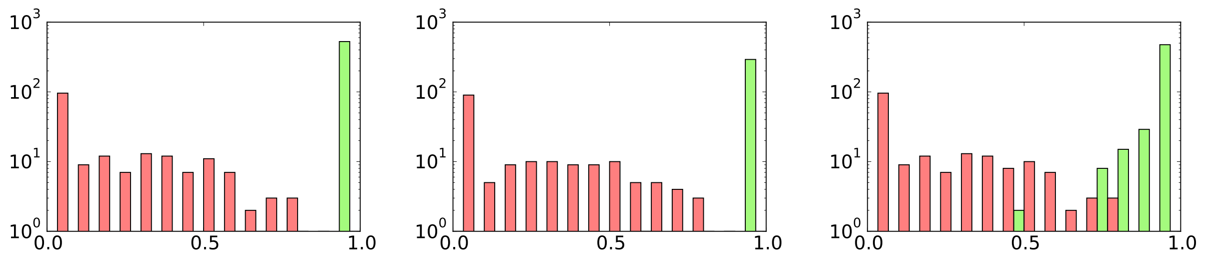

Figure 7.7: Confidence Histograms; From Left to Right: Regular, Shrunken, Obfuscated

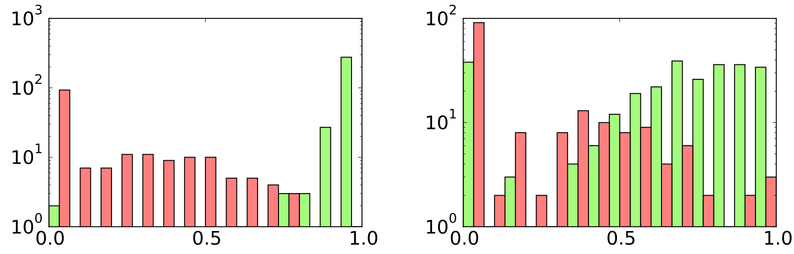

Figure 7.8: Confidence Histograms. Left: Shrunken and Obfuscated; Right: Shrunken, Obfs. and Optimized

Conclusion

This section defined the measure for confidence of a match and introduced the threshold t_{mc} for deciding on whether to accept or to reject a match. In order to find reasonable value for t_{mc} we transformed our multiclass problem into a binary classification problem and used ROC-curves to visualize the quality of the classifier. After analyzing the results we recommend $t_{\text{mc}} = 0.5$ because it is expected to yield the highest accuracy.

7.5 Comparing Matcher

In Section 5.4 we presented two approaches for estimating the similarity between two packages. In this section we evaluate both matchers in order to determine which one performs better. We first summarize the basic principles behind the matchers and compare them, followed by an explanation of our evaluation setup. We conclude the section with the evaluation results and its discussions.

Matchers

A matcher takes two packages p_a and p_l and computes the similarity $s(p_a, p_l)$ between those packages. We implemented three different approaches for matching:

- Inclusion Matcher[4]:

$$s_i(p_a, p_l) = \begin{cases} 1 & \text{if } p_a \subseteq p_l \\ 0 & \text{otherwise} \end{cases} \tag{7.3}$$

[4] See Section 4.5.3

- Similarity Matcher[5]:

$$s_s(p_a, p_l) = cost(M_{p_a,p_l}, f_{M_{p_a,p_l}}) \tag{7.4}$$

- Hybrid Matcher:

$$s_h(p_a, p_l) = \begin{cases} s_s(p_a, p_l) & \text{if } p_a \subseteq p_l \\ 0 & \text{otherwise} \end{cases} \tag{7.5}$$

The inclusion matcher was merely used for evaluation purposes in Section 7.2 because of its binary output. The other two matchers are intended for use in production. They differ in the fact that the hybrid matcher uses the package inclusion relation $p_a \subseteq p_l$ as a precondition for calculating the similarity, whereas the similarity matcher always computes the similarity. The implementation of the inclusion relation, on which the hybrid matcher is based, enables short-cut evaluation, which is used to speed up the computation in the case of $p_a \not\subseteq p_l$. This performance gain relies on the assumption that the sanitized signatures are invariant to all possible transformations. However, some Proguard optimizations alter the method signature (see Section 4.2.4). If sanitized signatures of methods are altered, the inclusion check might fail and we end up with $s_h(p_a, p_l) = 0$ and rejecting the match wrongfully, whereas $s_s(p_a, p_l)$ might still be high enough to achieve a correct match.

Evaluation Setup

We pose the following questions:

1. *How well does ASTLI recognize Application Packages?*

2. *Does our assumption that our chosen features are truly invariant to transformation hold? If not, in which cases it does not?*

3. *Which matcher performs better in terms of time and matching quality?*

In order to answer these questions, we leverage our FOSS sample set and analyze all apps in all build types. We perform the matching with both the hybrid (`hybmatch` configuration) and the similarity matcher (`simmatch` configuration). From the given results we derive the following multiclass performance metrics[13] using the formulas in Figure 7.9:

Accuracy tells how many application packages have been labeled correctly compared to all matches.

[5] See Section 4.5.4

Precision describes the ability of ASTLI to not mislabel packages.

Recall describes the ability of ASTLI to find all instances of a library package.

FScore describes the harmonic mean between Precision and Recall.

$$\text{accuracy} = \frac{1}{n}\sum_{i=1}^{l} tp_i \qquad \text{precision}_\text{M} = \frac{1}{l}\sum_{i=1}^{l}\frac{tp_i}{tp_i + fp_i}$$

$$\text{recall}_\text{M} = \frac{1}{l}\sum_{i=1}^{l}\frac{tp_i}{tp_i + fn_i} \qquad \text{F1Score}_\text{M} = \frac{2\ \text{precision}_\text{M}\text{recall}_\text{M}}{\text{precision}_\text{M} + \text{recall}_\text{M}}$$

Figure 7.9: Multiclass Metrics; $n \ldots$ amount of matches, $l \ldots$ amount of library packages

Results

This section compares the metrics of both matchers. Table 7.2 gives an overview over all metrics. The cells are colored according to which matcher performs better: green for similarity matcher and orange for hybrid matcher. The graphs in Figure 7.10, Figure 7.11, and Figure 7.12 compare the metrics. The build types are distributed along the x axis.

		regular	obfuscated	shrunken	obf.,shr.	obf.,shr.,opt.
ACCURACY	hybrid	97.17%	97.03%	96.51%	96.46%	57.63%
	similarity	96.76%	96.61%	93.30%	93.40%	78.83%
PRECISION	hybrid	97.11%	96.88%	96.10%	95.89%	34.31%
	similarity	98.03%	97.80%	94.81%	94.69%	70.64%
RECALL	hybrid	97.79%	97.56%	96.98%	96.72%	33.20%
	similarity	99.15%	98.92%	97.05%	96.80%	71.95%
F1	hybrid	97.13%	96.90%	96.02%	95.81%	33.31%
	similarity	98.17%	97.94%	94.94%	94.80%	70.32%
RUNTIME	hybrid	317s	336s	265s	294s	259s
	similarity	380s	388s	284s	310s	276s

Table 7.2: All Metrics of Hybrid Vs Similarity Matcher

ASLTI Performance. Then looking at the graphs in Figure 7.10 and Figure 7.11 we notice that all metrics perform well in all build types except the *obfuscated, shrunken and optimized* build type. In these build types the hybrid matcher scores above 95.8%, whereas the similarity matcher performs above 93.3%. The metrics of the *obfuscated, shrunken and optimized* build type collapse, especially the metrics of the hybrid matcher. After getting to the bottom of this observation, we realized: When apps are built with Proguard optimizations turned on, some of the optimizations alter method signatures. The altered signatures break the inclusion relation between original library package and transformed app package, which causes the hybrid matcher to wrongfully reject the match. With optimizations turned on, the metrics of the hybrid matcher drop below 50%, which is why the bar does not show up in the graphs of Figure 7.10 and Figure 7.11. The similarity matcher, which does not use the package inclusion relation as a shortcut, yields sub optimal but stable results when optimizations are activated.

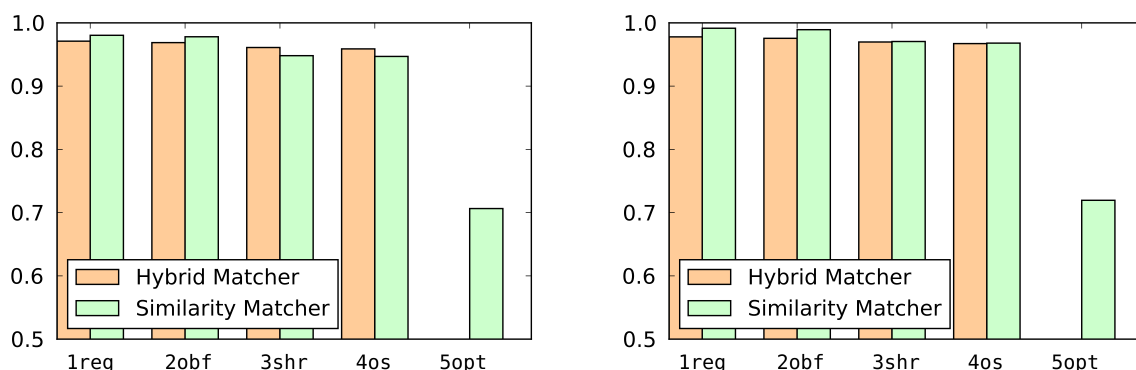

Figure 7.10: Precision (left), Recall (right)

The Better Matcher The color pattern in Table 7.2 reveal which matcher dominates which situation. The similarity matcher outperforms the hybrid matcher by far when it comes to optimized applications. On the other hand, the shortcut of the package inclusion relation helps the hybrid matcher to be slightly faster than the similarity matcher. The hybrid matcher is generally more accurate and more precise in with shrunken apps, but the differences in these sections are marginal. Overall, both matchers have their strength and weaknesses which depend on the situation. The biggest discrepancy lies in optimized applications, which is why we recommend the similarity matcher in production.

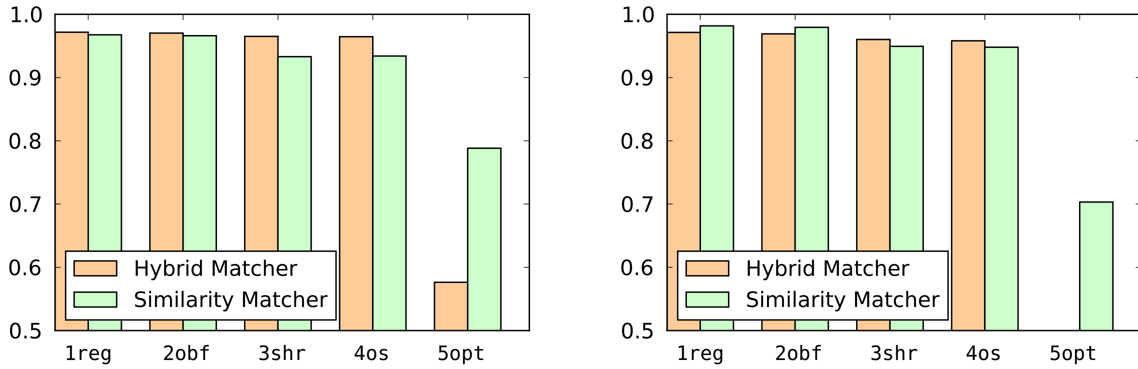

Figure 7.11: F1 (left), Accuracy (right)

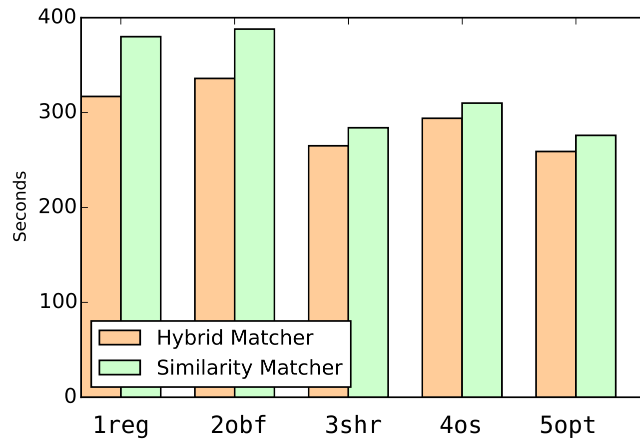

Figure 7.12: Runtime

Conclusion

This chapter discussed evaluation results. We learned that HSQL's embedded mode outperforms the server mode in all our scenarios. We examined AST vectors and purged signatures and learned that they identify obfuscated packages on their own, but improve the results if combined. In the confusion matrix we noticed that most confusion arises in packages with low particularity. For this we introduced the package particularity threshold t_{pp}. We tried different values for t_{pp} to see how it affects the accuracy of the results. We argued that $t_{\mathrm{pp}} = 80$ yields accurate results without dropping too many application packages. We introduced match confidence and the match confidence threshold t_{mc} for deciding on whether to accept or to reject a match. In order to find a reasonable value for t_{mc}

we remodeled our problem into binary classification and build ROC-curves. These curves showed how well ASTLI can distinguish known from unknown packages. After their analysis we recommended $t_{\mathrm{mc}} = 0.5$, which yields the highest accuracy. In our last evaluation we compared hybrid with similarity matcher by computing accuracy, precision, recall and f1 score from the matching results. We showed that the hybrid matcher runs faster but fails in the edge case of optimized apps. Apart from that, both matchers yield stable and reliable recognition results.

8 Conclusion

Libraries are a key ingredient in Androids app ecosystem because they fill gaps, ease development and enrich the user experience. At the same time their usage can be problematic for security, privacy, and even legally. This thesis presented the analysis tool ASTLI, which identifies libraries in Android apps. ASTLI uses libraries as ground truth: It learns libraries by extracting features and identifies libraries in apps by comparing them. The ground truth can lead to incompleteness of the library database, but is necessary to guarantee accurate results when dealing with obfuscation techniques.

Obfuscation techniques impede library recognition because they alter the code in various ways: *Identifier renaming* replaces names of variables, classes and packages, *shrinking* eliminates evidence of libraries by removing dead code, and *optimizations* rearange code segments and replace instructions. All these techniques are implemented by the obfuscator Proguard, which is of particular interest due of its seamless integration into the Android SDK and its convenient cost-benefit ratio for app developers.

AST vectors and *sanitized signatures* counter Proguard's transformations. The former is based on Abstract Syntax Tree of a method's body, and the latter constitutes a simplified version of a method's signature. ASTLI combines AST vector and sanitized signature to a *fingerprint*, groups fingerprints by their class and classes by their package. These groups of subgroups of fingerprints form a *package hierarchy* and represent both app and library packages. Package hierarchies are invariant to identifier renaming because neither AST vector nor sanitized signature depend on any identifier. They are also invariant to shrinking because the matching approach is geared to deal with loss of evidence. Package hierarchies are invariant to some but not all code optimizations, but since many optimizations strategies require careful tuning and testing, they are expected to occur rarely.

When analyzing an app and comparing app packages with library packages, ASTLI operates in two steps: First, it proposes a set of candidates by extracting *particular fingerprints* (*needles*) and searching for similar fingerprints in the database (*haystack*). After having a set of candidates, ASTLI computes the similarity between app package and candidates. The

similarity is expressed with the highest scoring assignment between fingerprints. ASTLI applies the Hungarian Algorithm to find this assignment. The most similar candidate is reported as a match, if the match is confident enough. ASTLI relies on `baksmali` to extract features, on `HSQL` and `Active Objects` to store and load features persistently, and on the build tool `dx` to convert `jar`-libraries into the `.dex` format.

The evaluation framework of ASTLI consists of three strategies: *Unit testing* helped us to verify the codebase sanity, the *quick evaluation* helped us to understand how design decisions affect ASTLI and the *FOSS evaluation* served to estimate the accuracy of ASTLI's predictions. The evaluation yielded the following conclusions:

- HSQL performs better in the embedded mode.
- AST vectors and sanitized signatures combined identify packages with little confusion.
- More particular app packages yield more reliable results.
- ASTLI is good at distinguishing known from unknown packages: The area under the ROC Curve is $> 99.5\%$ for obfuscated and/or shrunken apps and 87.7% for optimized apps.
- Depending on the applied code transformations, 96 - 97% (obfuscation, shrinking), resp. 78% (optimizations) of our predictions are correct.

The following listing summarizes ideas for future research:

Improve AST Vector AST vectors can be improved by covering *more instructions* of the Dalvik instruction set. More instructions increase the dimensionality of the vector, so the tradeoff between increased accuracy at the cost of increased complexity is subject to future research. Another way of covering more instructions without increasing the vector's dimensionality could be achieved by *grouping* semantically related instructions.

Improve Sanitized Signatures When encoding the types in sanitized signatures, ASTLI roughly differs between object types by assigning them to one of three categories. ASTLI could refine this distinction by leveraging the fact that *framework classes* (Activities, Views, Intents) cannot be obfuscated by Proguard because they are needed as entry points. A finer distinction between those types yields more unique signatures and is expected to improve accuracy. This idea was inspired by Derr *et al.*[11].

Cluster Similar Packages When learning library packages, *similar packages* can be grouped into clusters. This could speed up the matching process, because it reduces the similarity check to a single representative of the cluster.

Fine Grained Results Currently ASTLI reports matches between packages. The matching algorithm could be extended such that it reports matches on class- or method level. This could be useful when using ASTLI to detect license contamination, because developers do not always include entire FOSS libraries, but also smaller code chunks.

Leverage Relationships between Packages Many packages of the same library stand in relationship to each other. When visualizing packages in a hierarchy, two packages can have a common ancestor. Consider the packages `org.company.lib.pckgA` and `org.company.lib.pckgB`: Both packages have `org.company.lib` as common path. If ASLTI maps the app package `a.b.c.d` to `org.company.lib.pckgA`, it could favor other packages in `org.company.lib` as candidates for `a.b.c` because the relationship between package is preserved. ASLTI could further leverage the relationship between packages to exclude matches that violate established relationships.

CVE Lookup ASTLI could be extended such that it queries detected libraries for common vulnerabilities and exposures (CVE) in databases like the *National Vulnerability Database*[1]. This could speed up the app analysis.

[1] `https://nvd.nist.gov/`, accessed on 2017-02-02

Bibliography

[1] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security.," in *USENIX security symposium*, vol. 2, p. 2, 2011.

[2] R. Potharaju, A. Newell, C. Nita-Rotaru, and X. Zhang, "Plagiarizing smartphone applications: attack strategies and defense techniques," in *Engineering Secure Software and Systems*, pp. 106–120, Springer, 2012.

[3] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pp. 101–112, ACM, 2012.

[4] K. Yaghmour, *Embedded Android*. O'Reilly Media, 2013.

[5] M. McCullough and T. Berglund, *Building and Testing with Gradle*. " O'Reilly Media, Inc.", 2011.

[6] N. Elenkov, *Android Security Internals: An In-Depth Guide to Android's Security Architecture*. No Starch Press, 2015.

[7] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: local algorithms for document fingerprinting," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pp. 76–85, ACM, 2003.

[8] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Accurate and efficient structural characteristic feature extraction for clone detection," in *Fundamental Approaches to Software Engineering*, pp. 440–455, Springer, 2009.

[9] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on android markets," in *Proceedings of the 36th International Conference on Software Engineering*, pp. 175–186, ACM, 2014.

[10] L. Li, T. F. Bissyandé, J. Klein, and Y. L. Traon, "An investigation into the use of common libraries in android apps," *arXiv preprint arXiv:1511.06554*, 2015.

[11] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proceedings of the ACM Conference on Computer and Communications Security*, ACM, 2016.

[12] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval research logistics quarterly*, vol. 2, no. 1-2, pp. 83–97, 1955.

[13] M. Sokolova and G. Lapalme, "A systematic analysis of performance measures for classification tasks," *Information Processing & Management*, vol. 45, no. 4, pp. 427–437, 2009.

[14] J.-H. Ji, G. Woo, and H.-G. Cho, "A plagiarism detection technique for java program using bytecode analysis," in *Convergence and Hybrid Information Technology, 2008. ICCIT'08. Third International Conference on*, vol. 1, pp. 1092–1098, IEEE, 2008.

[15] A. Bartel, J. Klein, Y. Le Traon, and M. 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*, pp. 27–38, ACM, 2012.

[16] I. Santos, Y. K. Penya, J. Devesa, and P. G. Bringas, "N-grams-based file signatures for malware detection.," *ICEIS (2)*, vol. 9, pp. 317–320, 2009.

[17] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual*, pp. 421–430, IEEE, 2007.

[18] H. Tamada, M. Nakamura, A. Monden, and K.-i. Matsumoto, "Design and evaluation of birthmarks for detecting theft of java programs.," in *IASTED Conf. on Software Engineering*, pp. 569–574, 2004.