UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

A CROWD-SOURCING AND HEURISTIC APPROACH TO PROGUARD RULE GENERATION FOR THIRD PARTY LIBRARIES AND KEY SOURCE-CODE CONFLICTS

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

JUAN PABLO SUAZO SARROCCHI

PROFESOR GUÍA:
ALEXANDRE BERGEL
GEOFFREY HETCH

MIEMBROS DE LA COMISIÓN:
ALEJANDRO HEVIA ANGULO
PABLO GONZÁLEZ JURE

SANTIAGO DE CHILE
2021

RESUMEN DE LA MEMORIA PARA OPTAR
AL TÍTULO DE INGENIERO CIVIL
EN COMPUTACIÓN
BY: JUAN PABLO SUAZO SARROCCHI
DATE: 2021
PROF. GUÍA: ALEXANDRE BERGEL

UN ACERCAMIENTO USANDO CROWD-SOURCING Y HEURÍSTICAS PARA LA GENERACIÓN DE REGLAS PROGUARD PARA LIBRERÍAS EXTERNAS Y CONFLICTOS CLAVES

Dado que las aplicaciones móviles han adquirido protagonismo en rubro de la ingeniería de software, siendo estas uno de los tipos de software más utilizados en todo el mundo, se han creado diversas herramientas para facilitar su desarrollo. Los ofuscadores son una de estas, dedicadas a la optimización, reducción y ofuscación del código de las aplicaciones. Estos ofrecen claros beneficios al minimizar el tamaño de los archivos de las aplicaciones, haciéndolas más rápidas y reduciendo la legibilidad del código para proteger la propiedad intelectual.

A pesar de estas ventajas, la mayoría de las aplicaciones no utilizan ofuscadores. Esto es especialmente desconcertante en el ecosistema de Android, donde ofuscadores como ProGuard vienen incorporados y listos para su uso. Esta reticencia puede deberse a la dificultad que presenta implementar ofuscadores y mantener su código usable. Específicamente, la identificación de los conflictos entre la ofuscación y la funcionalidad del código, y la redacción de reglas de configuración para resolverlos. Estas dificultades incrementan cuando se trata de bibliotecas de terceros, ya que los desarrolladores tienen que lidiar con los conflictos provocados por código ajeno. La detección de los conflictos y la redacción de las reglas para abordarlos puede ser un proceso exhaustivo. Requiere que los desarrolladores tengan familiaridad con el código y con la sintaxis de las reglas para poder implementar ProGuard eficientemente. Esto podría disuadir a los desarrolladores de habilitar esta herramienta.

Se proponen dos soluciones para generar reglas de ProGuard para una aplicación de forma automática, eliminando los obstáculos que impiden a los usuarios habilitar ProGuard. La primera es un enfoque de crowdsourcing que pretende generar reglas para las dependencias de las aplicaciones. Nuestra solución estudia el código fuente de un número considerable de aplicaciones de código abierto. Nuestro enfoque limita los esfuerzos de los desarrolladores en la implementación de la ofuscación al código con el que están familiarizados. Lo hace detectando las dependencias de una aplicación, consultando la sabiduría de la multitud para conocer las reglas utilizadas en otras aplicaciones con las mismas dependencias y aplicando heurísticas para elegir las reglas pertinentes para la aplicación. La segunda solución pretende detectar algunas de las prácticas clave, que poseen conflictos reflexivos con ProGuard, dentro del código fuente de una aplicación y redactar reglas específicas para abordarlas.

Por último, observamos que la solución de reglas de dependencia es bastante precisa a la hora de decidir qué reglas debe incluir para las dependencias de una aplicación. Genera correctamente el 80% de estas reglas con un recall de 0,89. Por otro lado, lo hace arrastrando reglas que no son relevantes para la aplicación y, aunque no generen ningún error, desordenan los resultados y confunden al usuario. La segunda solución no tiene tanto éxito, ya que la detección de conflictos resulto ser un gran desafío. Aun así, la primera solución es de gran ayuda para poder habilitar ProGuard, ya que las reglas para dependencias representan la mayoría de las reglas necesitadas por aplicaciones.

ABSTRACT OF THE BACHELOR THESIS
TO OPT FOR THE TITLE OF
COMPUTING CIVIL ENGINEER
BY: JUAN PABLO SUAZO SARROCCHI
DATE: 2021
GUIDING PROF.: ALEXANDRE BERGEL

A CROWD-SOURCING AND HEURISTIC APPROACH TO PROGUARD RULE GENERATION FOR THIRD PARTY LIBRARIES AND KEY SOURCE-CODE CONFLICTS

As mobile applications have gained a prevalent spotlight in the field of software engineering, being one of the most used kinds of software worldwide, a diverse range of tools have been created to aid in their development. Obfuscators are one of these tools dedicated to the optimization, shrinking and obfuscation of an applications source-code. They offer clear benefits by minimizing applications file size, making them faster and reducing code legibility to protect intellectual property.

Despite the advantages of obfuscators, a majority of applications do without them. This is particularly perplexing in the Android ecosystem, where obfuscators such as ProGuard are built in and ready for use. This reluctance may be caused by the difficulties that developers must overcome to implement obfuscation while maintaining their code usable. Namely, identifying the inherent conflicts between obfuscation and code functionality, followed by redaction of configuration rules to address them. These difficulties are incremented when dealing with third party libraries as developers have to deal with conflicts brought on by external code.

The detection of the conflicts and the redaction of rules to address them can be an exhaustive process. It requires developers to have a firm understanding of the code and the redaction of rules in order to efficiently implement ProGuard. This might deters developers from enabling this tool.

We propose two solutions to generate ProGuard rules for applications automatically, removing the hurdles that prevent users from using ProGuard. The first is a crowd-sourcing approach that aims to generate rules for the dependencies of an application. Our solution mines the source-code of a sizeable number of open-source applications. Our approach limits the developers efforts in obfuscation implementation to the code they are familiar with. It does so by detecting the dependencies of an app, consulting the wisdom of the crowd for rules used in other applications with the same dependencies and applying heuristics to chose rules relevant for the application. The second solution aims to detect some practices that conflict with ProGuard within the code sources on an application and redact targeted rules to address them.

We observe that the dependency rule solution is quite accurate when deciding which rules need to be included by an application. It correctly generates 80% of dependency rules with a recall of 0.89. On the other hand, it does so by dragging along rules not relevant for the application that, while they may not generate any errors, clutter the results and confuse the user. The second solution is not as successful as the detection of reflective conflicts proves to be a great challenge. Still, the first solution greatly assists developers in enabling ProGuard as dependency rules are found to represent the majority of rules.

For my father Felipe, my mother Silvana,
my brother Andrés and my partner Jesu,
I could not have done it without you.

Love you always

Acknowledgements

First of all, I must thank my family. Your unconditional love and support has been essential throughout my life, especially during my time at university. Not only did you instill a sense of curiosity and wonder in me from a young age, which would guide me in my choice of career, you also made it possible for me to follow through with the ambitions that stemmed from it, supporting me every step of the way. You have provided an incessant stream of love and nurturing which enabled me to continue pushing forwards towards my goals, even in the most testing of times. For this and a multitude of other reasons, I am eternally grateful.

I must also thank Alexandre Bergel and Geoffrey Hecht for accompanying me throughout the process of this bachelor thesis. Your weekly guidance, insight and counsel was fundamental in the development of my work. Thank you for your dedication, continued commitment and, above all, thank you for helping me organize the way I work. Our weekly meetings allowed me to find some semblance of structure in a time where isolation and reclusion seemed to erase any traces of it.

I would also like to thank my wolf pack: Moira M., Andrés R., Felipe G., Antonia A., Camila C., Catalina U., Crescente E., Eric P., Macarena M., Natalia B., Trinidad C., Nicolás B. You are the best friends I could ever have asked for. Every moment I got to share with you filled me with joy and energy, which allowed me to find strength in the face of adversity. Thank you for being a source of light and laughter in my life.

Massive thanks go out to my scout family. To my amazing staff, Esther D. and Camilo. P., thank you for being a great team to work with. Your dedication, passion and capabilities are truly awe-inspiring. Thank you for your understanding and reliability, they allowed me to take the time I needed to work on my bachelor thesis while resting assured that my responsibilities were taken care of. To all "my kids" from the "Sol Naciente" clan, you are awesome! Thank you for allowing me to be a part of your life. Watching you guys grow and evolve into your best selves pushes me to do the same. I sincerely think you guys are amazing and that you should have the best scout guides the world has to offer. Striving to become the kind of example you deserve has made me a better guide, scout and person. Thank you.

A special thank you goes to my friend Romina M. Your understanding and support allowed me to complete my journey through the university. You were with me during my darkest hour and gave me hope that I could climb what, at the time, seemed like an impossibly tall mountain. Thank you.

Last but not least, I would like to thank my partner, Jesu. Your unwavering love and unconditional support has been essential during this last stretch of my education. Every sleepless night or unending day was made better by the smallest demonstration of your affection. Thank you for your kindness, encouragement and patience throughout this last process. Above all, thank you for believing in me every step of the way.

Table of Contents

Tables Index

Illustrations Index

Chapter 1

Introduction

1.1. Background

Currently, mobile applications represent one of the main development fields in the world. According to Evans Data Corporation, in 2016 there were up to 12 million mobile developers around the globe, near half of the total number of developers, projecting the number to increase to 14 million by 2020. Furthermore, 5.9 million of those developers were dedicated to Android applications [5].

Within the many development tools available in the Android Studio ecosystem, Android's IDE for app development, there are ProGuard and R8. Said tools are capable of shrinking, optimizing and obfuscating the bytecode of an Android application when building the APK file (Android Package). Shrinking is carried out by deleting unused classes, fields, methods, and attributes. Optimizations are made to further reduce its size by using more aggressive code removal tactics. Obfuscation is achieved by replacing the names of the remaining classes, fields, methods and attributes to short coded names.

Besides, reducing the applications size and making it faster, ProGuard also helps with protecting the intellectual property of the user. When obfuscated, code legibility is significantly reduced, making it much harder for a third party to reverse engineer the application and, therefore, avoiding the cloning or reproduction of the application. This is a highly relevant aspect of ProGuard as over 13% of all available Android applications are clones of legitimate apps [9].

It is clear then that obfuscators such as ProGuard are valuable tools to be used by developers, especially in the Android ecosystem where they are included and ready for use. Considering its benefits and ease of access, one would expect the use of ProGuard to be the norm for Android applications, but this is far from accurate.

1.2. The Problem

Despite all the benefits ProGuard offers, its use in existing applications is considerably low. Just 25% of applications have obfuscation enabled [6]. This might be due to the complications developers face when enabling ProGuard.

Said difficulties arise due to the inherent conflicts that code obfuscation generates while compiling and running an Android application. Practices like introspection and reflection will fail if ProGuard is allowed to act on the entirety of the source-code [14]. This is due to these practices depending on specific identifier names, which will be changed, or even removed, during shrinking, optimization or obfuscation.

In order to handle these conflicts, ProGuard uses rules stored in configuration files to control the way it operates on the code. These rules indicate which elements of the source-code must remain untouched for the application to be able to run without errors. These rules can be as broad as protecting a cluster of classes, or as specific as protecting a single field of a class.

Along with the rule system, warnings are raised when building the application, indicating possible conflicts that need to be resolved. These warnings can inform the user about a myriad of possible complications such as: conflicting rules, errors in the configuration of ProGuard and, more commonly, references to compile-time only dependencies.

Despite the efforts made by ProGuard to facilitate the use of obfuscation, these are not enough to convince developers to use this tool. From the 75% of developers who are aware of ProGuard and its benefits, 35% tried to enable it but abandoned these efforts due to difficulties while implementing it [6]. In fact, ProGuard's static analysis is not capable of identifying most of the problematic blocks of code. This is troublesome as a single error of this kind will prevent the application from being compiled or result in a run-time error. This is especially true in applications with a high use of reflection, as ProGuard admits in its documentation [17].

On the other hand, the redaction of ProGuard rules is an exhaustive process. One must go through the whole code identifying the sections that must be kept intact for the application to work. In addition, raised warnings will prevent the application from being built. It is then the responsibility of the user to examine each warning and decide if it is safe to be ignored, using a `-dontwarn` rule, or if it is indicating a real problem that must be fixed. This process is especially long in large applications.

Furthermore, all these problems are amplified by the use of external libraries, a prevalent practice in Android application development [13]. When using libraries, developers do not only have to redact rules relevant to their own code, they also have to redact rules for third party code. While some libraries may include a *consumerProguard* file which includes the necessary ProGuard rules, its use is not systematic. It is more probable to find rules in an applications README.md file, if at all [14].

1.3. Motivation

It can be seen then that, currently, the difficulties in the implementation of obfuscators such as ProGuard set aside an important portion of developers as they are the only ones responsible for overcoming the inherent conflicts between code functionality and obfuscation. This deprives them from the benefits and protection that ProGuard offers.

If a solution capable of generating rules for ProGuard existed, automatically protecting any class that requires it, all of the difficulties of implementing obfuscation would be removed. The said solution would remove the bottleneck that stands before ProGuard implementation and allow any developer to benefit from it. This would be of great help for the Android developing community, one of the largest developing communities in the world.

1.4. Objectives

Our general objective is to develop a solution capable of generating the rules needed by an application to enable ProGuard without any errors automatically. In order to achieve this, we focus our work on the completion of the following specific objectives:

1. **Studying the F-Droid Repository:** We aim to gain better understanding about the relationship between Android applications and ProGuard. To achieve this, we will look at the applications in the F-Droid repository, the largest catalogue of free and open-source applications for the Android platform.
2. **Redaction of Dependency Rules:** To address the conflicts brought on by third party code, we aim to generate the rules needed by the dependencies of an application in order to function correctly when ProGuard is enabled.
3. **Detection of Source-Code Conflicts:** We also seek to detect classes of the source-code of an application which may conflict with obfuscation and, therefore, cause errors while compiling or running the application. We will focus on detecting the following practices that are known to cause errors if ProGuard is enabled:
    - Data Classes.
    - Java code called by JNI.
    - Resource loading from APK.
4. **Redaction of Rules for Source-code Conflicts:** We aim to redact rules that will solve the conflicts detected in the source-code.

It is considered that the completion of these objectives will result in a solution that allows developers to easily enable ProGuard in their applications without having to worry about the redaction of any rules.

1.5. Developed Solutions

The work done resulted in two solutions. First, the main focus of our work, a dependency rules solution. We focused our work around the problem of dependency rules as previous work has been done in respect to rules for the source-code of applications. Additionally, dependencies present a particular problem for developers as the lack of knowledge about their source code makes the process of rule redaction difficult for them.

A secondary solution was developed in order to detect common practices that rely on reflection on the source-code of an application.

1.5.1. Dependency Rules Solution

The proposed solution for dependency rules works using a combination of crowd-sourcing and heuristics to determine the rules needed by the dependencies of an application.

To achieve this, we first parse F-Droid, a repository of 2955 open-source Android applications. This is done by developing a Python library consisting of classes designed to model F-Droid and Android Studio projects. These classes parse the files of the applications inside F-Droid, allowing us to study their composition. We then analyze each of the applications, determining their third party dependencies, whether they are enabling ProGuard or not and which rules are they employing and save this information into a database. This way we are crowd-sourcing the rules needed by the dependencies used in Android applications.

In order to generate rules for an application A, our solution uses the previously mentioned Python classes we developed to detect the dependencies of A. It then looks in our database for other applications which use one or more of said dependencies, have ProGuard enabled and contain rules in their configuration files. The solution retrieves said rules and then applies diverse heuristics to determine which rules being used across said applications could be useful for A.

Finally, the solution cleans up the selected rules, removing rules that are not relevant to the application, and writes the remaining rules into a ProGuard configuration file.

1.5.2. Application Specific Rules Solution

The solution created to detect classes with possible conflicts with ProGuard works by using a combination of static analysis of the code, studying of the composition of the classes and heuristics to determine which classes need to be protected.

The application specific rules solution analyses all the Java or Kotlin classes found in the source-code of an application and detects if they are data classes by looking at their

composition, specifically the relationship between their fields and methods. It also searches in these classes for methods calls looking to load resources from the APK. In both cases, the solution records which classes were detected to contain said practices to generate rules for them later.

Additionally, the solution analyses any native code present in the source-code of the application, namely its C/C++ code files. Here it searches for calls that are referencing Java/Kotlin classes of the source-code and the specific package names of said classes. The solution records these package names to, later, generate rules for them.

The solution then uses a rule redacting module to generate rules for all classes which were detected to need them. We generate specific rules designed to target each of the different conflicting practices detected.

1.6. Validation

Lastly, we show the results of an experiment designed to test our solutions. This experiment serves as validation for our solution, testing how effective they are at generating rules for applications.

1.6.1. Dependency Rules Solution Validation

The dependency rule validation experiment consists of attempting to replicate the rules present in applications which have ProGuard enabled. We randomly select 50 applications fit for testing, those which have a relevant use of third party libraries and ProGuard rules, and attempt to recreate their existing rules.

The dependency rules solution is then showed to be able to replicate correctly, on average, 73% of all original rules, and create 100% of the rules for 30% of the applications. These results exemplify the prevalent role third party libraries have in the difficulties of enabling ProGuard. In addition, only 6.38% of the correctly generated rules are protecting more code than necessary, so the resulting rules are quite specific.

Also, only 16% of the missing rules are dependency rules, showing this solution to be effective in generating rules for dependencies. In fact, the dependency rules solution has a recall of 0.89, meaning that it does not leave out useful rules when selecting them.

On the other hand, the computed F1-score is 0.06, meaning that many rules generated are false positives. This is a result expected when using heuristics, as by definition they are used to trade precision for better results. These extra rules were included by applications with ProGuard enabled, ergo, if not included in the original rules of an application, the referenced classes must also not be present. Therefore, their inclusion will have no effect, as they are pointing to nonexistent classes within the source-code of an application. While these extra

rules may have no negative effect, we still wish to remove them for the sake of the clarity of the generated ProGuard configuration files which hold the rules. By implementing a rule cleanup phase in our solution, we are able to nearly double the F1-score, but this result is still lower than desired.

It is seen that the solution, while generating unnecessary clutter in the results, is effective in generating rules for the dependencies of Android applications in the F-Droid repository.

1.6.2. Application Specific Rules Solution Validation

Similarly to the previously presented validation experiment, the one designed for the application Specific rules solution consists of attempting to replicate the rules present in applications which have ProGuard enabled. We randomly select 50 applications fit for testing, those which presents a relevant use of rules targeting their own source-code, and attempt to recreate their existing rules.

The experiment shows that only 7% of the generated rules are referencing conflicts targeted by rules in the original configuration files. But, if we disable data class detection, this result changes to 50%.

These results suggest that more work is needed to be able to effectively detect source-code classes which require detection, especially when detecting data classes.

Chapter 2

State of the Art

This chapter presents the context behind our problem. We talk about the technologies that are involved, how they work and how they are relevant to our solution.

First, we explain the general aspects of the Android Studio development platform which need to be understood in order to comprehend how our solutions work.

Secondly, we talk about the ProGuard obfuscator. We explain how this tool works and its effect on the source code. We also present the rule system used to configure ProGuard, the syntax of these rules and the different types that ProGuard offers.

Finally, we talk about the currently available solutions that seek to generate ProGuard rules for application. We present their approaches and what their results were.

2.1. Android Studio

Android Studio is the official IDE offered by Google for Android application development. It was announced and released as early access on May 2013 [16]. It was created by Google in collaboration with JetBrains. They based its design on IntelliJ, the Java IDE offered by JetBrains, as Android development is done in said language [8]. But in contrast to IntelliJ, Android Studio was designed specifically for Android development [16].

2.1.1. Projects & Structure

Each Android application is developed inside an Android Studio project. This is the directory where the whole work-space of an application is defined, from its source-code and assets to the tests designed for the application [3]. An example of an Android Studio project

can be seen in ***figure 2.1***.

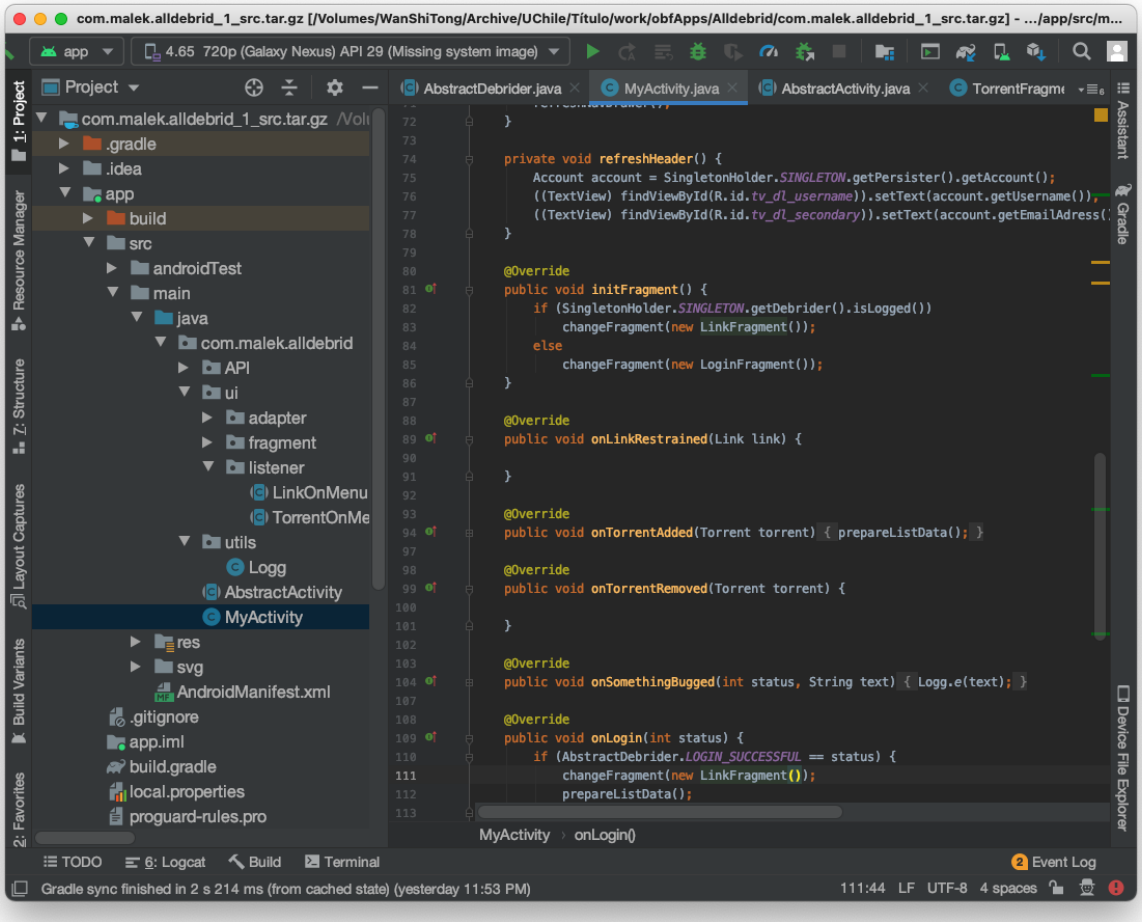


Figure 2.1: Android Studio Project of the Alldebrid Open-Source Application.

In the above figure, we can appreciate most of the standard structure that all Android Studio projects share. This structure is going to be relevant to the work presented in this bachelor thesis, therefore it is explained in more detail below, as done in the documentation of Android Studio [3]:

- `module-name`: Modules are a way of separating an application in many sub-applications, dividing your project into discrete units of functionality. A simple application may consist of just one module and a complex application may contain many of them.
  - `build`: This directory holds all the files outputted when generating a build of the application. This is explained in more detail in the ***Builds & Gradle section*** below.
  - `libs`: This directory, not shown on ***figure 2.1***, contains any private library the developer is using on its application.

  - `src`: This directory holds all of the code and resource files for the module. It is organized in the following sub-directories:
    * `androidTest`:This directory contains the code needed for testing the application on a hardware device or an emulator.
    * `main`: This directory holds the actual code and resources used by our module and application to run.
      · `java`: Contains all Java code sources.
      · `res`: Contains the resources of the module. Drawable files, layout files, and UI string are all held in this directory.
      · `jni (not shown)`: Contains all native code using the Java Native Interface (JNI).
      · `gen (not shown)`: Contains all of the Java code generated by Android Studio.
    * `test`: Contains the code for unit tests for the Java code of the application.
  - `build.gradle`: Build script that configures the building process of the module. This is explained in more detail in the ***Builds & Gradle section*** below.

- `build.gradle`: Build script that configures the building process of the whole application. This is explained in more detail in the ***Builds & Gradle section*** below.

Additionally, the code inside the `src/main/java` directory is structured using Java packages. This is the practice of saving Java classes in ordered directories depending on the type and functions of each class [15]. This can be seen in ***figure 2.1***, where the Java code sources are saved in different directories depending on if they are used either for an API, for the UI of the application or if they are utility classes. Each of these directories is then further separated in sub-directories if needed. The convention for naming of packages is to start their name with the reversed Internet domain name of the developing company, as to avoid naming conflicts, continued by the names of the directories which hold Java files. These names must use only lowercase letters, numbers and underscores [15].

This ordering of the Java files results in a **package structure** where each Java source file is contained by a package dedicated to files similar to it. Within this structure, each file has a **package name** that references its location within the package structure. For example, the package name for the ''`Logg`'' class seen in ***figure 2.1*** would be `com.malek.alldebrid.utils.Logg` while the package name for the ''`LinkOnMenu`'' class would be `com.malek.alldebrid.ui.listener.LinkOnMenu`. These names can then be used to reference these classes in a precise way. If we desired to use the ''`Logg`'' class in another Java class, it would suffice to use the following import declaration: `import com.malek.alldebrid.utils.Logg`.

Now that we know about the basic structure of a project, we need to learn how Android Studio takes the files contained inside this project and generates the executable file that will be installed on Android smart-phones.

2.1.2. Builds & Gradle

Android Studio allows the user to generate builds of the developed application. When talking about **generating builds**, or building an application, we refer to the process of packaging all of its parts in one file which is then used to install said application into a smartphone. The file generated by building an Android application is called an "Android Application Package", or APK, which is a compressed file composed of the code, assets and any other elements the application needs to run correctly.

When created, Google designed Android Studio to facilitate the configuration and customization of the build process and the easy creation of build variants, different versions of the same app [8]. To achieve this, Android Studio included Gradle in its ecosystem. [16].

Gradle is an open-source build automation tool that allows developers to generate builds of an application. It also allows the user to configure and customize the builds and even define different builds of an application meant for different purposes. This can be useful for developers as they might wish to have a debug build meant for testing, along with a free-to-use build with limited content and a premium build meant for paying users [8].

In order to configure the way Gradle builds an application, one must create a build script named ***build.gradle*** [16]. These scripts define all the parts that need to be packaged into an APK when generating a build. Also, the ***build.gradle*** file is used when defining the different possible builds of an application. On it, the user declares the names and configurations of each build variant [2]. The two main aspects of these configuration files we are interested in is how they configure the building process for different build variants and how they declare the dependencies of an application.

Configuring Build Variants

An example showing the declaration of two build variants can be seen in ***code 2.1***. This example is taken from the Android Studio documentation [2].

Code 2.1: Example ***build.gradle*** file.

```
android {
    defaultConfig {
        manifestPlaceholders = [hostName:"www.example.com"]
        ...
    }
    buildTypes {
        release {
            minifyEnabled true
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }

        debug {
            applicationIdSuffix ".debug"
```

```
        debuggable true
    }
  }
}
```

It can be seen that the declarations for a "release" and "debug" build are made inside the `buildType` block. These are the standard build variants that are created automatically by Android Studio [2]. By definition, all the code used by the release build is stored in the `src/main/java` directory of the project. The code contained in this directory is also shared with build variants.

In turn, any code used only by a specific build, must be placed in a directory named after its respective build. For example, all code used exclusively for the "debug" build must be placed in the `src/debug/java` directory. This way Gradle is able to determine which code is meant for which of the different builds. Therefore, the APK generated for the "release" build will contain all code in the main directory while the APK generated by the "debug" build will additionally contain the code in the debug directory. These APKs are stored inside the `build` directory of the project, where they are separated in different directories named after their respective build variant.

We also observe from the figure how both build variants have different configurations, declaring the value for different variables associated with the build process. For example, the "debug" build variant declares the `debuggable` variable as true. This will result in Gradle building an APK which includes the debugging tool offered by Android Studio.

Dependencies

As part of the build process, Gradle has to make sure to include all of the code that the application needs to run. If an application uses third party libraries to function, the code of said libraries must too be included in the APK generated when building it. This is done by declaring these third party libraries as dependencies of the application on the ***build.gradle*** file.

To add a dependency, this must be declared inside the `dependencies` bock of the build script in the manner shown in ***figure 2.2*** [1].

Code 2.2: Example ***build.gradle*** file declaring dependencies.

```
apply plugin: 'com.android.application'

android { ... }

dependencies {
    // Dependency on a local library module
    implementation project(":mylibrary")

    // Dependency on local binaries
```

```
10    implementation fileTree(dir: 'libs', include: ['*.jar'])
11
12    // Dependency on a remote binary
13    implementation 'com.example.android:app-magic:12.3'
14 }
```

The two initial declarations specify private libraries present in the developers work-space. The last dependency declaration specifies the name, version and group of a third party library being used by the application. In fact, `implementation 'com.example.android:app-magic:12.3'` is shorthand for `implementation group: 'com.example.android', name: 'app-magic', version: '12.3'` [1].

These dependencies can be declared with different configurations. For example, the `implementation` configuration will add the dependency to the compile path (all files that need to be compiled when building) and will package its source-code to the APK outputted when building. In contrast, if this dependency was declared as `compileOnly 'com...'`, the dependency would be added to the compile path but its source-code would not be included in the APK generated by the build. Other configurations include: `api`, `runtimeOnly`, `annotationProcessor`, `lintChecks`, `lintPublish`, `apk (deprecared)`, `compile (deprecated)` and `provided (deprecated)`.

This way, Gradle is able to know which third party libraries need to be downloaded, how they are being used by the application and what to do with them when building an application. From now on we refer to the third party libraries used by an application as its **dependencies**.

2.2. ProGuard

Next, we explain the basics of the obfuscators supported by Android Studio. Depending on the version, the supported obfuscator may be ProGuard or, in newer versions, the upgraded R8 [4].

From now on, we will be referring only to ProGuard as the standard obfuscator of Android Studio. This is done as R8 and ProGuard are very similar in how they work. Indeed, they are both configured in the same way [4], so they are equivalent for the purposes of the work presented in this bachelor thesis. We use ProGuard as a reference as it has been present in the Android Studio ecosystem since launch, while R8 was released in October 2018, and Android developers might be more familiar with this name.

ProGuard is a Java class file shrinker, optimizer and obfuscator. Its goal is to analyse the source-code of an application during the building process, making changes to it that minimize the size of the generated APK, make it work faster and reduce its legibility.

To enable ProGuard on an Android project, one must add the `minifyEnabled = true`

declaration on the ***build.gradle*** file of the application, along with specifying the location of the ProGuard configuration files [4]. This can be seen in the "release" build variant declared on ***code 2.1***.

2.2.1. How Does ProGuard Work?

Before ProGuard can start making changes to any source-code file, it first has to carry out an analysis of the contained code. In order for this analysis to work, the user must define the entry points of the application. These are the classes in the source-code where ProGuard will start its analysis. These entry points must include all classes that the Android platform may use to open the Activities or services of an application [4].

Android provides some general entry points which are defined in the standard android ProGuard configuration file [4]. This file also contains some base configurations for ProGuard to act on Android applications. These configurations can be added by using the `getDefaultProguardFile()` call, as seen in ***code 2.1***. Additionally, the user can define personalized entry points.

Once ProGuard knows the entry points, or seeds as they call them, it begins traversing all of the code to determine which classes, methods, field and attributes are actually being used, and which conform dead code that is never referenced. ProGuard is able to do this as it is capable of processing the Java and Kotlin languages [17].

ProGuard makes changes to the code in the following three steps:

1. **Shrinking:** The first step carried out is to delete all the code that ProGuard deems safe to remove. This includes all dead code, previously identified during the analysis, along with any annotations present. In this step is where we see the biggest reduction in file size [11].
2. **Optimization:** Next, ProGuard carries out a deeper inspection of the code with the objective of further reducing its size and making the application faster. This is achieved by deleting even more unused code or rewriting code into a shorter version of itself. For example, in this step ProGuard may make, among many others, the following changes to the source-code [4]:
   - Delete else clauses which are never reached.
   - Remove declared methods that are only called once and in-lining them at the call location.
   - Merge two classes if a class has a single unique subclass and the class itself is never instantiated.
3. **Obfuscating:** Finally, ProGuard assigns new short and coded names to the remaining classes and members that remain in the code and which are not entry points. It refactors the whole code to reflect these changes globally.

Example The shrinking effect that ProGuard has on a Java class can be seen in the following exemplifying code. For the purposes of this example, the presented class is the only one present in an application.

Code 2.3: Example of a Java Class.

```
public class AClass{
  private String AVariable = "Hello World !"

  // a used method
  private String aUsedMethod(){
    return self.aVariable
  }

  // not using this
  private String anUnusedMethod(){
    return "Bye Bye world!"
  }

  public static void main(String[] args){
    AClass object = new AClass()
    String message = object.aUsedMethod()
    System.out.print(message)
  }
}
```

As seen from ***codes 2.3*** and ***2.4***, during the shrinking step, ProGuard analyses the code and determines that the `anUnusedMethod()` method is never called by the rest of the code. Therefore, this method is removed from the code, ultimately reducing the size of the file containing it.

Code 2.4: Example of a Shrunken Java Class.

```
public class AClass{
  private String AVariable = "Hello World !"

  private String aUsedMethod(){
    return self.AVariable
  }

  public static void main(String[] args){
    AClass object = new AClass()
    String message = object.aUsedMethod()
    System.out.print(message)
  }
}
```

After shrinking the code, ProGuard then proceed to obfuscate its contents. The effect obfuscation has on a class are exemplified on ***code 2.5***.

Code 2.5: Example of an Obfuscated and Shrunken Java Class.

```
public class a{
  private String b = "Hello World !"
  private String c(){
    return self.b
  }
  public static void main(String[] args){
    a d = new a()
    String e = d.c()
    System.out.print(e)
  }
}
```

On this code snippet we can see how ProGuard renames and refactors the class and its members, using short meaningless codes for their new names. This simple example already shows how obfuscation significantly reduces the legibility of the code. It can be seen that the names and codes are mapped in the following way:

- AClass → a.
- AVariable → b.
- aUsedMethod → c.
- object → d.
- message → e.

Finally, ProGuard optimizes the class, resulting in the class seen in ***code 2.6***. Here we can see how the further modifications to the code result in the elimination of a method that was used a single time. Instead, ProGuard moves its definition to the place where it is called. ProGuard also detects the declaration of a variable that is only used once to hold the returned value and replaces it by writing its contents directly on the location they are needed. This results in a smaller code that takes fewer steps to complete its task.

Code 2.6: Example of an Optimized, Obfuscated and Shrunken Java Class.

```
public class a{
  private String b = "Hello World !"

  public static void main(String[] args){
    System.out.print(new a().b)
  }
}
```

2.2.2. ProGuard Configuration Files

We have talked about how the user has to define the entry points to its code so that ProGuard knows where to begin its analysis of the code. This, and other configurations, are communicated to ProGuard by declaring configurations files in the ***build.gradle*** build script as seen in ***code 2.1***, they are declared by stating their path relative to the build script. These are text files that contain rules which determine the way ProGuard acts on the source-code of an application.

There is a myriad of different rules that can be employed by a developer, for example, the `-optimizationpasses <n>` rule tells ProGuard to repeat its optimization step n times in the hopes of achieving better results. Additionally, the `-dontoptimize`, `-dontshrink` and `-dontobfuscate` tell ProGuard which steps of its workflow the developer wishes to omit. From all the kinds of rules that exist, only two are relevant for the work presented in this bachelor thesis. We present each in detail in the following sections.

Keep Rules

Keep rules are the first kind of rule we are interested in. They play two roles in the configuration of ProGuard. Firstly, they can be used to point to the entry points of the application so that ProGuard. Secondly, they can be used to inform ProGuard of all the sections of the code the user wishes to remain intact, protecting them from the effects of ProGuard [17].

Their syntax consists of a rule type that defines the kind of protection applied and a class specification that defines the code being protected. They are redacted in two segments as follows: `-keepRuleType classSpecification`. Each of the segments of keep rules is explained in the following sections.

Class Specification

In order for a developer to specify to ProGuard which parts of his code are targeted by a rule, the developer must design class specifications that target said sections of code. The general syntax of class specifications, as presented in the ProGuard manual [17], are as follows:

```
[@annotationtype][1] [[!]public|final|abstract|@...][2]  [!]interface|class|enum[3]
      classname[4]  [extends|implements [@annotationtype] classname][5]
[{[6]
      [@annotationtype]
      [[!]public|private|protected|static|volatile|transient ...]
      <fields> | (fieldtype fieldname [= values]);

      [@annotationtype]
      [[!]public|private|protected|static|synchronized|native|abstract|...]
      <methods> | <inti> (argumenttype,...) | methodname(argumenttype,...) |
                  (returntype methodname(argumenttype,...) [return values]);
}]
```

Figure 2.2: Class Specification Syntax

This definition may seem a little daunting at first sight, but class specifications are usually less complex than their definitions allow them to be. Before we explain this syntax, we must point out that any element inside brackets is optional for a class specification. Also, when elements are separated by a `|` it means that any of these elements are an alternative for this location. Finally, we can use the `!` symbol as negation. We explain each of the elements pointed at on ***figure 2.2***:

1. First, one can optionally add an annotation reference when specifying classes. This means that the class specification will only target classes which are marked with said annotation. Annotations are declared with a `@` symbol at the beginning followed by the package name of the annotation.

   Example If we wanted to target all classes marked with the `@Keep` annotation offered by Android, we would add `@android.support.annotation.Keep` in this location.

2. Then, we can opt to specify the declaration modifiers a class must posses to be targeted. This can be any of the modifiers offered by Java: public, final, abstract, etc. As Java lets us use annotations in the modifiers, this can also be specified here.

   Example To specify all public classes, we could write `public` on this location. On the other hand, if we wished to specify all non-public classes, we can opt to negate the modifier like this: `!public`.

3. Next, we must tell ProGuard if the specified code is inside a class, an enum, or an interface. Again, we can negate these declarations to target all but the negated type.

4. Following this, we make the actual reference to the package name of the code we wish to target. We call this the **class reference**. In this declaration, we may use wildcards to match with more than one package name. The offered wildcards are: `*` and `**`. The former matches with any character minus the package separator "." while the latter matches with any character including the package separator.

   Example If we wished to target an interface named "Listeners" which is located in the `com/app/listen` directory of the package structure, we could use the following declaration: `com.app.listen.Listeners`.

   The `*` wildcard can be used anywhere to define a pattern to be matched. For example, to match any name with the word Data on it in the `com.application.model` package location, we use `com.application.model.*Data*`. Alternatively, it can be used alone at the end of a class reference to specify all elements inside a location of the package structure. For example, `com.application.model.*` will specify everything inside the `com.application.model` directory.

   The `**` wildcard can be used at the end of a class reference to target all elements contained in a directory and all sub-directories of the package structure. For example, `com.application.model.**` will specify everything inside the `com.application.model` directory and all its sub-directories within the package structure. Alternatively, it

can be used in the middle of a class reference to target all elements in a directory, and all its sub-directories that posses a defined suffix in their package names. For example, `com.application.model.**.foo.bar` will target everything inside the `com.application.model` directory, and all its sub-directories, whose package name has the suffix `foo.bar`.

5. Optionally, we can decide to specify to specify classes which are either extending another class or implementing an interface. Here we reference the implemented or extended element in the same way we reference the primary element.

   Example `com.app.listen.EventListener extends com.app.listen.Listeners` will specify any class that has the package name `com.app.listen.EventListener` and extends the `com.app.listen.Listeners` class.

6. Next, we can choose to specify members of the class we are referencing. These members can be fields or methods. Their specification syntax is similar to classes, one can choose to specify annotation and declaration modifiers. To specify all fields, we can use the `<fields>` declaration. To specify a specific field, we must declare its type and name (optionally, we can define its value too).

   For methods we have four options. First, we could specify all methods using `<methods>`. Secondly, we may specify all initialization methods with the `<init> (argumenttype,...)` declaration. Here we must also declare the types of the arguments taken by the methods we wish to specify. Thirdly, we can specify methods by declaring their name and argument types, `methodname(argumenttype,...)`. Lastly, we can specify methods by specifying their returning values, `returntype methodname(argumenttype,...)`, here we must declare the type of the value returned and optionally specify the return declaration.

Keep Rule Types

Keep rules can have one of six total rule types, all of which offer different kinds of protection [17]. These are presented in ***table 2.1***.

Table 2.1: Keep Rule Types.

Keep	**From Being Removed or Renamed.**	**From Being Renamed.**
Classes and Class Members	-keep	-keepnames
Class Members Only	-keepclassmembers	-keepclassmembersnames
Classes and Class Members if Class Members Are Present	-keepclasseswithmembers	-keepclasseswithmembersnames

Here we can see that we have two groups of rules, those which protect from shrinking and obfuscation, and those which only offer protection from obfuscation. We can also observe

that some rules protect the entire class, while some only target the members of the class. Moreover, we may choose to protect classes only if they have certain members present.

Example To get a better understanding of the syntax of keep rules we present the following examples:

- `-keep public class com.example.MyMain`

  Protects the public class `MyMain`, found in the `com.example` package location, from obfuscation and shrinking.

- `-keepclassmembersnames public class com.example.MyMain {`
  `public void main(java.lang.String[]); }`

  Protects from obfuscation all public and void methods called "main" that take a list of strings as an argument and are members of the public class `MyMain`, found in the `com.example` package location.

- `-keepclasseswithmembers !abstract class ** implements`
  `@android.support.annotation.Keep **.*Data* {`
  `java.lang.String[] *Name*;`
  `*(java.lang.String[]);}`

  Protects from obfuscation and shrinking any class that is not abstract if, and only if:

  – Implements any interface in any directory that has ''`Data`'' in its name and is marked with the `@android.support.annotation.Keep` annotation.

  – Has a field which is a list of strings and has ''`Name`'' in its name.

  – Has any method whose only argument is a list of strings.

2.2.3. Warning Rules

The other kind of rules we are interested in are those that address warnings. Before explaining their syntax and function, we must first explain what a warning is.

Warnings

Warnings are error messages raised during the analysis of the code ProGuard carries out. They inform the user about possible problems in the configuration of ProGuard or in the source-code that will result in an error later on when running the application.

One of the probable causes of warnings are conflicting rules in ProGuard configuration files, like a keep rule protecting the name of a class while an `applymapping` rule is telling ProGuard the specific name to which the class has to be renamed. Other, more common reason warnings are generated is when referencing a compile-time only dependency [10]. For

example, the `javax.annotation.Nullable` annotation. If used in the code, ProGuard will not find this reference on the application's build path, and so it will raise a "reference not fund" warning.

Warnings raised by ProGuard will actually halt the building of an application. Therefore, all warnings must be addressed by either solving the error that generated it or including a `-dontwarn` rule in the ***build.gradle*** file of the project.

Rules for Warnings

Depending on the reason for their generation, warnings can be ignored to allow for the building process to run without being halted.

For example, we know warnings for provided compile-time dependencies can be ignored, as they are a result of ProGuard not finding something in the source-code when in reality it is provided by Java. This type of warnings can be ignored with a `-dontwarn classReference` rule. This **class reference** has the same syntax as the ones found inside **class specifications**. For example, the `-dontwarn javax.annotation.Nullable` will ignore any warnings for references to this annotation.

Since ProGuard may raise thousands of warnings, one may be tempted to ignore all of them using the `-ignorewarnings` rule. This is not advised since some warnings may inform the user about problems that will prevent the application from working correctly [10].

2.2.4. Benefits & Usage of ProGuard

The benefits that stem from code shrinking and optimization are easily understood. Having a faster application with a smaller file size just by editing its source-code, but not how it functions, is of great help to any developer. On the other hand, benefits taken from code obfuscation are less obvious but rather important.

The main advantage gained from code obfuscation is a reduced legibility of the applications source-code, which protects the application against being reversed engineered or cloned. A quick Google search will reveal that there is no shortage of applications and services dedicated to the decompilation of APK files, all of which ultimately reveal their source-code. This is a major issue faced by the Android development community, as over 13% of all available Android applications are clones of legitimate apps [9].

While these benefits may be seen as too good to pass on, the reality is that 75% of Android applications do not have ProGuard enabled [6]. This seems odd for a tool integrated to the Android Studio ecosystem and whose use is recommended in its documentation.

This reluctance by developers is not due to lack of awareness. As a study of obfuscation use in Google Play revealed, nearly 75% of android developers know about ProGuard, but 55% of them decided not to enable it and 35% had the intention to enable it but desisted because of the difficulties in implementing ProGuard [6]. These difficulties are part of the problem addressed in this bachelor thesis.

2.3. Related Work

When researching the current situation of this problem, to the best of our knowledge, no proposals or solutions were found to address this specific challenge. The only case of study in this area that could be found is the one carried out by Hecht et al [7].

They focused their solution on the joint use of crowd-sourcing and machine learning. Searching within F-Droid, a repository of 2038 free and open-source Android apps, 352 applications were found to have good practices in their obfuscation rules.

Three different learning models (SVM, neural network and random forest) were then trained with the source-code and rules of these applications, labeling each existing class in the source-code as maintained or not maintained post obfuscation.

With this method, the machine learning aspect was intended to recognize patterns within the source-codes and their relationship to the rules included in their ProGuard configuration files. Afterwards, classes from other applications were presented to the learning models in an attempt to find the previously documented patterns and writing the respective obfuscation rules deemed necessary.

Although this approach seems adequate, the results obtained were inconclusive. When testing the method with applications that had already been obfuscated, to see if it was possible to recreate the same rules already used, satisfactory keep rules were generated for 29% of applications. Then, it was tested with applications without active obfuscation and that crashed when trying to run after having been obfuscated. In this case, it was not possible to generate rules that would allow execution in any of the applications.

Additionally, this approach focused on the generation of rules for the source-code of an application, leaving out the rules needed by dependencies.

In conclusion, little work has been done on this topic. This means there is great potential for development in this area, specially in regards to the generation of rules for dependencies.

Chapter 3

Problem

As exposed in the ***State of the Art*** chapter, most Android developers do not enable ProGuard in their applications. This might be due to all the difficulties that the developer must overcome in order to implement ProGuard.

Indeed, the effects that ProGuard has on code come in direct conflict with some common practices of Android and Java development. These result in compile-time and run-time errors and bugs that stem directly from the use of ProGuard. Additionally, the warnings raised by the code analysis of ProGuard will prevent an application from being built, so they must all be addressed and solved. Therefore, the main problem addressed by this bachelor thesis is how to solve these problems, thus eliminating the bottleneck that keeps developers from enabling ProGuard.

In the following chapter we present the practices commonly used by developers that conflict with the shrinking, obfuscation and optimization of the code, the problems associated with warnings, and why these difficulties would turn a developer away from ProGuard.

3.1. Inherent Conflicts with Common Practices

While the utilization of ProGuard in applications brings great benefits to the developer, it can also cause headaches during its implementation. This is mainly because the shrinking of code and renaming of classes and members come in direct conflict with many common practices prevalent in many Android applications.

Reflection and introspection are responsible for these conflicts [17]. Both are practices in which the code refers to itself at run-time. The problem arises when ProGuard removes or renames the elements of the code that are being referenced. This will result in run-time errors or bugs as the reflective/introspective self-referencing aspects of the code will not be able to find what they are searching for. To ground these concepts, we present some common

reflective practices and explain why they would fail if compiled with ProGuard enabled.

Data Classes: The use of data classes in Java is a common practice. They are used to model data within the application in a way that is easier to understand and use. The problem is that these classes are usually loaded or serialized into another medium, like databases or JSON files, to store the data [14]. Therefore, conflicts arise as these serializations rely on reflection. Indeed, they inspect the names of the fields of the data class when representing the data. If these names are later changed this process will fail as the new coded names will not be found.

Example We present how a data class that is serialized into a JSON file conflicts with ProGuard.

Code 3.1: Example of a Java Class.

```
public class Example {
    private String example_name;
    private String description;
}
```

If we had an instance of the data class shown on ***code 3.1*** that holds the example name "Data Class" and the description "A data class example.", serializing it into a JSON file would result in something like ***code 3.2***. This JSON file is naturally saved in disk for later use.

Code 3.2: Example of expected JSON.

```
{
  "example_name" : "Data Class",
  "description" : "A data class example.",
}
```

But if the fields of this data class were renamed by ProGuard to "a" and "b" , the resulting JSON will also contain these new names. Therefore, any processes that use this JSON file assuming the original names of the fields will fail.

JNI: Calling Java code from the native side will conflict with ProGuard. While ProGuard provides a rule for protecting classes with methods implemented on the native side, it has no way of detecting whether Java code is being called from the native side. This is because ProGuard is not capable of reading C/C++ files [14]. Indeed, with the help of the JNI library, native code may be reflectively referencing classes and methods by name.

Example The JNI method `FindClass(classLocation)` call will not work if the class in question has been renamed, as the string contained in the `classLocation` variable will not match any class. It is also possible that the referenced class is only used on the native side. If this is the case, ProGuard will remove this class as it will consider it dead code. This will lead to an error too.

Resource Loading from APK: Plain Java provides the option to load resources from the JAR file or in the case of Android apps, the APK file. These resources are classified under the classes own package names. So if the class name is changed, trying to load resources will fail.

Annotations: As annotations are completely removed when shrinking the source-code, any reliance on the active use of annotations at run-time will fail if these are removed.

Another concrete example can be seen in ***codes 2.3 and 2.3***. If in another part of the code there were an instance of `getDeclaredMethod('aUsedMethod')`, there will be an error when running the application. Indeed, after obfuscation `aUsedMethod()` is renamed to `c()`. Consequently, when the method `getDeclaredMethod()` looks for a method called *"aUsedMethod"*, it will not find it and the application will fail to run, generating a *Method Not Found* error.

This kind of conflicts are difficult to locate without deep knowledge of the source-code. In its documentation, ProGuard states: "Obfuscating code that performs a lot of reflection may require trial and error, especially without the necessary information about the internals of the code." Additionally, warnings will not guide the developer through these conflicts, as ProGuard admits its analysis of the code is not capable of detecting reflection accurately [17].

Keep Rules

Keep rules can be used to resolve these kinds of conflicts. They can be employed to instruct ProGuard to leave intact any class or class member which is involved in a reflective functionality.

For example, to get rid of the *Method Not Found* error explained earlier, we could use the rule `-keepclassmembers class location.of.package.AClass` to maintain the whole class protected from the effects of ProGuard. If we want to target only the necessary method, we can use `-keepclassmembers class location.of.package.AClass {private String aUsedMethod();}`.

It is intuitive that the benefits of ProGuard are maximized when letting it affect as much code as possible without generating errors. Therefore, as a rule of thumb, we want keep rules to be as specific as possible, referencing just the specific parts of the code that have conflicts with obfuscation. Protecting more code than needed, while not resulting in any kind of error, will hinder ProGuard's desired effects on the source-code.

3.2. Warnings

The warnings raised by the code analysis ProGuard carries out will prevent the building of an application. Therefore, all warnings must too be addressed by the user. This can be

done by either solving the problem a warning points to or by recognizing that the warning is pointing out a non-issue and ignoring it with a `-dontwarn` rule.

The problem is that ProGuard may raise thousands of warnings for an application. The process of addressing all of them can be very time-consuming.

3.3. The User's Responsibility and Libraries

We can see that the implementation of obfuscations comes with a fair set of difficulties which have to be solved through the redaction of rules. Unfortunately for developers, the task of addressing the entirety of the conflicts brought upon by obfuscation lies solely on themselves.

Additionally, all of the previously exposed hardships are amplified when third party libraries are included as dependencies for applications. Certainly, using external libraries forces the developer to address bugs and warnings generated by code with which they are not familiar. And if we take into account that many libraries are used to outsource the act of serializing data (like GSON) or handling external reflective requests (like OKHTTP), we can see that this practice presents a prevalent hurdle for the implementation of ProGuard.

While some libraries include a set of standard rules to include in the configuration of ProGuard, some even including them automatically by including a *consumerProguardRules* declaration in their ***build.gralde*** file, its more common to find said rules in the libraries documentation if at all [11].

As mentioned previously, we want ProGuard to process as much of the source-code as possible to maximize its benefits. Therefore, in order for a developer to be able to use ProGuard efficiently, they must be able to write keep rules which are specific to the conflicting parts of the code and address every warning individually. This entails counting with the expertise and time to be able to carry out this process in an efficient manner. This can be an exhaustive process, especially in large applications with a heavy use of reflection, and may be the reason so many developers are discouraged when attempting to enable ProGuard.

3.4. Removing the Bottleneck

If a solution capable of generating the rules needed to solve these problems existed, it would allow for many developers to be able to enjoy the benefits brought upon by the implementation of ProGuard in their Android projects. During this bachelor thesis, said solution is the one we attempt to generate.

Requirements

In order for a solution to solve the presented problem completely, it must meet the following requirements:

- **Detection of third party libraries**: The solution must be able to detect which third party libraries are declared as dependencies of an application.
- **Generate rules for the dependencies**: Once detected, the solution must be able to generate a ProGuard configuration file containing all rules needed by the dependencies of an application.
- **Detect conflicts with ProGuard:** The solution must be able to detect which parts of the source-code of an Android project come in conflict with the functionalities of ProGuard.
- **Generate rules for the detected Conflicts:** Once detected, the solution must be able to generate a ProGuard configuration file containing all the rules needed to solve the conflicts in its source-code.
- **Address warnings:** The solution must address the warnings generated for both the dependencies of an application and its source-code.
- **Be easily used by an Android developer:** The solution must be easy to use for someone who has the capability of developing an Android application.

Additionally, to be considered a quality solution, the generated rules must allow for a maximum acting coverage of ProGuard on the source-code. In other words, they must only act on the parts of the code that are strictly necessary.

Chapter 4

Solution

In this chapter we present the solutions developed to address the presented problem. Firstly, we present a Python library consisting of classes meant to model key elements of an Android project. Additionally, we show how we use these classes to parse a repository of Android applications and then use the information held in their fields to conduct a study of the composition of the applications. Then we discuss the reasons that led us to implement a database for our solutions long with the design of said database. Finally, we present two solutions to our problem. The first solution focuses on the generation of rules for dependencies while the second focuses on the generation of rules for specific conflicts within the source-code of an application.

4.1. Modeling and Studying the F-Droid Repository

Before any development of a possible solution is undertaken, we first had to learn about how many Android applications use ProGuard, how many rules they use, how many of these rules are targeting dependencies and what type of problems they address. In order to gain a better insight into the relationship between Android applications and ProGuard, we conducted a study of the F-Droid repository.

F-Droid is a repository of free and open source software on the Android platform. On its website one can find a myriad of Android applications ready to be downloaded. It was selected as our data set for study as it not only offers the applications APK for download, but additionally, one can choose to download the source tarball. This compressed file contains the applications original Android Studio project files, the ideal resources we need to carry out a comprehensive study. On top of this, it is the largest data-set of open source apps and commonly used in studies related to Android applications.

4.1.1. Scrapping F-Droid

In order to obtain the desired files in a timely manner, we decided to make use of the scripting capabilities for web scrapping the Python language possesses. We used the *"request"* library to navigate the F-Droid website in tandem with the *"BeautifulSoup"* library to parse each of the pages HTML code. After a brief inspection of the structure of the F-Droid website and HTML code patterns of its pages, we were able to design a script that thoroughly traversed through the applications contained in the repository, locating the HTML tags containing the download links for their source tarballs and saving them into the local disk in an orderly fashion.

With this script, we obtained the source Android Studio projects of 2,955 applications. We were then required to transform this data into a format which would facilitate our study of these applications. For this we decided to continue using Python as our main development language. This was due to our familiarity with the Python programming language along with its versatility. Indeed, the support Python provides for object-oriented programming, reading and writing files, scripting and even functional programming were all valuable tools used in the entirety of the following work.

4.1.2. Python Meta-Model

The first step taken to study the downloaded applications was to extract the desired information from the F-Droid repository and store it in a way that facilitates their study. We are looking to learn if an application enables ProGuard or not, what dependencies it declares, which rules it is using and what are the contents and location of its source-code classes. For this we designed a meta-model using Python classes in an object-oriented programming approach.

For this model, four data classes were designed. They model the F-Droid repository, the Android Studio projects (Applications), their Java (or Kotlin) classes and their associated ProGuard configuration files respectively. These classes will store all of the previously specified data for us to use in our study and later work. The cardinality and ordinality relationships of these classes can be seen in ***figure 4.1***.

In order to obtain the needed information to fill the data classes and conduct our study, all classes contain a respective analyser class. These are in charge of the file parsing and data extraction required to obtain the desired information. All file parsing is done by making use of the file reading capabilities of Python along with regular expressions designed to match the specific locations where the data of interest is declared. The regular expression matching operations were provided by the *"Lib/re.py"* Python module.

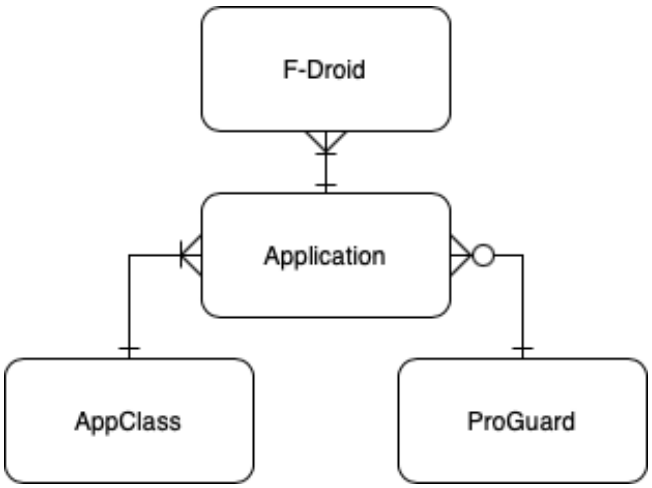


Figure 4.1: Relationships Between Classes.

F-Droid Class

These class models the F-Droid repository saved in the disk. It is initiated by providing the path to the location where the repository is stored. The object then iterates through all the directories contents (sub-directories containing the applications Android Studio projects) creating an Application class instance for each of them. It then stores these Application class instances in a field as a list.

It also sorts each application in different lists depending on whether they are obfuscated or not, thereby facilitating a number of useful class methods such as: getting all obfuscated or unobfuscated apps, getting a random obfuscated/unobfuscated app and getting the total number of obfuscated/unobfuscated apps among others. The UML diagram for this class can be seen on ***figure 4.2***.

This class is associated with its respective analyser class: FDroidAnalyser. This is the main class we use when exploring the composition of the F-Droid repository. It generates each of the desired graphs to represent this data. Its UML class diagram can be seen in ***figure 4.3*** .

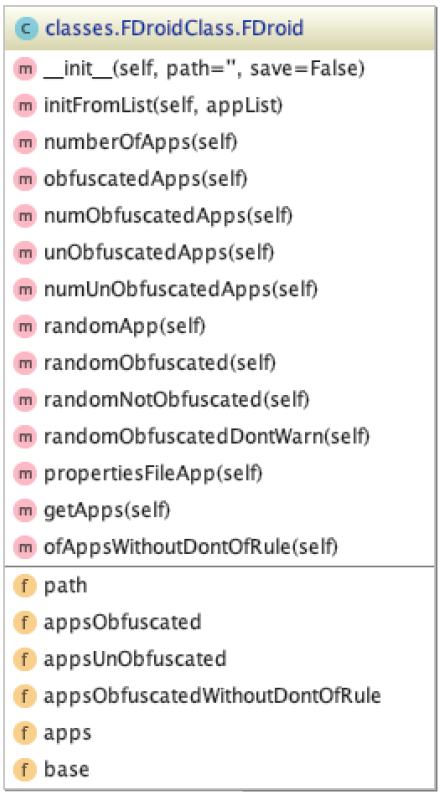


Figure 4.2: F-Droid Class UML Diagram.

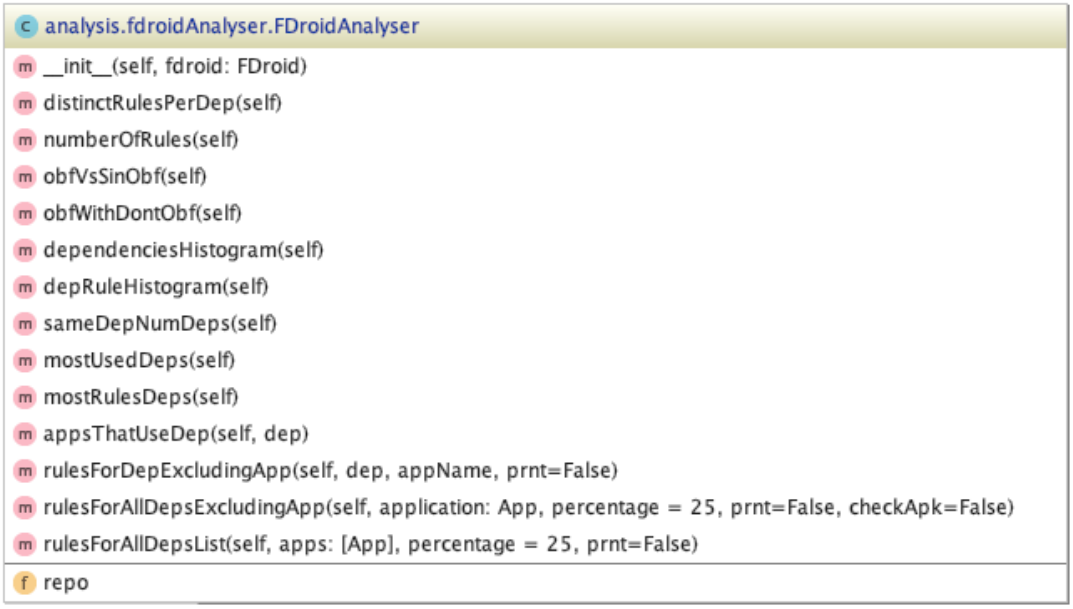


Figure 4.3: FDroidAnalyser Class UML Diagram.

Application Class

Models each specific Android Studio project of the F-Droid applications and stores relevant information needed for the study and posterior solutions. This includes its name, location in the repository, location of build scripts, location of ProGuard configurations files, if it has obfuscations enabled or not, its package structure and declared dependencies.

The initialization of this class is aided by the ***AppAnalyser*** class. This class is delegated the task of reading and analysing the key configuration files of an application in order to extract the needed data from them. Its first job is to determine whether the application has ProGuard enabled or not. This is achieved by inspecting the ***build.gradle*** file (or the ***project.properties*** file in some cases) contained in the project. This is the file in charge of configuring the build process of applications and where obfuscation is enabled with the `minifyEnabled` declaration. A standard declaration can be seen in ***Code 4.1***.

Code 4.1: Zapp App ***build.gradle*** File (extract).

```
buildTypes {
  debug {
    shrinkResources false
  }
  release {
    shrinkResources true
    minifyEnabled true
    proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
  }
}
```

The AppAnalyser method `isAppObfuscatedG()` reads this file looking for the `buildTypes` declaration and the `release` declaration within it, as we want to detect those apps that enable ProGuard in their release builds. The `minifyEnabled` can be declared true in different ways (t, true or enabled indifferent to capitalization), so once we know we are inside the release specifications, we use a regular expression designed to match with any of the possible variations (accounting for inconsistent white-spaces throughout different files). If our regular expression, `minifyEnabled\s*=*\s*[tTeE]`, detects a match in the lines inside the release specifications, we know ProGuard is enabled. In turn, if we reach the end of the release build declarations without any match, ProGuard is not enabled on this application.

In the former case, we need to learn the location of the ProGuard configuration files containing the rules used by the application. As seen in ***Code 4.1***, these locations are specified within the proguardFiles declaration. This can be declared in various different ways too, from simply listing them in a coma separated sentence, to declaring the directory where a group of files is stored. The former is easily parsed by a regular expression that will match with every string that is surrounded by quotation marks and has a file extension at the end, `'(.*?\..*?)'`. If, on the other hand, we detect the declaration points to a directory where the configuration is stored, we use a different regular expression, `dir: '(.*?)'`, to match with the directory location and then use Python's own `os.listdir()` function to list every element in said

location. The `isAppObfuscatedG()` method then returns a list of all the declared ProGuard configuration file locations (with an empty list signaling obfuscation is not enabled) which is saved temporarily in the Application class object. If not enabled, the Application class initialization is finished as there is nothing more we can learn about its relationship with ProGuard. But if enabled, we proceed to a deeper inspection of the application.

As we know, third party libraries contribute to the difficulties of implementing obfuscation. For this reason, we are interested in learning which third party libraries are being used in applications that enable ProGuard. Fortunately, in order to use libraries in an Android application, the user must declare them as dependencies in its ***build.gradle*** file. The `extractDependencies()` method provided by the AppAnalyser class is then used to read the Gradle file in search of declared dependencies. As the declarations of dependencies have multiple variations, which can also change depending on the version of Gradle being used by the application, we will not detail each of the different cases. As done before, when detecting the location of ProGuard configuration files, we make use of the file reading capabilities of Python to detect the `dependencies { ... }` declaration and the type of dependency declarations encountered within. We then use a number of carefully designed regular expressions, each one targeting a different kind of dependency declaration, to match with the library name. These are returned as a list of dependency names and saved in the Application class object.

Finally, we desire to know the location of the source-code files within the project. To do this we explore the standard directory in which Android Studio stores the release code: *"src/main/java"*. Here we look for every Java or Kotlin classes we can find by looking at file extensions. Next, we use their locations to initialize AppClass objects, which model the Java or Kotlin classes of an application, and store them in a list in their respective Application class object. The UML diagrams for both the Application and AppAnalyser class are shown in ***figures 4.4 and 4.5***. The Application class methods are omitted to avoid an oversized image, a full UML diagram can be appreciated in ***Annex A.1***.

AppClass Class

Models each source-code class belonging to the application. As Android applications can contain both Java and Kotlin classes, two sub-classes are created to handle the differences between these two types of classes during initialization: the JavaClass and KotlinClass subclasses.

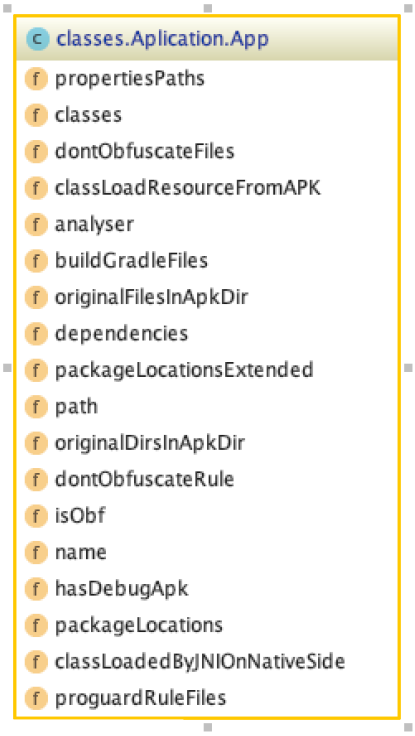


Figure 4.4: Application Class UML Diagram.

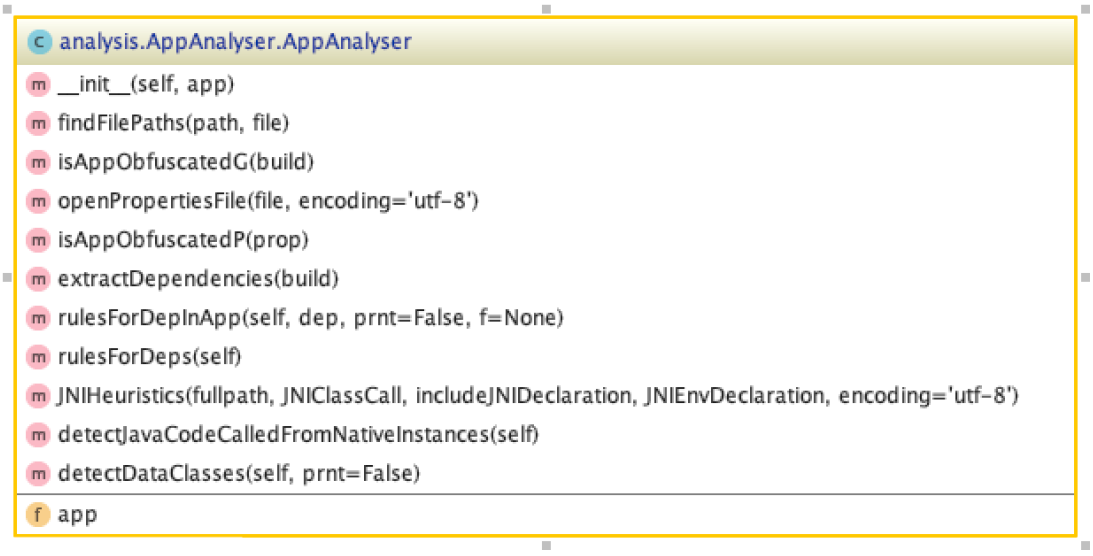


Figure 4.5: AppAnalyser Class UML Diagram.

This class stores the name, path and imports of the applications Java/Kotlin class files. The imports are extracted by the AppClassAnalyser class, where, depending on whether it is written in Java or Kotlin, different regular expressions are implemented to match with the import declarations considering the slight differences in the two. For this, we also need to adapt how the *"re"* module carries out the pattern matching. This is done by using flags

contained in the module (for example: `re.MULTILINE` indicates a string containing the "\n" character). To do this in an efficient way, we implement a functional programming trick. We design a Lambda function that takes the *"re"* module as an argument and returns the desired flags, and pass it as an argument when analysing the source-code class file. This can be appreciated in the ***code 4.2*** extract. All elements being imported are then saved in the AppClass object.

Code 4.2: Functional Programming Trickery for Java Classes.

```
# When calling the analyse method
self.analyser.analyseClass(self, path, '^import(.*?);', lambda x: x.MULTILINE | x.
 ↪ DOTALL, app)

# In the Analyse Function
def analyseClass(appClass, path, imprt, flags, app):
    imports = re.findall(imprt, file, flags(re))
```

We are also interested in registering the Applications package structure, this is the general structure of the applications directories holding the source-code. To do this, when initializing an AppClass object, we detect its location within the package structure by inspecting its path. As the package structure begins in the *"src/main/java/"* directory, we just have to split the class's path in half where this sub-string is found and take the latter part of it. We then save this package location in the Application class object respective to the AppClass. The UML diagrams for these classes can be seen in ***figure 4.6***.

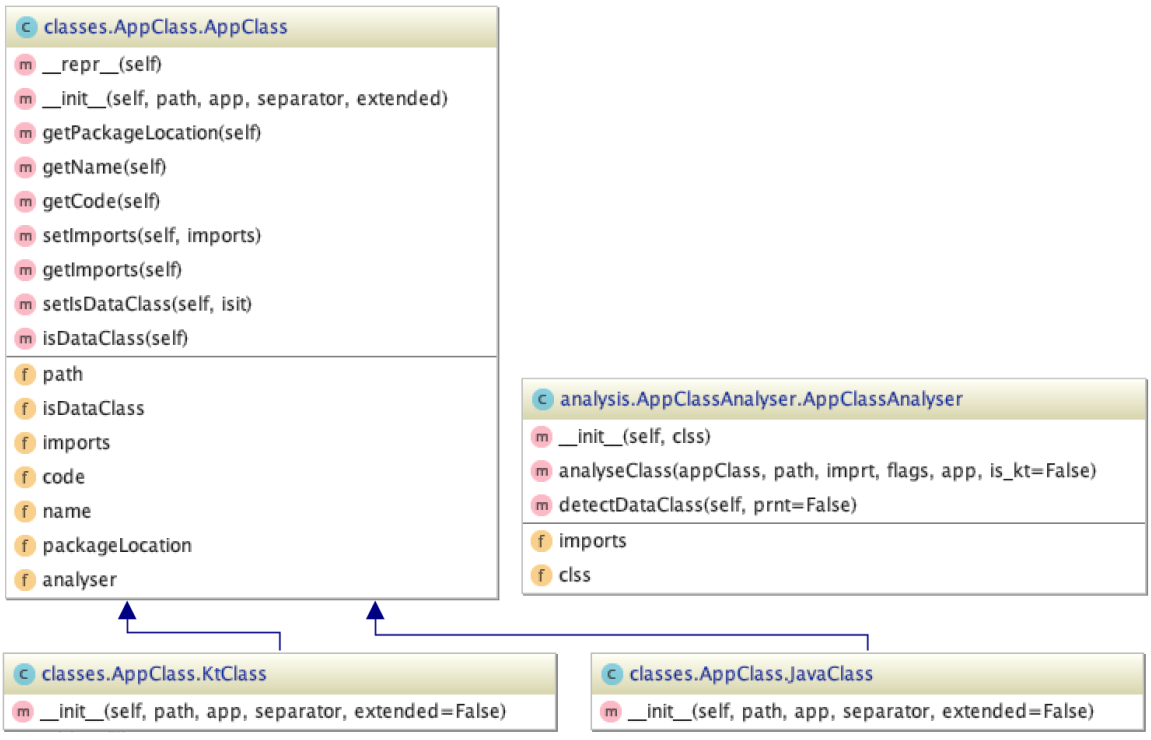


Figure 4.6: AppClass and AppClassAnalyser Class UML Diagrams.

To store the package locations in the Application class, we use the pure Python implementation of the radix tree data structure provided by Google: pygtrie. In this implementation, we use the classes package locations as keys and the classes file name as values. An example of a stored package structure can be seen in ***figure 4.7***.

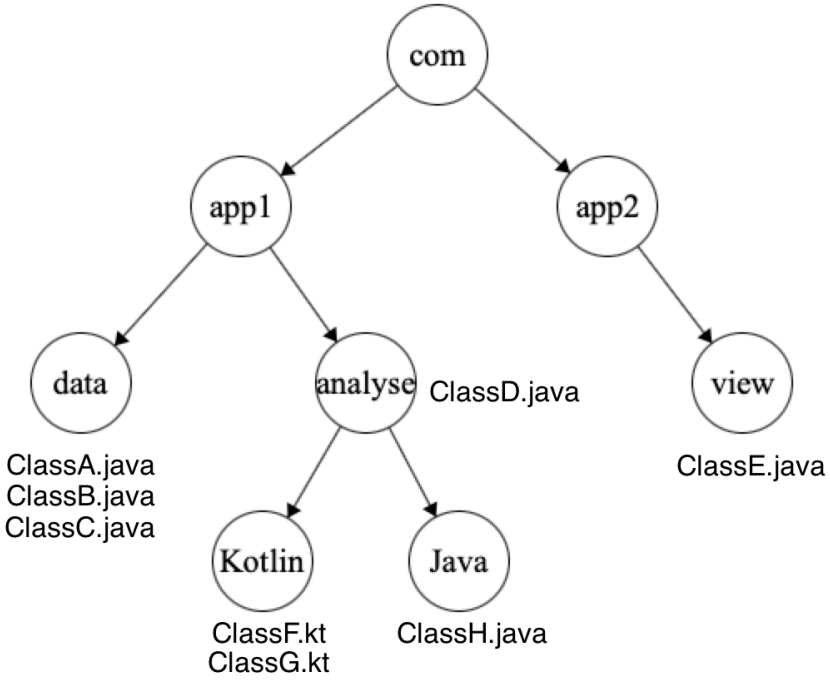


Figure 4.7: Visualization of Radix Tree Containing the Package Structure of an Application.

This decision was made because at some point we are going to use the package structure of an application to determine if a rule is referencing classes in the source-code. If we stored them in a list, when faced with n directories containing classes, we would take $\mathcal{O}(nk)$ time to check if every rule is referencing a directory in the package structure, where $k =$ the length of the class reference contained inside the rule. Instead, the radix tree offers string insertion and lookup with $\mathcal{O}(k)$ time. As we are going to be parsing thousands of applications with their rules and classes, especially while validating our work, taking too much time in this process will result in a slow development and solution.

Other data structures like self-balancing Binary Search Trees and Hash Tables (Python dictionaries) were considered but discarded. BST has a slower search time of $\mathcal{O}(klog(n))$ and Hash Tables, while offering a search time of $\mathcal{O}(k)$, offers no advantages when dealing with keys that share prefixes, a common occurrence in package locations.

Additionally, this particular implementation offers some attractive methods, such as the ‘‘`has_subtrie(key)`’’ method. With it we can determine if the key argument has an existing prefix in the keys inside the tree. This will be useful when determining if rules that use wildcards are referencing a package location.

ProGuard Class

Models the ProGuard configuration files which hold the rules used by an application. We use the locations of these files, previously discovered when building the Application object, to initialize these objects. Initially, we store the name and path of the configuration files in their fields. We then initialize an instance of the ProGuardAnalyser class to inspect the contents of each file. ***Code 4.3*** shows some of the rules contained in the ProGuard configuration file of the *"AELF"* application.

Code 4.3: Initial Contents of a ProGuard Configuration File.

```
1  -keep enum org.greenrobot.eventbus.ThreadMode { *; }
2
3  # Only required if you use AsyncExecutor
4  -keepclassmembers class * {
5      @org.greenrobot.eventbus.Subscribe <methods>;
6  }
```

This analyser class first reads the content of the file and removes all comments present on it. It does this by taking advantage of the `re.sub(pattern, substitute, string)` method, which substitutes the matching sections of a string with another desired string. Using the `(?m)^\s*#.*\n?` pattern to match with comment lines in the entirety of the file, we replace all of them with an empty string as seen in ***code 4.4***.

Code 4.4: ProGuard Configuration File with comments removed.

```
1  -keep enum org.greenrobot.eventbus.ThreadMode { *; }
2
3  -keepclassmembers class * {
4      @org.greenrobot.eventbus.Subscribe <methods>;
5  }
```

Then, we use the Python String `split()` method in tandem with the Python String `join()` method to eliminate any unnecessary white-spaces. The `split()` method transforms a string into a list, making the wherever any white-space is found, independent of the amount or kind (space, new-line, tab, etc.). When we join this list into a new string, using a single space as the item separator, we make sure all extracted rules will be standardized. This will help us avoid considering two rules different, even if they are the same, just because one developer used tabs and another used a group of spaces. The results are shown in ***code 4.5***. In these example rules should be on the same line separated by a single space, but are separated for the sake of clarity.

Code 4.5: ProGuard Configuration File with Standardized White-Spaces.

```
1  -keep enum org.greenrobot.eventbus.ThreadMode { *; }
2  -keepclassmembers class * { @org.greenrobot.eventbus.Subscribe <methods>; }
```

Next, as all rules are preceded by a hyphen, splitting this string wherever this character is present will return a list containing each individual rule in the configuration file. As some rules might contain multiple class references in their class specification, separated by commas, we separate these declarations in individual rules for each of these references. This will avoid considering two equivalent rules to be different just because one of them contains an extra class reference. The extracted rules are then saved in a field of the AppClass instance, and this instance is in turn saved in its respective Application object.

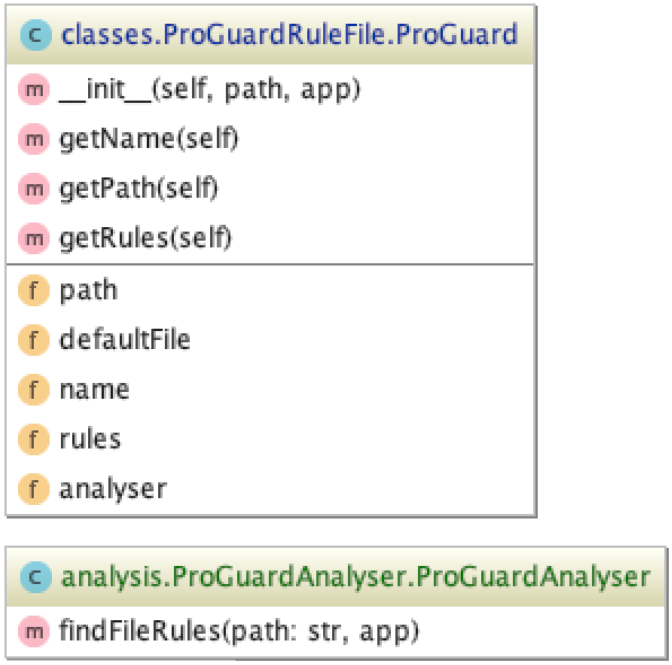
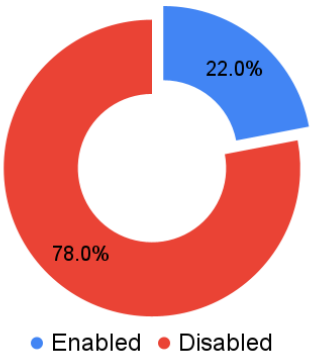


Figure 4.8: ProGuard and ProGuardAnalyser Class UML Diagrams.

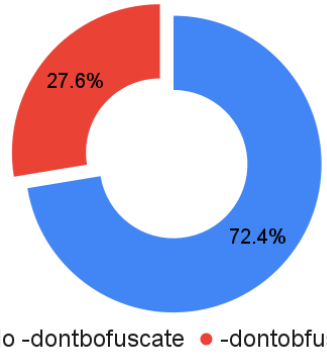
Once we have all of these classes designed and available, we are able to study the applications inside the F-Droid repository as to better understand their relationship with ProGuard and guide our next course of action.

4.1.3. Results of the Study

The next graphs are generated by our FDroidAnalyser class to achieve a better insight of the composition of F-Droid. First, we want to know the ratio of applications with obfuscation enabled vs not enabled. As we can see in ***figure 4.9.a***, the number of applications which use ProGuard is congruent with the previously exposed data. Barely a quarter of the repository, 649 of the total applications, enables ProGuard on their release build.



(a) Ratio of Apps with ProGuard Enabled.



(b) Presence of Obfuscation Supression

Figure 4.9: Use of ProGuard and Obfuscation..

But a closer look is needed to determine how many applications are making use of the entirety of ProGuard tools. Indeed, by inspecting the rules contained in the applications which do enable ProGuard, we find that many of them include the `-dontobfuscate` rule. This rule disables the obfuscation module of ProGuard, leaving only the shrinking and optimization modules. As seen on ***figure 4.9.b***, 28% of applications which enable ProGuard suppress the obfuscation module through this rule. This might indicate that the conflicts that arise from obfuscation are harder for the user to resolve, in contrast with shrinking and optimization, as a significant number of developers explicitly declared their intentions to do without it.

Next we intend to learn more about the specifics of each application and their rules. Fields of interest include the number of rules used, number of dependencies declared and the average number of rules that target a declared dependency. It is worth mentioning that in all the following histograms the greatest and sporadic values are gathered in the last bar for the sake of clarity.

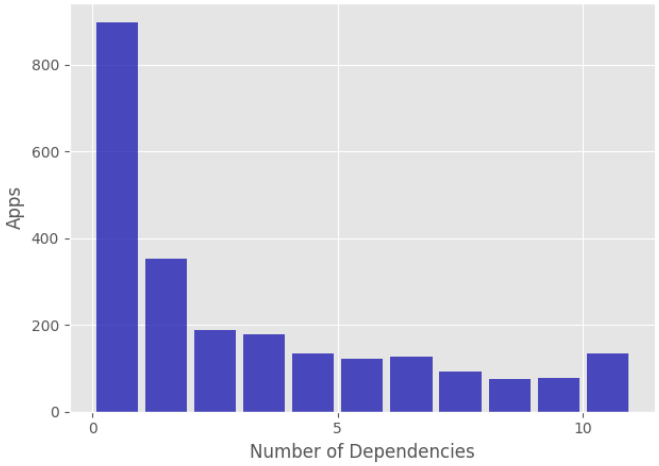


Figure 4.10: Dependencies per App

From ***figure 4.10*** we learn that all applications declare at least one dependency, with most declaring between one and five dependencies. On ***figure4.11*** we see the number of

rules used by applications.

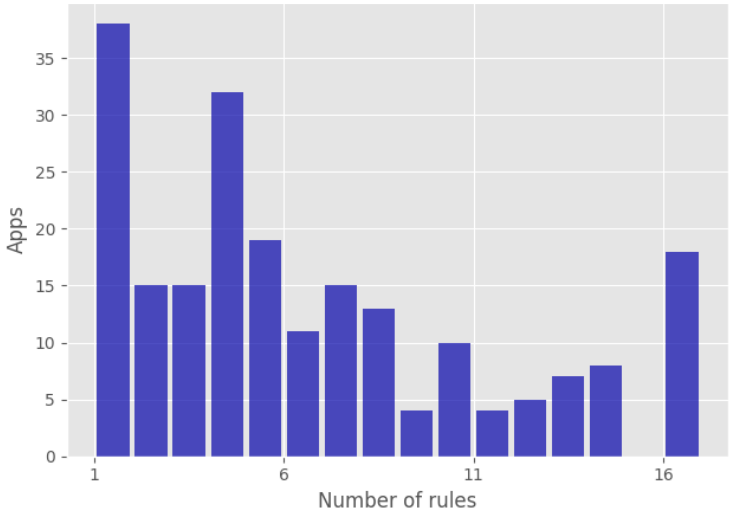


Figure 4.11: Rules per App.

Here we can observe that, excluding applications with no rules which have ProGuard disabled, all applications have at least one rule with most having between one and ten. Finally, on ***figure 4.12*** we explore the number of rules dedicated to specific dependencies included in the ProGuard configuration files of applications.

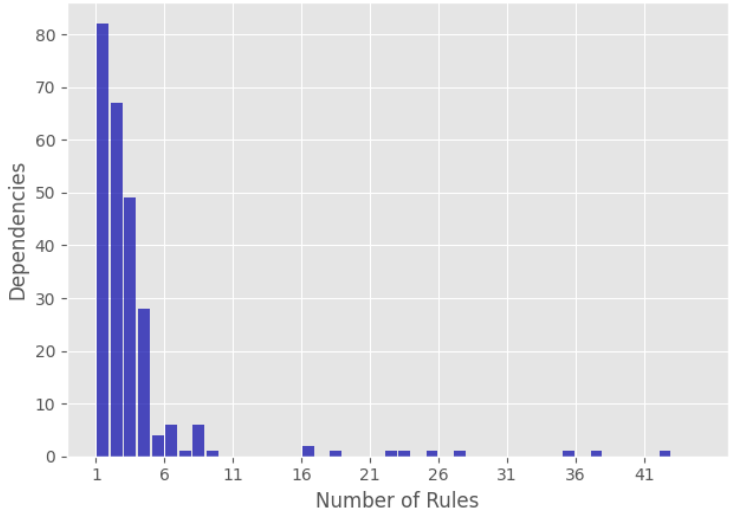


Figure 4.12: Rules per Dependencies of an App.

From this histogram we learn that most dependencies have between one and five rules dedicated to them. This, paired with the knowledge that most applications declare at least one dependency and have between one and ten rules if they enable ProGuard, indicates that rules dedicated to dependencies have a prevalent role in the total number of rules for applications.

We can also tell, by the number of dependencies exposed that the number of total dependencies declared is far lesser than the number of applications. This is a revealing piece of data that, paired with the knowledge that rules for dependencies conform an important number of all rules, leads us to make the following conjecture:

Conjecture 4.1 *Let A be an application that uses dependency d and r be the group of rules needed by A to be able to use the dependency d with ProGuard enabled. If R is the group of all rules used by other applications which use d with ProGuard enabled, we suspect that $r \subseteq R$. In other words, we should be able to find the rules needed by an application for a specific dependency within the rules of other applications that also use said dependency.*

With this data on hand, and looking to prove the veracity of our conjecture, we set out to develop a method which generates rules for the dependencies of an application by searching for relevant rules included by other applications that declared the same dependencies.

But before we advanced, a development hurdle needed to be addressed: The parsing of the repository takes a significant amount of time. Even after reordering the applications in the disk into different directories depending on the necessity to avoid parsing all applications every time (splitting them between obfuscated apps, unobfuscated apps and even a reduced mixed set for testing) this was a time-consuming task. A solution was therefore needed to avoid delays in development.

4.2. MySQL Database

A MySQL database was deemed necessary for two reasons. Firstly, the previously described model had to parse the applications every time it was initiated, which took 15 minutes to parse 650 obfuscated apps. This might not seem an exceedingly long wait, but this time begun pilling up while developing. As this model had to be loaded every time a test was run, it was clear a faster approach was needed. Secondly, this load time will only get worse if we add more applications to the repository, as might be done to grow our data-set and broaden our rule selecting possibilities.

Initially, a parallelism approach was thought of to tackle this inconvenience, but this was a solution whose effort in implementation begun outshining the benefit. Additionally, even when parsing multiple applications at the same time, adding new applications to the repository will result in a proportional increase to the load time.

An associated database brought multiple benefits. By using the indexed searching provided by MySQL and strategically designed queries, we are able to quickly obtain any data needed. With this implementation, we only need to load the full repository in an object when initially populating the database and when changes are made to the model. Along with this, a database allows us to scale the size of the repository without greatly affecting search times.

4.2.1. Database Tables

The tables needed to mirror the data needed from the python model can be seen in ***tables 4.1 through 4.4***:

Application:

Table 4.1: Application Database Table.

Attribute	Type	Description
Id	Int	Unique primary key.
Name	Varchar	Unique name of the app.
Path	Varchar	Path of the app.

Dependency:

Table 4.2: Dependency Database Table.

Attribute	Type	Description
Id	Int	Unique primary key.
Name	Varchar	Name of the dependency.
AppId	Int, Foreign Key	App to which the dependency belongs to.

Import:

Table 4.3: Import Database Table.

Attribute	Type	Description
Id	Int	Unique primary key.
Import	Varchar	Full import declaration.
AppId	Int, Foreign Key	App to which the import belongs to.

Rule:

Table 4.4: Rule Database Table.

Attribute	Type	Description
Id	Int	Unique primary key.
Rule	Varchar	The redacted rule.
AppId	Varchar	App to which the rule belongs to.

The entity-relationship diagram for this database can be seen in ***figure 4.13***.

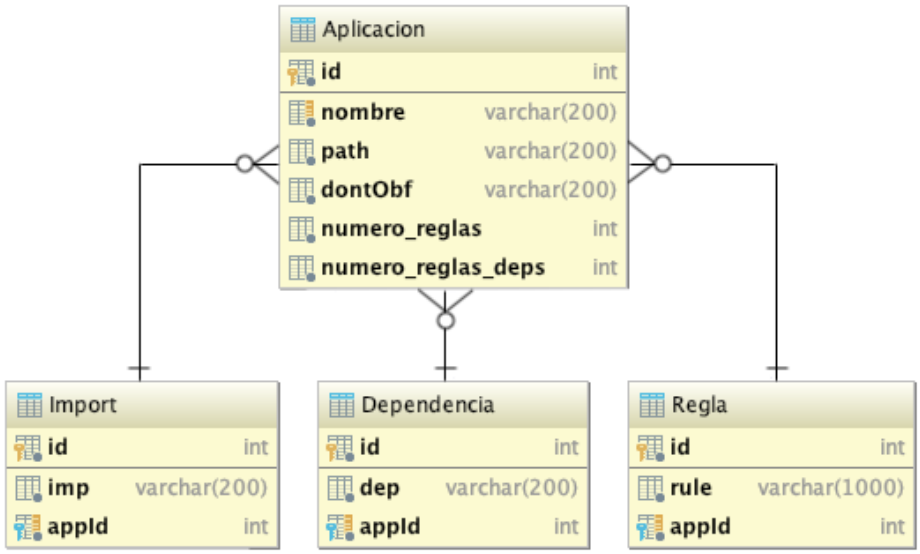


Figure 4.13: Entity-Relationship Diagram of the Database.

4.2.2. DBConnect Class

To handle the communication between the Python meta-model and the database, the DBConnect class was designed. This class uses the `mysql.connector` library to connect with the database. It also employs methods to create and execute queries meant to save the desired data in the respective tables. Its UML diagram can be seen in ***figure 4.14***.

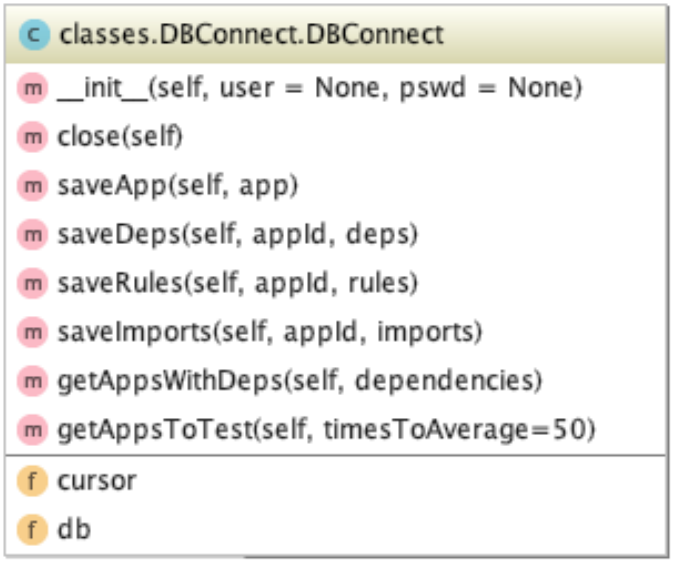


Figure 4.14: DBConnect Class UML Diagram.

When using the DBConnect class to obtain data, we wanted to make as little calls to the database as possible as making the connection to it is the most time-consuming aspect of using a database. Therefore, instead of creating a dedicated method for each of the objects of the meta-model, we design custom methods for each of the instances where data is needed. We can assure this way that the queries made are as efficient as possible.

For example, we create a method that receives a list of dependency names as an argument. This list is then used to build a nested query that will search for all applications which contain one of the dependencies, along with its associated rules and any of its imports which fetch classes or methods from the dependencies.

This also resulted in the need for creating the DataBaseAnalyser class. As we are obtaining data in a case-by-case manner instead of loading it into our previous model, this class is used to define the methods that will take the results of our queries as arguments. Its UML diagram can be seen in ***figure 4.15***.

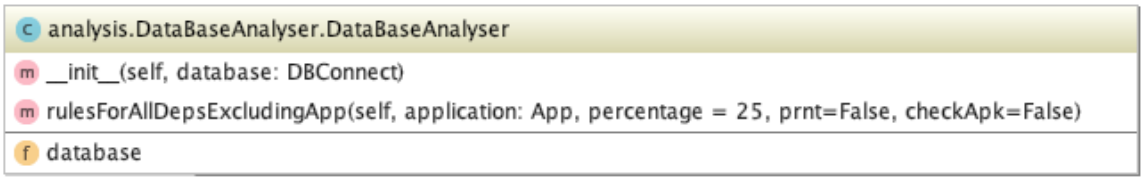


Figure 4.15: DataBaseAnalyser Class UML Diagram.

4.3. Dependency Rules Solution

With the meta-model, database, DBConnect and DataBaseAnalyser class ready, we have the necessary base tools to develop a solution for the dependency rules of an application. We decided to focus our work on this solution for two reasons. Firstly, the previously conducted study revealed that rules for dependencies conform a significant amount of the rules included by applications. Secondly, as the redaction of rules entails solid knowledge of the code, dependencies present a particularly challenging aspect of the implementation of ProGuard.

The solution works in two modules: the data-loading module and the dependency generation module. The first module uses the previously presented meta-model classes to parse a collection of Android projects saved on disk. It employs the F-Droid class to iterate through the projects and the Application, AppClass and ProGuard class to ascertain their dependencies, imports and rules. Finally, it uses the DBConnect class instance saved inside the initial F-Droid class instance to save all the data to a database.

The second module is the one in charge of generating rules for the dependencies of an application. First, it uses the Application class to detect the declared dependencies of the Android project of the application. It then uses the `rulesForAllDepsExcludingApp()` method, inside

the DataBaseAnalyser class, to consult the database for all the rules included in the ProGuard configuration files of other applications with one or more of the same dependencies. Next, it uses heuristics to determine which of the candidate rules could actually be of use to our application. Finally, it cleans up the generated rule file by removing redundant rules.

4.3.1. Data Loading

Firstly, we present the data loading portion of the solution, where we use the previously presented classes and database to collect and store the data needed by the second module. This process is in charge of building the pool of collective knowledge surrounding dependencies, and their rules that we are later going to consult when generating rules for the dependencies of an application. For this means we continue using the Android projects previously downloaded from the F-Droid repository.

The workflow of this process is presented in ***figure 4.16***. Here we can see how we use the created classes to learn which applications have ProGuard enabled, what dependencies they declare, which rules they use for them, where their source-code classes are and what their contents are.

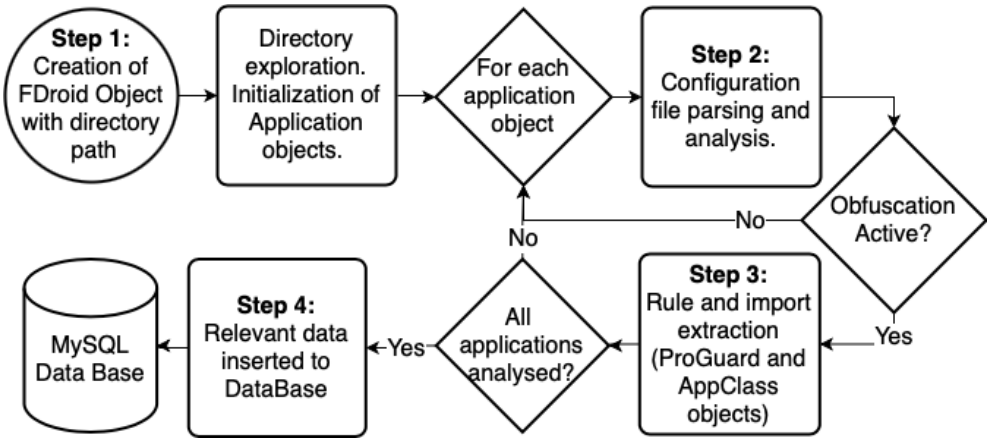


Figure 4.16: Data Loading Workflow.

Step 1: We initialize an instance of the FDroid class object F with the path of the repository to be parsed. We use F iterate through the application directories and create an Application class instance for each of them.

Step 2: Each Application class instance A performs the analysis of its configuration files to determine the declared dependencies and if ProGuard is enabled on their release build. If indeed enabled, we will continue to a further extraction of information. If not, we continue with the next Application class instance on our F-Droid class instance.

Step 3: We extract and store the rules included in the ProGuard configuration files by initializing and instance of the ProGuard class for each ProGuard configuration file found. Configuration rules which are irrelevant to our problem are filtered out, leaving only the rules dedicated to the protection of code and handling of warnings. Namely, the `-keep` and `-dontwarn` rules.

We also filter out rules which represent bad practices in rule redaction as their inclusion on the final returned file will hinder the functioning of ProGuard. This bad practices include the complete suppression of warnings, `-ingnorewarnings` and `-dontwarn **` rules, and rules that are too broad to be practical. For example, if `-keep class **` or `-keep class com.app.**` rules are included, you might as well disable ProGuard entirely.

We also filter out rules referencing source-code classes that have nothing to do with dependencies. The process of detecting application specific rules is detailed in the ***Application Specific Rule Detection*** section below.

Additionally, we initialize an instance of the AppClass class for each Java/Kotlin file in the project to extract and store all import declarations made in the source-code of the application.

Step 4: Finally, we use the FDroid class instance F to iterate through all applications which enable ProGuard, using the DBConnect class to save all of the data associated to the application, its dependencies, its rules and its import declarations.

Application Specific Rule Detection

To aid in the detection of application specific rules, we created a method that isolates the class references inside of the class specification of rules. It does so by parsing rules in accordance with their documented syntax, recognizing the locations in which class references must be declared.

Once obtained, we check if the location referenced is present in the package structure of the application, which is saved as a radix tree in the respective application object. To check if it is an application specific rule, we have three cases to consider:

1. **Specific Class Reference:** If the class reference is pointing at a specific class, which we check if it does not end in a wildcard, we must check if its declared location is present as a key in the tree and if the name of the class is a value related to said key.

   Example To check if the rule ‘‘`-keep class com.app1.data.ClassA {*;}`’’ is referencing the package structure represented in ***figure 4.7***, we first extract its class reference: ‘‘`com.app1.data.ClassA`’’. As ‘‘`com.app1.data`’’ is a key present in the tree, and ‘‘`ClassA.java`’’ a value associated to it, we know the rule is application specific.

2. **Class Reference Using "*"**: As this wildcard is used to represent all elements in the same directory, it is enough to check if the location reference is in fact a key in the package structure tree.

   Example To check if the rule ''`-keep class com.app1.data.* {*;}`'' is referencing the package structure seen in ***figure 4.7***, we notice that the ''`com.app1.data`'' prefix is a key present in the tree, as there are values associated to it. This indicates it is an application specific rule.

3. **Class Reference Using "**"**: As this wildcard is used to represent all elements in the same directory and all sub-directories, it is not enough to check if the prefix of the class reference is a key in the package structure tree, as the location specified might not be a key. We must in turn check if the prefix is a sub-tree of the package structure tree. If that is the case, it means that there must be sub-directories of the specified location which are keys. Therefore, the rule is application specific.

   Example We wish to determine if the rule ''`-keep class com.app1.** {*;}`'' is referencing the package structure seen in ***figure 4.7***. At first glance, we can see that ''`com.app1`'' is not a key of the tree (it has no associated values) but rather it is a sub-tree of it. This means that there are keys in the package structure tree that contain the ''`com.app1`'' prefix. Therefore, it is an application specific rule.

4.3.2. Dependency Rules Generation

Once our database has been populated, we can use this data to predict which the known rules are needed by the dependencies of an application. The `rulesForAllDepsExcludingApp()` method of the DataBaseAnalyser class is in charge of this process. Its workflow can be seen in ***figure 4.17***.

This figure shows how, to generate rules for the dependencies of an application, we first build a query for the database based on its declared dependencies. This query retrieves all the applications that declare one or more of specified dependencies, along with their rules and imports. We then use heuristics to determine which of the rules will be useful for the application.

Step 1 First, to recollect rules for the dependencies of an application A, we create an instance of an Application object with A's path as the initializing argument. This object will contain all of the information of A we need to proceed.

Example We initialize an Application object instance for the application *"FreeTusky"*: app = App('/F-Droid/FreeTusky'). This application was chosen as an example as it already enables ProGuard, has rules in its ProGuard configuration files and includes comments specifying which rules are meant for each one of its dependencies. We will try to replicate the rules for one of its dependencies.

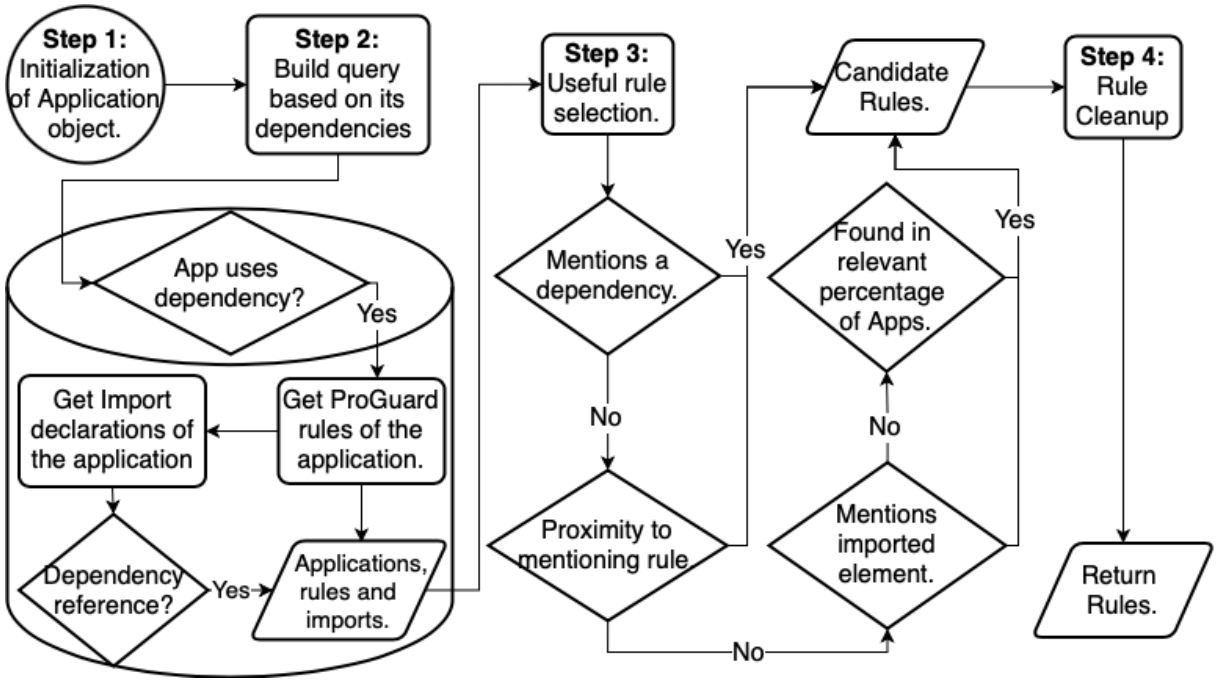


Figure 4.17: Dependency Rule Recollection Workflow.

Step 2 Using the Application object method `getDependencies()`, we learn which dependencies A has declared in it's build.gradle file. We then use the `getAppsWithDeps()` method of the DBConnect class to build a query that will retrieve all the data needed from the database in one call.

This query will first search for all the applications which have a dependency in common with A. Then, it finds all rules and imports associated to the selected applications. We select only the imports which are referencing classes or methods of a dependency, making use of the ''`LIKE`'' query operator to search for the dependency name within the import declaration. The response of the database then consists of rows containing an application name, the names of its dependencies, its import declarations and its rules.

Example The `X.getDependencies()` method reveals *"FreeTusky"* uses, among others, the *"okhttp"* library. The built query then retrieves the names, dependencies, imports and rules from the *"ArchWikiViewer"* and *"AuroraDroid"* applications, among others which use the *"okhttp"* library.

Step 3 We then proceed to select candidate rules from all those returned by our query by using the following heuristics:

1. The first criteria for selection is explicitly mentioning one of the dependencies used by A. This means that all rules which contain the name of a dependency declared by A in their class reference will be selected.

2. As a second criteria, we select rules that explicitly mention classes, methods or annotations being imported from dependencies, but that might not mention the dependency itself. This is useful as there might be rules mentioning this kind of elements which are imported from the dependency but not the dependency itself. Such is the case when importing a class and then targeting every class that extends it with a rule.

3. Thirdly, assuming developers add rules for a specific dependency in the same block of rules, we search for rules near those which do mention dependencies. As we learned from the initial exploration of the composition of F-Droid, on average, each application has between one and four rules for each dependency as seen in ***figure 4.12***. Therefore, we select rules that are within four lines of a dependency mentioning rule.

4. Lastly, we study the remaining rules not selected. For each of them, we calculate what percentage of applications using dependencies of A they are present in. Those which have a high inclusion percentage within applications that share dependencies with A, are selected too.

   The selected percentage is 25%. This decision was made by testing our method with different percentages, and comparing the resulting recall and F1-scores of each percentage. We wished to maximize the values of the F1-score and recall, but, as the recall value decreased when the percentage increase, the F1-score increased when the percentage decreased and the F1-score was consistently lower than the recall, we had to make certain considerations. We decided to use the following function to find our ideal percentage:

$$PonderatedSum(percentage) = 0.3 * recall(percentage) + 0.7 * f1score(percentage) \tag{4.1}$$

   The maximum value of this function will reflect a case when both recall and f1 score are high, but gives more weight to the value of the f1Score as this is notoriously lower than the recall. The results showing 25% to be our ideal percentage are shown in ***figure 4.18***.

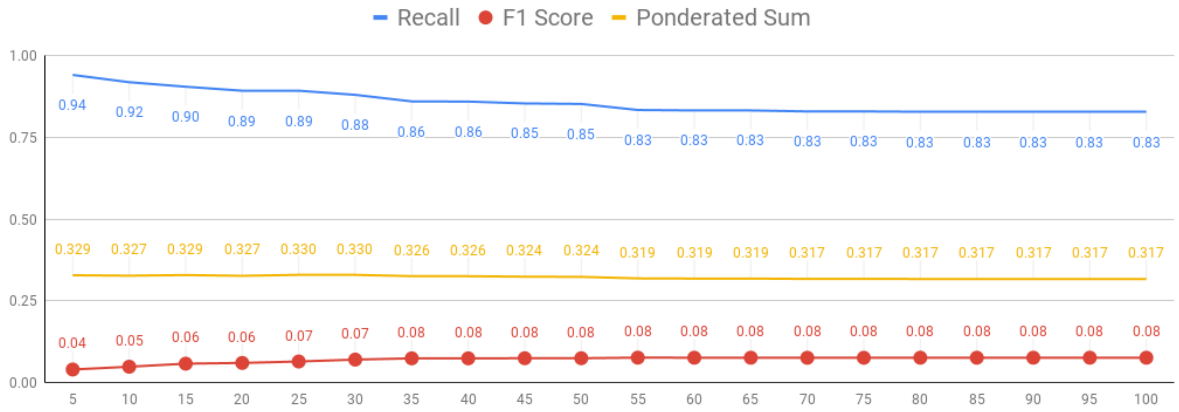


Figure 4.18: Recall and F1-Score of Different Percentages of Inclusion.

Example All four rules originally used for the okhttp library in the ProGuard rule file of FreeTusky are found:

- ‘‘`-keepnames class okhttp3.internal.publicsuffix.PublicSuffixDatabase`’’: Is found in the ArchWikiViewer application proguard rule files (among others) while searching for rules mentioning okhttp.
- ‘‘`-dontwarn okhttp3.internal.platform.ConscryptPlatform`’’: Is found in the ArchWikiViewer application proguard rule files (among others) while searching for rules mentioning okhttp.
- ‘‘`-dontwarn org.codehaus.mojo.animal_sniffer.*`’’: Is found in the AuroraDroid application proguard rule files (among others) while searching for rules in proximity of others which mention okhttp.
- ‘‘`-dontwarn javax.annotation.**`’’: It is found because it was included by 25% of all applications that use the okhttp library.

Step 4 This step takes advantage of the existence of an APK from a debug build with obfuscation disabled. This build must be generated by the user (a trivial task for a developer). If such an APK is present in its default directory, the general package structure of the app, with all dependencies included, is extracted from it. We can then compare the classes referenced by rules to the APK package structure, and remove any rules that reference nonexistent locations in the structure. This way we avoid adding unnecessary rules to the final file returned. This process is presented with more detail in the ***Rule Cleanup*** section below.

Step 5 The selected rules are then written into two different text files, one for keep rules and another for `-dontwarn` rules. This is done to help with the clarity and legibility of these files. Finally, the *build.gradle* file of the application is edited to indicate the location of these new ProGuard configuration files.

Rule Cleanup

To carry out the rule cleanup, we employ a process similar to that detailed in the ***Application Specific Rule Detection*** section. The difference is that during this process we use the extended package structure of an application, including library packages, to filter the rules instead of just the source-code package structure. The process of acquiring and storing the full package structure from an APK is detailed in ***figure 4.19***. This process is added as an optional stage of the initialization of an Application class instance.

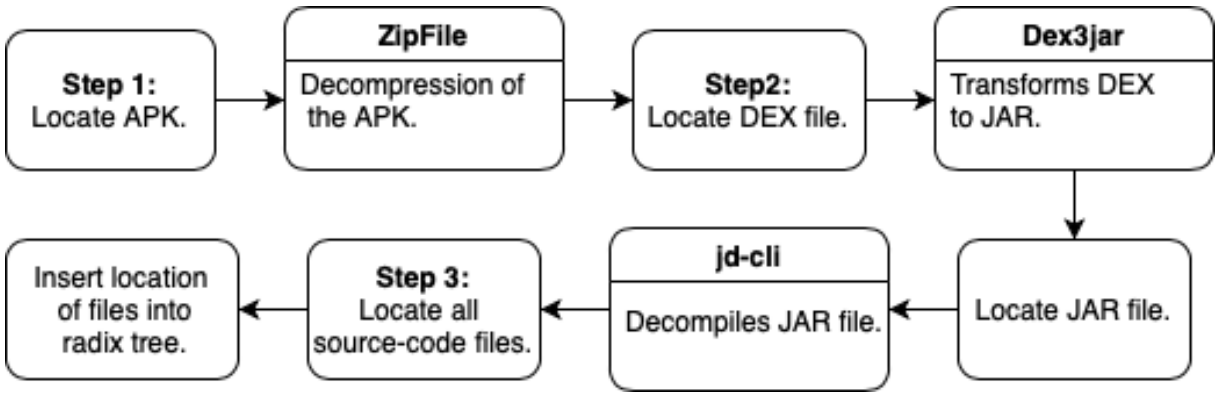


Figure 4.19: Full Package Structure Extraction Workflow.

Step 1: To acquire the extended package structure, we require the presence of a debug build APK in the project. This build must be generated with ProGuard disable, as to preserve the names of the packages. This is a viable requirement for our solution as the building of a debug APK is a basic process of Android application development and easily carried out by the user. If this APK is detected while initializing an instance of the Application class, we can proceed to extract the extended package structure from it.

When located, as APK files are basically just compressed files, we use the *"ZipFile"* Python library to extract all of its contents. Within the extracted files there will be a file with the *".dex"* extension, a Dalvik Executable, this file holds all of the class definitions and adjunct data for the source-code and dependency classes.

Step 2: We need to transform this DEX file into a format we can use. To do this, we use the *"Dex3jar"* library in combination with the *´´jd-cli"* library. *"Dex3jar"* is an open-source command line tool which takes a DEX file and reformats it into a JAR file. Once we have the JAR file, we use *"jd-cli"*, an open-source command-line Java decompiler, to extract the source files from it. This leaves us with a directory containing all Java and Kotlin classes that belong to the app or its dependencies, ordered by packages.

Step 3: We traverse this directory saving all of the class locations into a radix tree as previously done with the source-code package structure. This way, we obtain a package structure representation similar to the one seen in ***figure 4.7***, but that also includes the package structure of the dependencies of the application. This radix tree is saved as a field of the Application class instance, to be used to filter the rules selected during the dependency rules recollection.

At the end of the process which generates rules for the dependencies of an app (***figure 4.17***), we compare the class references in the selected rules to the extended package structure stored in the Application class object. This is done in the same manner explained in the ***Application Specific Rule Detection*** section. Those which reference locations that are

not present in the extended package structure are removed, as they will have no effect on the code of the dependencies.

This process can only be used to cleanup keep rules. As we know, `-dontwarn` rules might be referencing compile-time only dependencies, which will not appear on the applications extended package structure, and their exclusion will result in a process stopping warning being raised when building the application. This is not something to be worried about, as any `-dontwarn` rule that points to a nonexistent location will be telling ProGuard to ignore warnings that are never going to be raised. The only downside to these extra rules is the clarity and legibility of the resulting ProGuard configuration file, hence the separation of `-keep` rules and `-dontwarn` rules in two files when returning the selected rules.

4.3.3. Packaging the Solution

The solution is packaged as a command-line tool. This decision was made as it was an easy and practical way to distribute the functionalities of the developed Python library. Additionally, it is assumed that the knowledge and experience needed to download, configure and use a command line tool are congruent with the expected capabilities of an Android Developer.

To use the solution, the user must use the command: `generateDependencyRules <path-to-andorid-project> <options>`. The options offered are:

- `-v`: Makes the process verbose, printing details while running
- `-d <directory>`: Specifies where to save the generated ProGuard configuration files.
- `-h`: Prints the options information.

While the solution is running, it prints a progress bar on the console which informs the user of the current stage of the process, the time elapsed and the estimated time to completion. When all processes end, it prints a message notifying the user of the location of the generated ProGuard configuration files.

4.4. Application Specific Rules Solution

Along with dependency rules, it is also desirable to be capable of generating rules specifically for the source-code of the application. We complement our solution for dependency rules with another that is dedicated to the generation of rules for the parts of the code that conflict with ProGuard functionality.

To be able to do this, we first must be able to detect possible conflicts between code functionality and obfuscation on the source-code classes. In order to do so, we carry out a

static analysis of the source-code by using the previously presented Application class. As the general detection of reflection is a problem known to be challenging, we focused our efforts on detecting three common practices that clash with code obfuscation:

1. Resource loading from the APK.
2. Java code called from the native side.
3. Data Classes

4.4.1. Resource Loading from the APK

This is the practice of saving data on the APK (images, audio, text, etc.) so that these are available for the code later on. The convention is to search for these resources under the package name of the class that uses them. Therefore, if the class name is obfuscated, trying to load resources from the APK will fail as it will be searching for the resources under the old name associated to the class.

To carry out resource loading in Java, one must use the `Class.getResourceAsStream()` / `getResource()` or `ClassLoader.getResourceAsStream()` / `getResource()` method calls. The presence of these calls inside of a class indicates that the class is loading resources from the APK.

To detect these calls, we take advantage of the Application class and AppClass class. As we know, the Application class initializes instances of the App class for each of the classes in its source-code. This AppClass instances already carry out a static analysis of the code to detect import declarations. We extend this analysis by also searching for the desired method calls. We use the regular expression `\.getResource(AsStream)?\(` to detect any of the resource loading call variations.

We save the package locations of all classes that have a match with the regular expression into a radix tree. This tree is saved as a field in the Application class object.

4.4.2. Java Code Called from the Native Side

With the Java Native Interface (JNI) it is possible to find and call Java classes and methods from C/C++ code. As these calls are made referencing the package name of a class, obfuscating the names of the called classes will result in an error.

To detect any usages of this practice, we must analyze the native side source-code files, namely, all code written in the C or C++ language. We detect the location of these files during the Application class object initialization and then proceed to analyse them.

The detection of the classes involved in this practice is tricky. It is not the class making the

call that must be protected, but rather the class being called by the `env->FindClass(classReference)` JNI method. This can be difficult as the class package name may not be present directly on the method call. More commonly, it will be saved in a variable either locally or in another file entirely, as all JNI interactions may be delegated to a single file which is then included by others.

To select which classes must be protected because of this practice, we inspect the C/C++ code files searching for files which are probably declaring string variables which are then going to be used to call Java code. In order to do this, code files are selected as candidates if they have either a `''JNIEnv''` en variable declaration (which holds the `env->FindClass(classReference)` call), include the JNI header file (which contains the `env->FindClass(classReference)` method declaration) or actually call the class finder method.

Then, we use the `\"([a-zA-Z0-9_]+?/[a-zA-Z0-9_]+?[a-zA-Z0-9/_]*?)\"` regular expression. This will match with any string declaration that follows the standard package name convention, allowing us to extract the package locations of the classes which will be called by the native code.

Lastly, we create a new radix tree in which we save all the package locations of classes being referenced in the native side. This tree is saved as a field in the respective Application class object.

4.4.3. Data Classes

While data classes have no inherent conflicts with code obfuscation, they are commonly used to model data which is then serialized into another medium. This serialization often relies on reflection, which will fail if the name of classes or its members are changed.

In the Kotlin language, data classes are defined at the class declaration in the following manner: `data class className`. Therefore, to detect data classes in the Kotlin language, it suffices to check if the word `data` is included in the class declaration. In Java, there is no special syntax for data classes. This means that we must inspect the composition of classes to deduce if they are data classes.

To do this, we make use of the python library `javalang`. This library can parse Java code and present the elements of it in an ordered manner. To deduce if a class is possibly a data class, we look at its composition and the proportion between their number of private fields and non-null returning methods. We assume that every data class must have at least one method that is used to retrieve the value of a certain private field, the so-called accessor methods. We suspect a class to be a data class if they have n private fields and m methods which return non-null values, where $m \geqslant n$ and $n > 0$.

The locations of all suspected data classes are stored in a radix tree saved as a field in the Application class object.

4.4.4. Redacting Rules

Once we have the package locations saved in radix trees, one for each kind of conflict, we can employ a method to redact rules for these classes.

We design the `getRulesForClasses()` method for the Application class. This method takes a radix tree, a rule prefix and a rule suffix as arguments. The rule prefix is the section of the rule that goes before the class specification. In turn, the rule suffix is the section of the class specification that specifies members of a class. The tree will be one of the three that hold the location of conflicting classes (JNI, APK or data class). This method generates the desired rules by concatenating the rule prefix, the package locations held in the provided tree and the rule suffix respectively.

The rules needed for each of the conflicts are:

1. **APK resource loading classes:** We must protect the names of these classes from obfuscation, as these names will be used when looking up the resources. Therefore, the rule prefix needed will be `-keepnames class` and the rule suffix will be empty. This was, we redact rules of the form: `-keepnames class package.location.of.APKclass`.
2. **Classes called from JNI:** We must protect the code of this class from shrinking and obfuscation. Names are protected from obfuscation because class and method names will be used in JNI calls. Shrinking must also be disabled for these classes as there might be methods deemed to be dead code as they are not referenced by the Java code but are used on the native side. We also need to protect the classes in the type descriptors of the methods and fields of the class. This is to make sure that the parameter types of native methods are not renamed. To achieve this effect, we use the preffix `-keep, includedescriptorclasses class` and the suffix `{ *; }`. This results in rules of the form: `-keep, includedescriptorclasses class package.location.of.JNIclass { *; }`.
3. **Data classes:** As these classes may be used in diverse ways to serialize the data they are holding, we must completely protect the classes suspected to be data classes. This is done with the prefix `-keep class` and an empty suffix. This result in rules of the form: `-keep class package.location.of.dataClass`

In addition, we wanted to take advantage of rule wildcards to avoid returning an unnecessary large number of rules. To achieve this, the method compares the package locations held in the provided tree with the ones saved in the package structure tree of the Application object.

If all files in a directory of the package structure are present in the provided tree, we can use the ‘‘`*`’’ wildcard to redact a rule that points to all files in said location instead of a single rule for each file.

4.4.5. Packaging the Solution

The functions provided by this solution are added to the command line tool packaging by creating the command `generateApplicationRules <path-to-android-project> <options>`. This command has the same options as the `generateDependencyRules` command with the addition of the following:

- `-noRes` disables rule generation for resource loading from the APK;
- `-noJNI` disables rule generation for Java classes called from the native side;
- `-noData` disables rule generation for data classes.

Chapter 5

Validation

This chapter presents the different tests we carried out in order to validate our solutions. Firstly, we introduce the experiment designed to test our methods for rule generation, followed by a brief explanation of the tools developed to carry out said experiment.

Secondly we present the results of our experiment when testing the dependency rule generation module and discuss the significance of said results. We also present the effects of our cleanup module on the generated dependency rules.

Then, we show the effects the generated dependency rules have on the size of an APK built with ProGuard enabled. We compare the results of using generated rules with and without the cleanup module and how they measure up to the effects the original rules of an application have on the APK.

Next, we present a possible complication of our dependency rules solution, how it may be protecting more classes than necessary. We explain the development of tools meant to detect which generated rules are broader than those in the original configuration file and how we use them to see the prevalence of these kinds of rules in our results.

Finally, we test our solution for application specific rules, present the results and discuss their significance.

5.1. Replicating Existing Rules

In order to validate our solutions, we desire to know if they are able to recreate ProGuard rules used by applications that enable obfuscation. To this end, the following validation experiment was designed:

1. From the F-Droid applications, we randomly select 50 applications out of the 649 determined to have ProGuard enabled and which are suited to be tested. The requirement for applications will change depending on which solution we are testing.

2. With our solutions, we generate dependency rules or application specific rules for each of the selected applications. For the dependency rules solution, as the tested applications are present in the crowd-sourced data, we must make sure to exclude their own saved data when generating rules for it.

3. Finally, we compare the generated rules with the original rules present in the ProGuard configuration files of the application. We note how many of the generated rules are relevant for the application and, for the dependency solution, calculate the accuracy, precision, recall and F1-score of our method.

This test is considered adequate to validate our solutions, as we use real applications which have enabled ProGuard on their release build and try to replicate the actual rules the developers included in order to do so. In this way, we can test how our solutions perform in a grounded manner.

5.2. The Tester Class

To be able to run this experiment automatically and efficiently, the Tester class was developed. The methods included in this class give us all the tools needed to run the previously described experiment. The UML diagram of this class can be seen in ***figure 5.1***. The method arguments are omitted for the sake of clarity, a full version can be found in ***Annex A.2***.

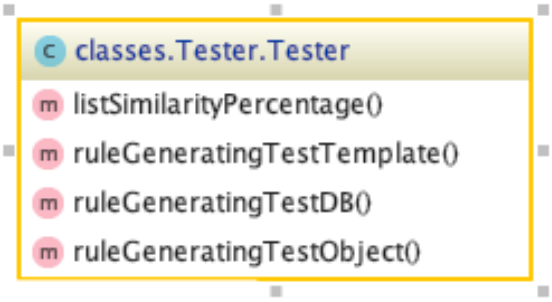


Figure 5.1: Tester Class UML Diagram.

To run an experiment, we use the `ruleGeneratingTestDB()` method. The arguments of this method let us customize different aspects of each test we run, such as: number of applications to test, database to use and inclusion percentage to use for heuristics. The workflow of running a test is seen in ***figure 5.2***

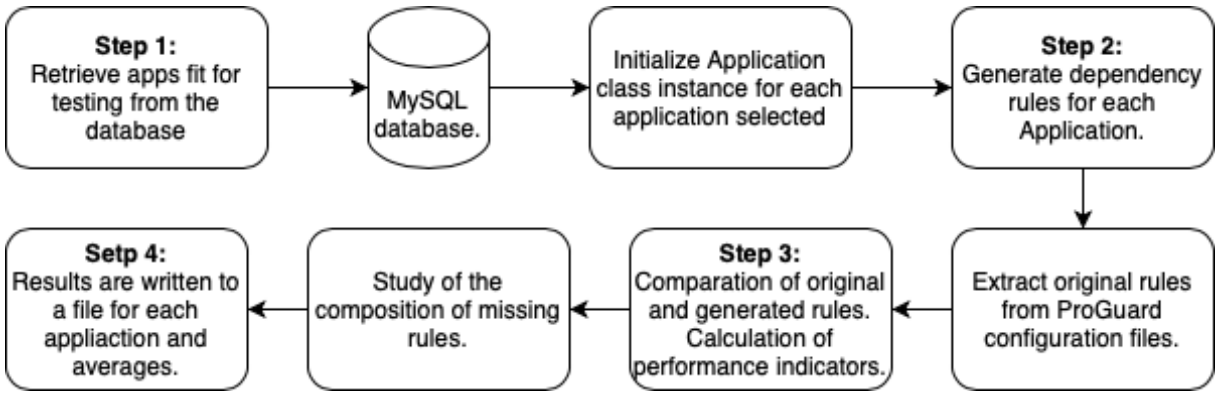


Figure 5.2: Workflow of the Tester Class.

Step 1: The method uses a DBConnect class instance to connect with the specified database and uses its `getAppsToTest()` method to select applications that fulfill our test requirements, getting their paths. Using these paths, we initialize an instance of the Application class for each of these applications. Finally, the specific method that runs our solution is assigned to a variable. The selected applications, specifications and method to test are passed as arguments to the `ruleGeneratingTestTemplate()` method.

The `ruleGeneratingTestTemplate()` method was designed as initially solutions used data loaded in a FDroid class instance rather than in a database. When making the changes needed to adapt our solutions for database usage, we designed tests for each of these methods to compare their results. Here, a design pattern was noticed that allowed us to create a higher order method which receives a list of applications to test and a rule generating method. With these arguments, the `ruleGeneratingTestTemplate()` method then runs our designed experiment. This gives our Tester class better extensibility. If we want to test new methods for rule generation in the future, it will suffice to create a new test method that feeds the needed arguments to our template along with the new rule generating method.

Step 2: Once the `ruleGeneratingTestTemplate()` method has the applications selected for the experiment and the method associated to the solution to test, it carries out our designed validation experiment. For each Application class instance, it uses the specified method to generate dependency rules for it. It also uses the Application class instance to obtain the rules present in its original ProGuard configuration files.

Step 3: It then uses the `listSimilarityPercentage()` method to study the similarities between this two lists of rules, namely calculating the percentage of correctly generated rules. This method also calculates the performance indicators for our solution. This entails detecting which rules are true positives (correctly generated), false positives (rules generated which are not in the original ProGuard files), true negatives (rules correctly ignored by our heuristics) and false negatives (rules that should have been selected by our heuristics but were not) and then using this knowledge to calculate the accuracy, precision, recall and F1-

score of our solution. Lastly, the `listSimilarityPercentage()` method studies the rules that were not generated, determining why these rules were present in the ProGuard rules of the application, and returns all results.

Step 4: Finally, the `ruleGeneratingTestTemplate()` method records results for each application, along with the average of all results, in separate dedicated files.

5.3. Validation Results For Dependency Rules Solution

For this experiment, we wish to test our solution for generating rules for dependencies. For an application to be able to be selected for this test, it must have ProGuard enabled, have at least two rules which mention any of its dependencies and at least two dependencies. We will be generating rules for the dependencies of these applications by choosing from a pool of 5.589 total rules included by 600 applications of the F-Droid repository.

The `rulesForAllDepsExcludingApp()` method of the DataBaseAnalyser class is the one that starts the processes of this solution. This method and the requirements for applications are passed to the `ruleGeneratingTestDB()` method to carry out the experiment. The results of this validation experiment are presented in ***table 5.1***.

This data is the result of testing our solution with the cleanup module deactivated. This decision was made as the cleanup step requires the presence of a debug build APK. This entails the task of building an APK for each one of the apps to be tested, many of which were developed with different versions of Android Studio, Java, Gradle and dependencies. This made it impractical to include this step in random testing, as we would have to generate a debug APK for every application, solving the specific conflicts in versioning and configuration for each case. As this step is used to reduce the false-positive results of our method, we expect the precision and F1-score of this test to be negatively affected. Later we present an experiment made to illustrate how this phase reduces the false positives in the results.

Table 5.1: Performance Measurements for Generated Rules.

Performance Indicators	**Score**
Avg. Correct Rules	0.73
Accuracy	0.69
Precision	0.03
Recall	0.89
F1-score	0.07

As we can see in ***table 5.1***, nearly three fourths of all original rules were generated correctly, this means they are exactly the same as the ones present in the original rules. This is a very telling result. As we are testing the solution against the total number of original rules, application specific rules included, these results convey that dependency rules conform a huge portion of rules in F-Droid applications. In ***table 5.2*** we present the composition of the missing rules. We can see that 31% of the missing rules can be attributed to rules that

Table 5.2: Analysis of missing rules.

Type of Rule	Percentage of missing rules
App Specific	31.0
Dependency	15.93
Java/Android	18.87
Others	34.2

target code specific to the application they are from.

If we take this into account and remove these rules when calculating the success of our method, we see that we are generating correct rules for dependencies 80% of the time. In fact, we are generating all the rules needed for dependencies for 30% of applications tested. If we pair this with our recall score of 0.89 (as shown in ***table 5.1***) we can see that when determining if a rule should be added, most times, we do not leave out actual rules needed to resolve conflicts brought upon by third party code.

This also conveys that the 15.93% of missing rules that reference dependencies are ones that are likely unique to their applications, either by referencing dependencies that are not popular among the community or having been redacted in a peculiar way. The rest of the missing rules present are either rules referencing elements provided by Java and Android studio (i.e, java.lang.String, android.app.Activity) or rules which are referencing elements whose origin cannot be determined. This former set of rules may be referencing dependencies which cannot be determined by parsing a project's *build.gradle* file, such as second degree dependencies: dependencies needed by an actual dependency of the application.

On the other hand, the precision and F1-score of our method calls for a closer look. This indicators tell us that most rules generated are false positives, which may lead to ProGuard protecting blocks of code unnecessarily. While at first this can look detrimental to the result of the solution, the nature of ProGuard rules assures the extra ones will not affect the acting coverage of ProGuard. This is because the extra rules are pointing to classes which required said rule when present on other applications. We can then assume that if said rule isn't on the original rule file, it's because said class isn't being used. Therefore this extra rules will be pointing to elements that are not present in the application's source code, and thus not affecting the coverage in which ProGuard acts. We take advantage of this fact while doing cleanup step of our solution.

5.4. Cleaning Up The Generated Rules

To illustrate the effect that our cleanup module has on the results, we select an app with a recent release, build a debug APK locally and use this to carry out the cleaning phase. The app selected is *"AELF"*, which latest release was added to F-Droid on the 13$^{\text{th}}$ of may, 2020. Results are shown in ***table 5.3***.

With the cleanup module enabled, we went from generating a total of 348 rules to 173

Table 5.3: Performance Measurements for AELF.

	Cleanup Off	**Cleanup On**
Avg. Correct Rules	0.57	0.57
Accuracy	0.74	0.86
Precision	0.03	0.06
Recall	0.76	0.76
F1-score	0.06	0.11

rules. As seen by the results, when we carry out the cleaning phase of the solution, our precision and F1-score are nearly doubled. Still, their value is low. This is due to our inability to filter `-dontwarn` rules, as they may be referencing compile-time dependencies which will never appear on the final APK. In fact, from which 61% of all false positive rules remaining after cleanup are either `-dontwarn` rules. On the other hand, our recall remains the same, so the cleaning process is not disrupting the correct selection of rules.

5.5. Effect of Generated Dependency Rules on Final APK Size

Finally, we wish to know how our generated rules are doing their job. To do this, we are going to observe how the generated dependency rules affect the final size of the APK of an application.

This is a tricky aspect to test. To carry out this experiment, we must find an application for which we can generate all dependency rules and that does not need application specific rules. While a number of applications met this criteria, most were small applications that included a small number of rules. This are not very interesting cases to study.

We finally selected the *"Alldebrid"* application. While we cannot generate all of its rules, it is still a valid case study as the only rule that eludes us is a `-dontwarn` rule. When building the APK with our generated rules, a single ``Reference not Found'' warning prevented the build. But with our knowledge of the probable causes of these kinds of warnings, we were able to realize it was a non-issue and use a `-dontwarn` rule to solve this problem. While it did not generate the entirety of the rules, our solution permitted a user with little knowledge of the application to quickly generate the rules needed by it to generate a build.

To test the effects of the generated dependency rules on the final size of these applications APK we had to generate 4 builds. The first is our control APK, generated with ProGuard disabled. The second build is generated with ProGuard enabled and using the original rules included in the ProGuard configuration files of the application. Thirdly, we generate a build with ProGuard enabled using the generated dependency rules without enabling our cleanup module. Lastly, we generate a build with ProGuard enabled using the generated dependency rules with the cleanup module enabled. The results of this test can be seen in ***table 5.4***.

Table 5.4: Size of Builds for Alldebrid.

Conditions of Build	Size of APK (MB)
ProGuard Disabled	2.6
ProGuard Enabled, Original Rules	1.6
ProGuard Enabled, Generated Dependency Rules Rules, Cleanup Off	2.0
ProGuard Enabled, Generated Dependency Rules Rules, Cleanup On	1.9

We can see that the generated dependency rules, while not at the same level of the original personalized rules which reduce the size of the APK in 38%, were quite effective in reducing the size of the generated file. The uncleaned rules reduced the original size by 23% while the ones that were cleaned up resulted in a decrease of 27%. This validates our conjecture that the false positive rules contained in our generated ProGuard configuration files do not hinder the effects of ProGuard in a significant manner. Still, the results are better when carrying out the cleanup, so this process is valuable for our solution.

5.6. Overprotective Rules

There may be another way in which these extra rules are affecting the coverage of ProGuard. As we saw before, rules can use the `*` and `**` wildcards to match various classes at once. Also, different rule types protect the referenced classes in different ways. This means that while a generated rule might not be in the original rule file, it may be protecting blocks of code protected by an original rule by referencing them in a broader way. Before we continue, we must explain the concept of weaker and stronger rules. We formally define weaker or stronger rules in the following manner:

Definition 5.1 *A rule type F is weaker than a rule type E if, and only if, the protection from ProGuard offered by F is lesser than that of E. In the inverse case, we say F is stronger than E.*

Example The rule type `-keppclassmembernames` is weaker than `-keepnames`, as the former protects only class members (fields, methods, variables, etc.) from obfuscation while the latter protects class members and classes from obfuscation. In turn, `-keep` is stronger than `-keepnames`, as the latter offers protection against obfuscation and the former protects against obfuscation and shrinking.

We use the definition of weaker rules to define overprotective rules:

Definition 5.2 *Let A be a rule which references a group of classes C using a rule type E and B a rule which references a group of classes D using a rule type F. Then, A is overprotective in reference to B if, and only if, $D \subseteq C$ and F is weaker than E.*

Example ‘‘`-keepnames class a.specific.class.A`’’ and ‘‘`-keep class a.specific.class.**`’’ both protect the class A from being affected by ProGuard, but the former also protects all classes in the same directory as A as well as all classes in all sub-directories by using the `**` wild-

card. Additionally, the first rule uses the `-keepnames` rule type, which is weaker than the `-keep` rule type used by the latter. We say then that ''`-keep class a.specific.class.**`'' is overprotective in reference to ''`-keepnames class a.specific.-class.A`''.

The question then arises: when generating rules for the dependencies of an application, are we overprotecting its classes? As the benefits of ProGuard are maximized when applied to the largest number of classes possible without incurring an error, we wish to avoid rules that protect more classes than needed. To explore this issue, we designed an algorithm capable of detecting if a rule is overprotective in reference to another rule.

5.6.1. Overprotective Rule Detection Algorithm

The algorithm takes as arguments two lists, one for the candidate overprotective rule A and one for the rule B. Each list is the result of splitting A and B by their white spaces. It then traverses the elements of the list from start to finish determining if B is overprotective in reference to A.

To do this, it follows the following steps:

1. First, it checks if the length of both rules is equal. Rules with different lengths have different compositions and are therefore different kinds of rules that can't be related in an overprotective way.
2. Then, as the rule type declarations are the first part of rules, we determine if the rule type of A is weaker than the one of B.
3. We traverse the rest of the elements of both rules in tandem, recognizing if they are class references or not.
    - If the current element is a class reference, we use an auxiliary algorithm to determine if the classes referenced in A are encompassed more broadly by the class references of B.
    - If not, it checks if the element in A is the same as the element in the same position in B.

A pseudo-code explaining in more detail the inner workings of this algorithm can be found in ***Annex B.1***. An example of how this algorithm works is found below.

Example We wish to check if A is overprotective in reference to B.

A: `-keep class classReference extends classReference2`
B: `-keepnames class classREference3 extends classReference4`

The steps that our algorithm takes are the following:

0. $len(A) = len(B)$?: As A and B are the same length, we continue to examine each element of both rules. If they had different lengths, it would mean one has more elements than the other, signaling they are different types of rules and therefore not comparable. Such is the case with `-keep class aClass extends anotherClass` and `-keep class aClass`, where the first protects classes with the `aClass` name that extend the `anotherClass` class, while the second protects classes with the `aClass` name that are not extending another class. These are two disjointed groups of classes.

1. (`-keep`, `-keepnames`): The algorithm recognizes the first elements `-keep` and `-keepnames` as rule type definitions. As `-keepnames` protects only from obfuscating and `-keep` from both shrinking and obfuscation, we can say the rule type of B is weaker than A. So we continue. If it were the other way around, and A was weaker than B, we can immediately say A is not overprotective of B.

2. (`class`, `class`): We check the next element, as both are `class`, we continue. If in turn, one was `enum` or `interface`, they would be targeting different types of Java files and are therefore not comparable.

3. (`classReference`, `classREference3`): We use our auxiliary algorithm to check if `classReference` also references the code referenced in `classREference3` in an equal or broader way. If this is the case, we will continue. If not, A is not overprotective of B.

4. (`extends`, `extends`): Same as step 2.

5. (`classReference2`, `classREference4`): Same as step 3.

6. (`__`, `__`): If we reach the end of both rules without interruptions, we know A is overprotective in reference to B.

5.6.2. Overprotective Class Reference Detection Algorithm

The overprotective class reference algorithm was designed to be able to determine if all the classes or directories referenced by a rule Y are also being referenced by a rule X.

It does so by traversing the elements of the class reference of Y using the class reference of X as a map. This means, starting by the first element of the class reference of Y, we only advance to the next element if this is a path that would be taken by the class reference of X when referencing a package structure. These determinations are done by applying our knowledge of class reference syntax and wildcard use.

This algorithm is quite long and considers various possible cases. Instead of explaining each case here, a pseudo-code explaining in more detail the inner workings of this algorithm can be found in ***Annex B.2***. An example of different class references and whether they are overprotective or not can be seen below.

Example 1 We wish to check the overprotective relationship between the following class references:

A: `a.specific.class.**`
B: `a.specific.class.reference`

The steps that our algorithm takes for (A, B) are the following:

1. `(a, a)`: As at this point both rules point to the same class/directory name, we see if the rest of rule A is overprotective in reference to the rest of rule B. If they were different, they would point to different directory/class names and therefore A cannot be overprotective in reference to B.
2. `(specific, specific)`: Same as 1.
3. `(class, class)`: Same as 1.
4. `(**, reference)`: Until this point we know that both rules are referencing the same location. B point to a specific class in the current location. As A finishes with a `**` wildcard, it references all classes in the current directory and all sub-directories. We can see that A is overprotective in reference to B. This would also happen if A ended in a `*` wildcard, as it references all classes in the current directory.
5. `(__, __)`: If we reach the end of both, A is overprotective in reference to B.

Example 2 We wish to check the overprotective relationship between the following class references:

A: `a.**.class.reference`
B: `a.more.specific.class.reference`

The steps that our algorithm takes for (A, B) are the following:

1. `(a, a)`: As at this point both rules point to the same class/directory name, we see if the rest of rule A is overprotective in reference to the rest of rule B. If they were different, they would point to different directory/class names and, therefore, A cannot be overprotective in reference to B.
2. `(**, more)`: A has a `**` wildcard at this position. As this wildcard represents any character including the package separator ".", and is present at the middle of a class reference, it is used to reference all classes which match with the continuing suffix. Basically, A is referencing classes in the current directory, or all sub-directories, whose package name matches`class.reference`
. We are going to be looking for a reference to the next element of A, `class` in B, as this will indicate if B is possibly referencing the same classes. As the next element of B is not `class`, we continue to check the following element.
3. `(**, specific)`: Almost the same as 2, but this time we see that the next element of B is `class`, so we continue our search advancing in both rules. In turn, if we had reached the end of B never finding `class`, A cannot be overprotective in reference to B.

4. (class, class): As at this point both rules point to the same class/directory name, we see if the rest of rule A is overprotective in reference to the rest of rule B.

5. (reference, reference): Same as 4.

6. (__, __): If we reach the end of both, A is overprotective in reference to B.

5.6.3. Prevalence of Overprotective Rules in Generated Dependency Rules.

We integrate these algorithms to the `listSimilarityPercentage()` method with the intention of finding out how prevalent overprotective rules are in the rules generated for dependencies.

Table 5.5: Prevalence of Overprotective Rules in Generated Dependency Rules.

Total Overprotective Rules.	11.7% of all generated rules.
Over Protected Correct Rules.	6.38% of correctly generated rules.

From ***table 5.5*** we learn that, on average, 11.7% of all generated rules are so called overprotective rules. Furthermore, 6.39% of the correctly generated rules are being overprotected. So while some of the code of the applications might be protected needlessly, the majority of it is being adequately protected, even if a small fraction of the rules are broader than needed. From the results of our APK size test, we can infer these broader rules are responsible for the difference in size between the APK built with original rules and the one built with generated rules.

Additionally, while we could remove all detected overprotective rules to reduce the clutter in the returned rules, we would also be hindering our results. In fact, when removing all overprotective rules for our test, the average of the correctly generated rules drops from 73% to 62%.

5.7. Validation Results For Application Specific Rule Solution

Finally, we test our ability to generate rules for classes detected to need them. To do this, we conduct a similar experiment to that of dependency rule generation. We generate application specific rules for 50 applications and compare them to the previously existing app specific rules included in their ProGuard configuration files. The main change is that for an application to be eligible for testing, it has to include at least one application specific rule in its ProGuard configuration files.

We consider this an appropriate validation method as if these practices are present on

an application with ProGuard enabled, there must exist a rule within its original ProGuard configuration files addressing it. The results can be seen in ***table 5.6***.

Table 5.6: App specific Rule Generation.

Avg.	Correct	Generated For
Resource Loading	0.50	0.8
JNI	0.0	0.10
Data Classes	0.05	0.84

For resource loading, we can see that 50% of the generated rules are also present on the original rule files and that this practice is detected in 8% of the applications. But an inspection of these results revealed an important detail: the results were quite binary, either 100% of the rules were relevant or 0% were.

A deeper manual inspection of this case was conducted as the number of involved applications was low. This inspection revealed that applications with 0% of the generated rules present in their original ProGuard configuration were obfuscating most of their classes with general rules. For example, we found 13 cases of this practice inside the application *"Xabber"*, but the generated rules were not considered to be inside the original rule files as Xabber protected all of its classes. This case repeated itself with all applications manually inspected that had 0% correct rules for resource loading. Taking this into consideration, 100% of the generated rules for this practice were correctly generated.

For Java code called from the native side, we see that none of the rules generated were included in the original ProGuard configuration files. This result was quite alarming, so a deeper inspection of the case was conducted.

This inspection consisted in using our method to detect all classes that employed this practice and then using its path to determine the ***build.gradle*** file in charge of configuring its build. This inspection revealed that in 100% of the cases, the classes detected to be called by the native side were relegated to an independent secondary module within the application which did not enable ProGuard in its build, therefore these rules would never appear on the original rules.

This meant that our automatic validation could not be used for validation of the detection of this practice. Alternatively, we conducted a manual inspection of the detected classes and if they were being called from the native side. This inspection revealed that 100% of the classes detected had an actual call from the native side. Still, this validation is inconclusive as we are not able to tell if there are classes being called that are not detected.

For data classes, we see that while detecting them in 84% of the applications, only 5% of the generated rules were present in the original rule files.

Chapter 6

Conclusions

In this chapter we present the conclusions of this bachelor thesis. Firstly, we present a brief summary of the work carried out. Secondly, we recount the objectives we proposed for our solutions and the degree to which they were, or were not, achieved. Thirdly, we discuss the relevance a solution with our results has on the addressed problem. Next, we reflect on the experiences and lessons we obtained during this bachelor thesis and how the unique context in which it was developed affected our work. Finally, we present future works that could stem from the solutions presented on this bachelor thesis. We consider new kinds of solutions along with upgrades that could be made to the current solutions.

6.1. Summary of Work Done

During this bachelor thesis we first presented the problem we set out to solve. This consisted in the difficulties developers experience when trying to enable ProGuard for their builds and all the hassle they had to go through in order to solve them with the redaction of rules. We explained how long and difficult to complete this process can be, resulting in a technical bottleneck that keeps most developers away from ProGuard, and how developing a solution for rule generation would benefit facilitate the implementation of obfuscators.

We then presented the design and functions of a library of Python classes meant to model Android projects along with the main parts of them that are involved when implementing ProGuard. We explained how these classes extract the data related to ProGuard from the projects and how we later used this information to conduct a brief study of the applications contained in the F-Droid repository. We then presented the results of this study and how it led us to ideate a possible solution for dependency rules.

This solution consisted of two phases. First was the data loading phase. On it we repurposed the developed classes. We used them to recollect the data surrounding the rules included by developers to address conflicts brought on by the dependencies of their appli-

cations. Then, we presented the DBConnect class and how it is used to save this data in a database along with creating specific queries for quick and efficient access to it. This process was meant to crowd-source the collective knowledge of developers which have successfully implemented ProGuard. This was done in the hopes of making use of this wisdom to generate rules for the dependencies of an application.

The second phase of the dependency solution was in charge of the actual generation of rules. On it we used the Application class to analyse the application we wanted to generate dependency rules for. We used the methods of this class to detect the declared dependencies of the application. Additionally, we used the DBConnect class to build a query based on these dependencies to retrieve the rules of all applications that share at least one of the detected dependencies. Then we showed how we applied diverse heuristics to select, from the retrieved rules, those that would be useful for our application. Finally, we presented a method for cleaning up the generated rules in an attempt to reduce the number of false positives in the results.

Then we presented a complementary solution that aimed to detect some of the most common conflicts code sources have with ProGuard. Instead of focusing on detecting reflection in the code, which is known to be a challenging task, we focused on detecting specific common practises on the code. We attempted to detect the presence of data classes, Java code being called by the native side and resource loading from the APK by carrying out a static analysis of the code paired with the use of heuristics. We then presented a module which generated rules for the classes detected to need them, explaining the reasons why each problem had to be addressed by a specific rule.

Afterwards, we proceeded to validate our solutions through different tests. First, we presented a test designed to measure the efficacy of our solutions and the class developed to carry it out automatically. The test consisted in using our solutions to generate rules for applications which have successfully implemented ProGuard. We intended to see if our solutions were able to replicate the rules found in the original ProGuard configuration files of the applications.

We then proceeded to show the results of this test for our dependency rules solution. Additionally, we tested the effects of our cleanup module on the generated rules and how they affected the size of an APK when used to generate a build. We also presented a possible complication of the generated solution in the form of overprotective rules. This concept refers to when generated rule protects more code than necessary in reference to the original rules. We presented the design of an algorithm meant to detect overprotective rules and the results of applying said algorithm to the generated rules.

Finally, we tested our solution for application specific applications in a similar manner. We wanted to know how many of the rules generated by our solution were targeting classes that were too targeted by the original rules of applications that already enabled ProGuard. We present the results for our solution and discuss their significance.

6.2. Objectives & Accomplishments

After carrying out the development of our solutions, which aimed to solve the problem addressed by this bachelor thesis, we must now analyse how these solutions measure up to our proposed objectives. Below we recount these objectives and discuss to which degree they were fulfilled. The employed categorization of the completion of an objective is (From higher to lower).

1. **Completely achieved:** The objective is considered to be completed in its entirety without any reservations.
2. **Satisfactorily achieved:** The objective is considered to be completed but with some minor reservations about how it was completed.
3. **Partly achieved:** The objective is considered to be almost completed but not entirely. Further validation is needed to determine if it was completed.
4. **Somewhat achieved:** The objective is not completed, but some advancements were made that signal an eventual completion of the objective.
5. **Not achieved.**

Studying the F-Droid Repository: We aim to better understand the relationship between Android applications and ProGuard.

We consider this objective to have been completely achieved.

Not only were we able to develop the necessary tools to conduct our study, we were also capable of using the results of this study to guide the work done subsequently. The information gained through the study made us realize the prevalent role dependencies play in the difficulties of the implementation of ProGuard. We learnt that the majority of the rules of F-Droid applications were targeting dependencies, and in turn, we learnt that most dependencies were used by multiple applications. This knowledge led us to devise a solution that took advantage of the rules different developers had used to successfully resolve the conflicts brought upon by dependencies and use them for other applications which employed the same dependencies.

Complementary to this, the tools developed for this study were valuable for the entirety of the work done. The classes and methods generated in order to carry out our study were permanently used in our solutions, proving this study not only helped us to better understand the problem, it also set us on the right path to generate solutions for it.

Redaction of Dependency Rules: We aim to generate the rules needed by the dependencies of an application in order to function correctly when ProGuard is enabled.

We consider this objective to be satisfactorily achieved.

As seen by the results of our validation of our solution for dependency rules, we were able to design and develop a solution capable of generating rules for the dependencies of an application. Our test showed the solution is capable of generating, on average, 80% of dependency rules for applications with a recall of 0.89. This shows our solution is generating most of the rules needed by the dependencies of an application while being quite accurate when deciding which rules of our crowd-sourced pool of rules are useful.

While not generating the entirety of the rules, the generated ones are of massive help to a developer. This can be witnessed in our study of APK sizes were, while not being able to generate all rules and having no experience with the source-code of the tested application, the generated rules significantly reduced the effort from our part and permitted us to generate builds of this application with ProGuard enabled.

This objective is considered satisfactorily achieved and not completely achieved because of the number of false positive rules included in the ProGuard configuration file generated, reflected by an F1-score of 0.07. This is something we knew to be a possibility as the nature of using heuristics entails trading precision in favor of accuracy, which is the reason for the presence of so many false positives in our generated files. This is also caused by our inability to filter out `-dontwarn` rules, as this can be used to ignore warnings associated with compile-time only dependencies which will never end up on the APK used for this cleanup. A new method is needed for this task.

While these extra rules generally do not have a major impact in the resulting configuration of ProGuard, and some can be removed in a cleanup phase (increasing the F1-score to 0.11), there are some that are overprotecting classes from the application which includes them. These overprotective rules are suspected of being responsible for the differences between the sizes of the APKs generated with and without the cleanup module enabled.

Additionally, these excess rules could confuse a user by their inclusion on the generated files as they can be misinformed about the rules needed by their dependencies. More work needs to be done in the cleanup phase of the solution in order to consider this objective completely achieved.

Detection of Source-Code Conflicts: We seek to detect classes of the source-code of an application which may conflict with obfuscation.

- **Resource loading from APK:** This objective is considered to have been partly achieved.

  We were able to detect parts of the code that employ this practice as its presence is announced in a straight-forward way by the use of certain method calls. Nevertheless, validation of this detection turned out to be quite difficult as the repository used for validation presented only a few cases for us to study, and those that did use this practice often relied on bad practice rules to address these problems (like protecting the entirety

of their classes).

In order to further validate the detection of this practice, and redaction of rules for it, we need to procure a validating data-set which has a more prevalent use of this practice and that does not include bad practice rules in their original files.

- **Java code called by JNI:** This objective is considered as partly achieved.

  We were able to design a method that detects this practice in an application. Nevertheless, validation of this detection turned out to be difficult. This was because the times this practice was used, it was relegated to a separate secondary module of the project that was built with ProGuard disabled.

  While a manual inspection of the detected instances of this practice revealed it to be quite successful, this inspection is inconclusive as only the cases where it was detected and ended up being present were observed. A deeper and more complete validation where we thoroughly document the presence of this practice on multiple applications and then test our ability to detect them is required. Unfortunately, this kind of study requires an appropriate amount of time to be done correctly, and these deficiencies were detected too late in the process of our work.

- **Data Classes:** This objective is considered not achieved.

  This is because of the method employed to deduce if classes are data classes is quite inaccurate, unable to detect instances of this practice correctly. This is reflected by the high detection among the applications used for validation while the generated rules were rarely included in the originals.

  The reason behind this is our simplistic approach to the detection of data classes. Definitely, the method of observing the number of private fields and non-null returning methods is not enough to let us deduce if a class is indeed a data class and a more thought-out method needs to be designed.

Redaction of Rules for Source-code Conflicts: We aim to redact rules that will solve the conflicts detected in the source-code.

This objective is considered satisfactorily achieved.

The methods designed to redact rules for these conflicts were successful in doing so. The research done on the problem each of these practices presented allowed to correctly select the type of rule needed to address them. Then, the module designed to redact rules for classes, which allows us to define the specific rule we wish to redact, made it possible for us to successfully generate rules for the classes deemed to need them.

Additionally, the module takes advantage of the wildcard system offered by ProGuard to reduce the number of rules it generates by using the * wildcard to target numerous classes within the same directory when possible.

Still, the utilization of the pattern matching system could be included to generate rules that use regular expression to match with specific classes.

6.3. Relevance of the Developed Solutions

6.3.1. Dependency Rules Solution

We find our dependency rules solution to be highly relevant in regards to the redaction of ProGuard rules by developers as this removes most of the hassle associated with the joint use of dependencies and ProGuard.

Firstly, to the best of our knowledge, this is the first solution that targets this specific problem. Its mere existence might shine a light on this problem and inspire more developers to come up with their own solutions. The solution and results were considered relevant enough to redact an academic paper detailing them.

As previously mentioned, ProGuard mentions in its documentation that in order to get proper results, a developer must be familiar with the inner workings of the code. As this solution is able of generating the majority of rules needed by dependencies, it will allow developers to dedicate less time to addressing the difficulties in ProGuard implementation brought on by third parties. This will be of great help to developers as it will remove the need to redact rules for code they are completely unfamiliar with and allow them to concentrate on their own code.

Additionally, this solution will reduce the time invested in the redaction of rules significantly. As we learnt from our initial study, dependency rules conform a significant portion of the total amount of the rules an application includes. Being able to generate 80% of them will greatly reduce the total time spent redacting rules.

Finally, as this solution reduces the time needed for ProGuard implementation and the complexity associated with the redaction of rules for third party code, it might encourage more developers to implement ProGuard. Consequently, this might aid in reducing the average size of Android applications and decrease the number of applications which are cloned.

6.3.2. Application Specific Rule Solution

While less relevant than the one developed for dependency rules, our solution for the generation of application specific rules might still be of use for developers.

Specifically, our methods for detecting Java code called from the native side and resource loading may be helpful to developers by pointing out probable causes of conflicts with ProGuard in their source-code and writing rules for them. This would reduce the time invested in the redaction of rules.

This can also be applied to the source-code of third party libraries. In the case our

dependency rules solution is not able to generate all the needed rules for a determined dependency, we could use the application specific rules solution to detect possible conflicts in its source-code and help developers fill in the missing rules.

In conclusion, we can see that both our solutions serve to alleviate the effort made by developers while implementing ProGuard in their applications, promoting its use and spreading the associated benefits.

6.4. Lessons Learnt

While the process of carrying out the work related to the generation of our solutions was an enriching experience that resulted in many lessons, we can single out three which were found to be paramount among them.

6.4.1. Learning From Reality

While it might seem obvious that knowing more about a problem will help in the development of a solution, this relates more to the source of said knowledge.

At the beginning of the process of this bachelor thesis, after learning about ProGuard and the problem surrounding it in an academic manner, starting development on the proposal of a solution was difficult. This was because there were no clear paths as to where to start working as there were almost no studies made about the matter at hand and no comparable solutions. Additionally, only comprehending the problem and its causes in a theoretic manner played against us. The perceived complexity of it made the idea of a possible solution quite daunting.

It was at this moment that the guiding professors made the suggestion of carrying out a short study of real Android applications that implement ProGuard. This small study of real life cases was able to ground our knowledge and helped us devise a possible solution to the problem.

This taught us that before being intimidated by our theoretical knowledge of a problem, exacerbated by anxious imagination, we should always remain calm and study the problem in a real manner.

In our case, the study not only provided guidance as to where to focus our efforts, it also served as motivation to develop the Python library that ended up being used in the entirety of the following work. This leads us to the next lesson learnt.

6.4.2. Good Design is a Valuable Tool

When we started working on the Python library to conduct our study, we were faced with a choice: Do we take the easy way out and just parse the files of the projects and immediately study them, or do we make the effort to create classes to hold this data first? Fortunately, the latter option was chosen.

By creating these classes early in the development process, we inadvertently created valuable tools that were used for the rest of the work to come. As we followed an object-oriented programming approach to generate these classes, what resulted was a library modularized by its functions respecting the different actors involved in the problem.

As we continued our work and re-purposed these classes for different uses, the design decisions made at the beginning facilitated the modification, extension and debugging of each of the classes of the library. Even when making the change from holding the data on class instances to the use of a database or going from focusing on dependencies to focusing on the source-code, the design of these classes made development easier. If this design had not been present from the beginning in a simplistic form which allowed us to gradually increment its complexity, the creation of the library may have resulted in a much more convoluted and messy group of methods.

This taught us that an early and thoughtful design, even if tedious or seemingly unnecessary, is incredibly helpful to a developer by allowing a slow and manageable escalation of its complexity.

6.4.3. Work Ethic During Isolation

For this section I will be speaking in the first person as it is is a more personal subject.

The unique context in which the development was carried out obligated me to take a more aggressive approach to the organization of the work that needed to be done.

During the whole process of this bachelor thesis the world was shaken by the Covid-19 pandemic. This resulted in being in a state of physical isolation and reclusion while working on the solutions. This had a myriad of effects.

Firstly, any semblance of a routine or structure was quickly erased. The small rituals I commonly used to pace my life were erased. My commute to and from the university, which I commonly used to plan what I had to do that day or the next, disappeared. As did the spaces I usually used to focus my mind. During my life as a student I had grown accustomed to only working in the faculty and resting at home. The separation of these spaces helped me concentrate and rest whenever it was needed. The lockdown we experienced made me have to rewire my habits and transform my resting spaces into my work spaces.

Secondly, the psychological effects of the lockdown played against me. The lack of social interaction and the monotony of life spent in isolation had a negative effect on my mental health. Finding motivation to work became exceedingly difficult the more time passed.

Thirdly, the nature of this bachelor thesis, in which I had no development team to rely on besides my professors, meant that I was the only one in charge of organizing and distributing the workload. Normally this would not have been a major issue, but in summation to the previously mentioned lack of structure in my life and worsened mental health, it resulted in having trouble organizing my times and avoiding procrastination.

This meant a more concrete solution for the organization of my time was required. The method I employed was the Kanban framework. I used masking tape and post-its to create a board on the wall of my room, where I would be obligated to see it when waking up and when going to sleep. On it I did not only organize the work related to my solutions, but also mundane things of my life like making my bed and exercising.

This solution helped me find structure again as I would grow accustomed to the routine of waking up, updating the board and start completing my morning personal tasks to then move on to general work tasks. It also helped me with my mental health, as the small successes of moving post-its from "to-do" to "doing" and finally "done" kept me motivated to continue completing tasks and served as rewards. This resulted in me being able to have a better organization of my work, allowing me to work faster and more efficiently.

6.5. Future Works

6.5.1. Further Validation of App Specific Rules Solution

The first work that should be carried out is to carry out a further validation of the application specific rules solution.

In order to do this, we must procure a data-set more suitable for the job, where these practices are used along with ProGuard. Open source applications may not be ideal for this, and we must procure another source of testing.

Additionally, more thorough testing of our detection capabilities must be carried out, taking the time to document the appearances of practices on applications and testing if our solution is capable of detecting them completely.

6.5.2. Further Testing with Real Cases

Another work that stems out of the solutions developed is to carry out further testing of them. This will help us better understand their strengths and weaknesses in order to guide

future development.

The testing we have in mind is to use our solutions on applications that have release builds with ProGuard disabled. To do this, we must find applications that present errors or bugs when built with ProGuard enabled and then generate rules for them using our solutions. We then must generate a new build of the application and see what problems are solved and which remain.

With this test we look to test if our solutions are actually capable of working in applications that did not have ProGuard in mind during development.

6.5.3. Growing Nurturing & Cleaning our Pool of Knowledge

An important thing to do after the development of the dependency rules solutions is to grow our data-set. But this growth must be careful, we must procure to be nurturing our collective knowledge and not hindering it. When adding applications to the database we must ensure they employ specific appropriate rules.

It would also be beneficial to develop further our detection of bad practice rules in order to clean our pool of knowledge from any element of it whose presence is more damaging than its absence.

6.5.4. Perfecting and Expanding Conflict Detection on Source-Code

More work needs to be poured into our detection of data classes. As we were not able to carry out a precise detection of data classes for our solution, this must be amended during future works.

Also, expanding our knowledge about practices that cause conflicts with ProGuard functionality, how to detect them and which rules to redact for them will better our application specific rules solution.

6.5.5. Paring with Reflection Detectors

Lastly, as we realized that the detection of conflicts with ProGuard functionality is quite difficult, we could take advantage of tools that enable users to detect reflection on their apps.

Although not specifically related to the generation of rules for obfuscators, the work done by Li et al [12] is considered important. They focused on the development of DroidRA, an open source tool capable of detecting reflection in Android applications. With their approach

they achieved significant results, detecting 81.2% of reflection instances in 500 randomly selected apps.

Although their paper does not cover the use of this tool in the generation of obfuscation rules, it does mention that it could be of great help for traditional code analyzers that have problems with reflection. As this tool provides an equivalent version of the application but with the reflective calls enhanced with standard java calls, facilitates the static analysis of the code, this tool could aid us in analysing source-code in search for possible conflicts with ProGuard within them.

Additionally, this tool can be very useful in the field of generating obfuscation rules. As it contains a module that provides a list of classes, methods and values that use reflection, one of the big problems when obfuscating, it could be paired with our rule redaction module to generate rules far all detected reflective instances.

Bibliography

[1] Android: *Add build dependencies*, Jul 2021. https://developer.android.com/studio/build/dependencies, visited on 13 July, 2021.

[2] Android: *Configure build variants*, Jul 2021. https://developer.android.com/studio/build/build-variants, visited on 13 July, 2021.

[3] Android: *Projects overview*, May 2021. https://developer.android.com/studio/projects, visited on 13 July, 2021.

[4] Android: *Shrink, obfuscate, and optimize your app*, Jul 2021. https://developer.android.com/studio/build/shrink-code, visited on 13 July, 2021.

[5] Corporation, Evans Data: *Mobile developer population reaches 12m worldwide, expected to top 14m by 2020*, 2016. https://evansdata.com/press/viewRelease.php?pressID=244, visited on 6 May, 2020.

[6] Dominik Wermke, et al: *A large scale investigation of obfuscation use in google play.* In *34th Annual Computer Security Applications Conference*, pages 222–235, 2018.

[7] Hecht, Geoffrey, Cyprien Neverov, and Alexandre Bergel: *Vision: Alleviating android developer burden on obfuscation.* In *IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems (MOBILESoft '20)*, New York, NY, USA, 2020. Association for Computing Machinery. https://doi.org/10.1145/3387905.3388611.

[8] Hohensee, Barbara: *Introducción a Android Studio*, page 8–58. BadPress, 2014.

[9] Kai Chen, Peng Liu and Yingjun Zhang: *Achieving accuracy and scalability simultaneously in detecting application clones on android markets.* In *In Proceedings of the 36th International Conference on Software Engineering*, pages 175–186, 2014.

[10] Kaliciński, Wojtek: *Troubleshooting proguard issues on android*, 2017. https://medium.com/androiddevelopers/troubleshooting-proguard-issues-on-android-bce9de4f8a74, visited on 24 June, 2021.

[11] Kaliciński, Wojtek: *Practical proguard rules examples*, 2018. https://medium.com/androiddevelopers/practical-proguard-rules-examples-5640a3907dc9, visited on 24 June, 2021.

[12] Li, Li, Tegawendé F. Bissyandé, Damien Octeau, and Jacques Klein: *Reflection-aware static analysis of android apps*. In *31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, page 756–761, New York, NY, USA, 2016. Association for Computing Machinery, ISBN 9781450338455. https://doi.org/10.1145/2970276.2970277.

[13] Michael Backes, Sven Bugiel, Erik Derr: *Reliable third-party library detection in android and its security applicationss*. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, page 356–367, 2016.

[14] MindOrks: *Things to care while using proguard in android application*, 2019. https://bit.ly/3wTHjSL, visited on 7 May, 2020.

[15] Oracle: *Creating and Using Packages*. https://docs.oracle.com/javase/tutorial/java/package/packages.html, visited on 13 July, 2021.

[16] Pelgrims, Kevin: *Gradle for Android*, page 1–33. Packt Publishing Limited, 2015.

[17] ProGuard: *Proguard manual*, 2019. https://www.guardsquare.com/manual/home, visited on 7 May, 2020.

Appendix A

Complete UML Diagrams

A.1. Application Class

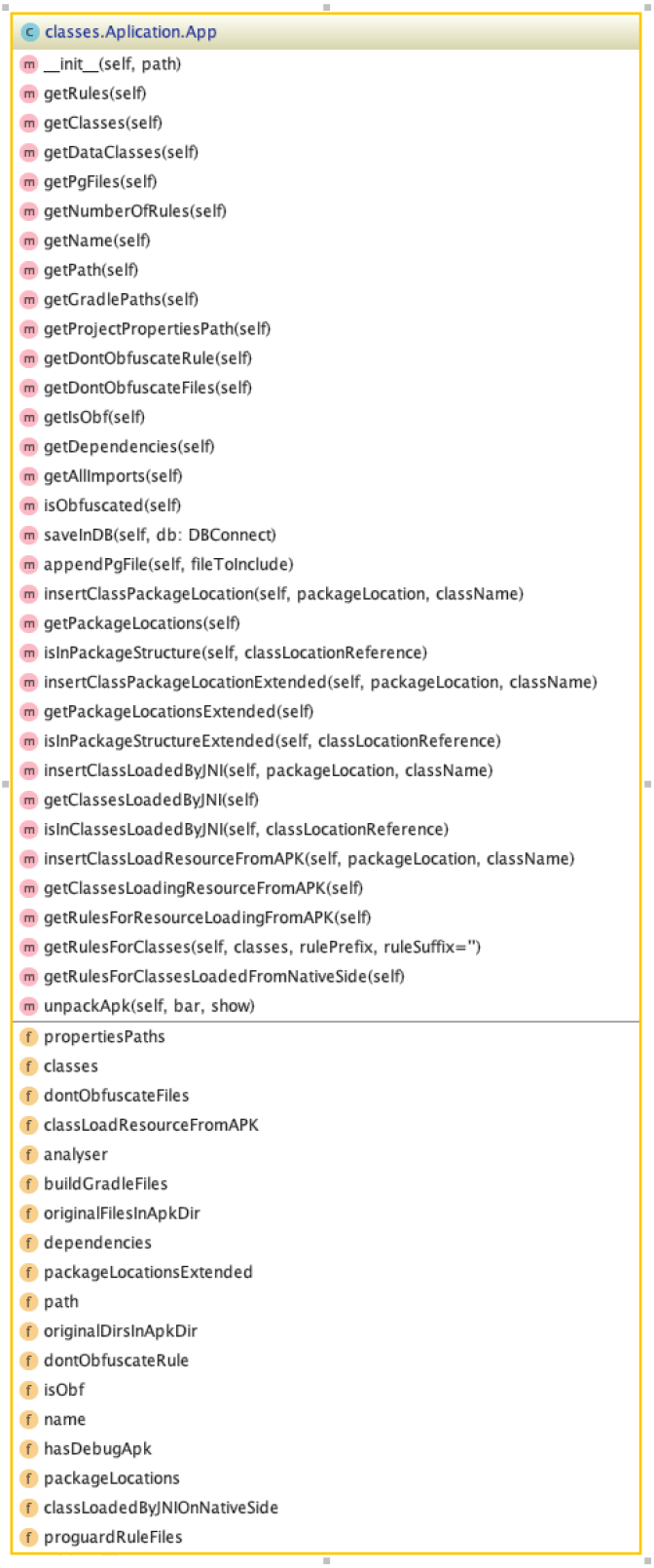


Figure A.1: Application Class UML Diagram.

A.2. Tester Class

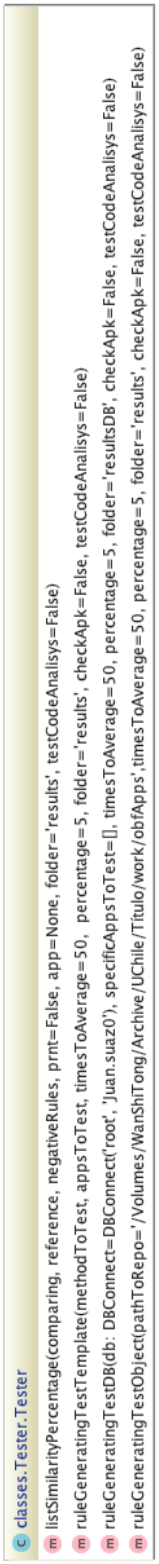


Figure A.2: Tester Class UML Diagram.

Appendix B

Code

B.1. Overprotective Rule Detection Algorithm

Code B.1: Overprotective Rule Detection Algorithm Pseudo-Code.

```
# Inputs are lists where each list element is a singular rule element,
# B is the rule candidate to be overprotective in reference to A.
input: list<string> A, list<string> B
output: boolean

begin
    # Detect the amount of elements in the rules
  int lenA ← length(A)
  int lenB ← length(B)

  # Detect the class references of the rule
  list<string> classreferencesA ← isolate_class_references(A)
  list<string> classreferencesB ← isolate_class_references(B)

  # Detect the amount of class references in the rules
  int lenClassA ← length(classreferencesA)
  int lenClassB ← length(classreferencesB)

  # If the lengths of the rules are no the same  or
  # the number of class references is not the same the
  # rules have different definitions and are therefore
  # not comparable
  if lenA = lenB and lenClassA = lenClassB
    for i in range(lenA)
        # we compare each element from the rules
        segmentA = A[i]
        segmentB = B[i]

        # If the classes referenced by A are not encompassed by the
        # class references in B, B  is not overprotective in reference to A,
```

```
        # we use an auxiliary algorithm to check this
        if segmentA and segmentB are both class references
            if not recursive_call(segmentA, segmentB)
                return False
        # Ff the rule type of A is not weaker than the rule type of B,
        # B  is not overprotective in reference to A
        elif segmentA and segmentB are rule type declarations
                if segmentA is not weaker rule type than segmentB
                    return Flase
        # Finally, if any segment that is not a class reference or rule type
        # declaration are not the same, i.e segmentA = "class" and segmentB = "interface"
        # the rules are not coparable
        elif segmentA ≠ segmentB
            return False
    else
        return False

    # If we arrive here, both rules have been traversed completely
    # and we can assure that B is overprotective in reference to A
    return True
end
```

B.2. Class Reference Sub-Group Detection Algorithm

Code B.2: Overprotective Rule Detection Algorithm Pseudo-Code.

```
# Inputs are lists where each list element is a singular class reference element.
# Y is the class reference to determine if is sub-group of X
input: list<string> X, list<string> Y
output: boolean

begin
    # Detect the head and tail for both lists
    string headX ← head(X)
    string headY ← head(Y)

    list<string> tailX ← tail(X)
    list<string> tailY ← tail(Y)

    # Detect we reached the end of either X or Y
    bool end_of_X ← is_empty(tailX)
    bool end_of_Y ← is_empty(tailY)

   # We know a single * wildcard is use to signal all classes in a directory.
   # The single * can only be used at the end of a class reference
   if headX = "*"
       # If Y has also a * at the end they are equivalent
        if headY = "*"
```

```
23              return True
24          # If Y has a ** it cant be overprotected by X, as X points
25          # to all classes in a directory and Y to something past this directory
26          elif headY = "**"
27              return False
28          else
29              # If not a wildcard, and Y ended, X is pointing to all classes in the same location as
30              # the class referenced in Y
31              if end_of_Y
32                  return True
33              # If Y not ended, it still has classes to specify further
34              # but as X has * at the end, Y points to a location unaffected by X
35              elif not end_of_Y
36                  return False
37      elif headX = "**"
38          # If head of Y is *, it has to be the end of Y
39          if headY = "*"
40              # If end of Y and not X, X points to a directory deeper than that of Y
41              if not end_of_X
42                  return False
43              # If both end, as ** is broader than *, X overprotects Y
44              elif end_of_X
45                  return True
46          elif headY = "**"
47              # If both end in **, they are equivalent
48              if end_of_Y and end_of_X
49                  return True
50              # if neither ends, we check if what is specified beyond the ** in X is referenced
51              # by the tail of Y. As Y has a ** in this location, by definition it cant be followed
52              # by ** or * (redundant), therefore to reference the tail of X in the next location
53              # the next reference must be equal. We continue checking their tails.
54              elif not end_of_Y and not end_of_X
55                  if tailY[0] = tailX[0]
56                      return recursive_call(tailX, tailY)
57                  # if not, X can still be referenced in the tail of Y, we check if its true
58                  else
59                      return recursive_call(X, tailY)
60              # If Y ends in ** but X continues to specify, Y is broader than X
61              elif end_of_Y
62                  return False
63              # If X ends but Y not, X is referencing everything referenced by Y by definition
64              elif end_of_X
65                  return True
66          else
67              # if X end in ** and Y in a class name, X is overprotective in reference to Y
68              if end_of_Y and end_of_X
69                  return True
70
71              # If neither end here, the next position of X must be a directory or class name
72              # and the next element of Y can be **, * or class/directory name
73              elif not end_of_Y and not end_of_X
74                  # if the next elements are equal, they are both directory/class name. Meaning,
```

```
75              # to this point they are referencing the same directory/class name. We check
76              # if the tail of X is overprotective of the tail of Y
77              if tailY[0] = tailX[0]
78                  return recursive_call(tailX, tailY)
79              else
80                  # If not equal, if the next element of Y is **, as the next element of X
81                  # must be pointing to a directory/class name, Y is broader than X
82                  if tailY[0] = '**'
83                      return False
84                  # Else, the next element of Y must be a directory name or *.
85                  # If a name, X can be still be referencing elements in the tail of Y.
86                  # we make a recursive call to check. If its *, this is the end of Y and
87                  # the recursive call will return false as intended
88                  return recursive_call(X, tailY)
89          # if Y ends but X continues, Y points to a specific class and
90          # X to something past this class. X cant be overprotective to Y
91          elif end_of_Y
92              return False
93          # if X ends in ** but Y doesn't, X is overprotective to Y.
94          elif end_of_X
95              return True
96      else
97          # If X has a class/directory in this location and Y ends in a *,
98          # if, X ends, Y is brader than X. If X continues, it points to something
99          # in a deeper location than Y. Either way we return False
100         if headY = "*"
101             return False
102         # If X has a class/directory in this location and Y has a **,
103         # in this location X traverses a single directory/class while
104         # Y traverses all. Y is broader than X.
105         elif headY = "**"
106             return False
107         else
108             # If both end in a class name, if they are equal, they are equivalent
109             if end_of_Y and end_of_Y
110                 return headY = headX
111             # If both have a directory name in this location, if its the same it means
112             # that until this point X references the same directory than Y. We check if
113             # tail of X overprotects Y. If not, they reference different directories and
114             # therefore X cant be overprotecting Y.
115             elif not end_of_Y and not end_of_Y
116                 if headY = headX
117                     return recursive_call(tailX, tailY)
118                 else
119                     return False
120             # If X does not end but Y does, Y is pointing to a class in a
121             # deeper location than that referenced by X. X cant be
122             # overprotecting Y
123             elif end_of_Y
124                 return False
125             # If X ends but Y does not, X its pointing to a class and Y
126             # to something in a deeper location.
```

```
127             elif end_of_X
128                 return False
129 end
```