
**NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS**

**SCHOOL OF SCIENCE**
**DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS**


**BSc THESIS**

# Integration of static analysis results with ProGuard optimizer for Android applications


**Christos V. Vrachas**

**Supervisors:** **Yannis Smaragdakis,** Professor NKUA
**Anastasis Antoniadis,** PhD Student NKUA



**ATHENS**

**OCTOBER 2017**


**ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ**

**ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ**
**ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ**


**ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ**

# Ενσωμάτωση αποτελεσμάτων στατικής ανάλυσης στο βελτιστοποιητή ProGuard για εφαρμογές Android


**Χρίστος Β. Βραχάς**

**Επιβλέποντες:** **Γιάννης Σμαραγδάκης,** Καθηγητής ΕΚΠΑ
**Αναστάσης Αντωνιάδης,** Διδακτορικός Φοιτητής ΕΚΠΑ



**ΑΘΗΝΑ**

**ΟΚΤΩΒΡΙΟΣ 2017**

**BSc THESIS**

# Integration of static analysis results with ProGuard optimizer for Android applications

**Christos V. Vrachas**
**R.N.:** 1115201300024

**SUPERVISORS:** **Yannis Smaragdakis,** Professor NKUA
**Anastasis Antoniadis,** PhD Student NKUA

**ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ**

# Ενσωμάτωση αποτελεσμάτων στατικής ανάλυσης στο βελτιστοποιητή ProGuard για εφαρμογές Android

**Χρίστος Β. Βραχάς**
**Α.Μ.:** 1115201300024

**ΕΠΙΒΛΕΠΟΝΤΕΣ:** **Γιάννης Σμαραγδάκης,** Καθηγητής ΕΚΠΑ
**Αναστάσης Αντωνιάδης,** Διδακτορικός Φοιτητής ΕΚΠΑ

# ABSTRACT

In the era of the widespread usage of mobile devices, the requirement for more lightweight and faster applications continues to exist, aiming for better system resources utilization. In order to achieve this, many tools have been developed for optimizing applications as much as possible, with the aid of program analysis. In the case of Android applications, the main tool used is the ProGuard optimizer. Given configuration files, either default or defined by the programmer, ProGuard performs optimizations at the bytecode level.

The default configurations are quite simple and conservative, thus leading to missing chances for further optimizations. Moreover, many applications usually come with library code, or use dynamic language features, such as reflection, for which further configuration may be required by the developer.

We present an attempt to automate the generation of ProGuard configurations, using Doop framework: a Java static analysis framework. We include an experimental evaluation of the various configurations produced by the Doop framework, against the default suggested configurations.

**SUBJECT AREA:** Static Program Analysis and Program Optimization

**KEYWORDS:** static program analysis, program optimization, doop framework, proguard, reflection

# ΠΕΡΙΛΗΨΗ

Στην εποχή της ευρείας χρήσης των κινητών συσκευών, η ανάγκη για ελαφρύτερες και ταχύτερες εφαρμογές εξακολουθεί να υπάρχει, με σκοπό την καλύτερη αξιοποίηση των πόρων του συστήματος. Προκειμένου να επιτευχθεί αυτό, έχουν αναπτυχθεί διάφορα εργαλεία για τη βελτιστοποίηση των εφαρμογών όσο το δυνατόν περισσότερο, μέσω της ανάλυσης προγραμμάτων. Στην περίπτωση των Android εφαρμογών, το κύριο εργαλείο που χρησιμοποιείται είναι ο βελτιστοποιητής ProGuard. Με βάση είτε κάποια προεπιλεγμένα αρχεία ρυθμίσεων, είτε κάποια που ορίζονται από τον προγραμματιστή, το ProGuard εκτελεί βελτιστοποιήσεις σε επίπεδο bytecode.

Οι προεπιλεγμένες ρυθμίσεις, είναι αρκετά απλές και συντηρητικές, με αποτέλεσμα να χάνονται κάποιες ευκαιρίες για περαιτέρω βελτιστοποιήσεις. Επιπλέον, πολλές εφαρμογές συχνά εμπεριέχουν κώδικα βιβλιοθηκών, ή κάνουν χρήση δυναμικών χαρακτηριστικών της γλώσσας, όπως η ανάκλαση, για τα οποία πιθανώς χρειάζονται περαιτέρω ρυθμίσεις, απο τον προγραμματιστή.

Παρουσιάζουμε μια προσπάθεια αυτοματοποίησης της παραγωγής των ρυθμίσεων του ProGuard, χρησιμοποιώντας το Doop: ένα framework στατικής ανάλυσης προγραμμάτων Java. Συμπεριλαμβάνουμε μια πειραματική σύγκριση των διαφόρων ρυθμίσεων που παράγονται από το Doop, έναντι των προτεινόμενων προεπιλεγμένων ρυθμίσεων.

**ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ:** Στατική ανάλυση και βελτιστοποίηση προγραμμάτων

**ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ:** στατική ανάλυση προγραμμάτων, βελτιστοποίηση προγραμμάτων, doop framework, proguard, ανάκλαση

*To my family.*

# ACKNOWLEDGMENTS

I would like to thank my supervisor, Prof. Yannis Smaragdakis for giving me the chance to work on this subject, as well as for his motivation and support during these months.

I would also like to thank my supervisor, PhD student Anastasis Antoniadis, for the help towards completing this thesis, and every PLaST Lab member, for sharing their knowledge and wisdom.

*October 2017*

# CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# PREFACE

This project was developed in Athens, Greece between May 2017 and October 2017. At its initial stages, it was important to get familiar with basic Android application development, ProGuard for optimizing Android applications and reflection. Consequently, it was essential to understand how the Doop framework performs static analysis, and how to use it in order to produce ProGuard specifications. Yet, the most important part was to experiment with the various configurations produced by Doop framework, besides the default configurations proposed by Android, and application specific ones.

# 1. INTRODUCTION

In the context of Computer Science, the task of analyzing a program aids in inferring information regarding every execution of the program, without actually executing it. Pointer or Points-to Analysis [1] is a static program analysis that aims for the computation of an approximation of the set of objects that a variable or expression may point to. Doop [2] is a static Analysis framework that performs Pointer Analysis for Java programs, declaratively using the Datalog language.

Optimization tools, such as the ProGuard optimizer [3], encapsulate a set of analyses or may utilize other static analysis frameworks' results, in order to transform a program for the best possible performance and system resources utilization. This is a challenging task if dynamic language features like reflection are used within a program, because of the static nature of the analyses. Modeling all possible behaviors of reflection operations is not practically achievable in the context of static analyses. Thus static analysis frameworks attempt to approximate the modeling of these behaviors or consider reflection to be absent for their analyses [4].

In this thesis, we present an approach for enriching a ProGuard configuration with Doop's static analysis results, along the lines of automating the production of such configurations, for Android applications, by considering the set of reachable methods of a program.

The rest of the thesis is organized as follows:

1. In Chapter 2 we give an overview of Android applications, ProGuard, Reflection, and Pointer Analysis in the Doop framework.
2. In Chapter 3 we describe how the Doop framework was used in order to produce specifications for ProGuard configuration.
3. In Chapter 4 we present our experimental evaluation.
4. In Chapter 5 we summarize our conclusions.

# 2. BACKGROUND

## 2.1 Android

Android[5] is an operating system developed and released by Google, in September 2008. It is based on the Linux Kernel, and it is primarily used for mobile devices. Android gives developers the chance to design and develop their own applications, using the Android Software Development Kit (SDK), and primarily the Java programming language, although there is support for a variety of other programming languages running on the Java Virtual Machine (JVM). Contrary to the Android SDK, the Android Native Development Kit (NDK), offers the opportunity to developers to write code in the C or C++ programming languages, which is finally translated to native code. Android applications can be developed in Android Studio, which is the official Android Integrated Development Environment and is equipped with every tool offered by the Android SDK or the Android NDK tool sets.

The Android SDK comes with plenty of tools that can enhance the application development and testing process. The Android Debug Bridge (ADB) [6] and UI/Application Exerciser Monkey[7] are some prominent tools aiding in the development process. Monkey generates streams of user events, as well as system ones, and can be configured by the developer. The Android SDK is responsible for compiling Java code along with any other associated files, such as resource files into an Android Package Kit (APK), which is a type of archive file, used by Android for the installation of mobile applications. An APK is quite similar to the Java Archive (JAR) package file. A typical APK file, consists of the following:

```
1   META-INF directory:
2           MANIFEST.MF: the Manifest file.
3           CERT.RSA: The certificate of the application.
4           CERT.SF: The list of resources and SHA-1 digest of the
                corresponding lines in the MANIFEST.MF file.
5   lib: the directory containing the compiled code that is specific to a
        software layer of a processor. The directory is split in more
        directories within it:
6         armeabi: compiled code for all ARM based processors only.
7         armeabi-v7a: compiled code for all ARMv7 and above based processors
              only.
8         arm64-v8a: compiled code for all ARMv8 arm64 and above based
              processors only[7][8].
9         x86: compiled code for x86 processors only.
10        x86_64: compiled code for x86 64 processors only.
11        mips: compiled code for MIPS processors only.
```

**Figure 1: APK file contents**

```
1        res: the directory containing resources not compiled into resources.arsc.
2        assets: a directory containing applications assets, which can be
             retrieved by AssetManager.
3        AndroidManifest.xml: An additional Android manifest file, describing the
             name, version, access rights, referenced library files for the
             application, which may be in Android binary XML.
4        classes.dex: The classes compiled in the dex file format understandable
             by the Dalvik virtual machine and by the Android Runtime.
5        resources.arsc: a file containing precompiled resources, such as binary
             XML for example.
```

**Figure 1: APK file contents**

The source code of Android applications, usually written in Java and compiled to Java bytecode for the JVM, is translated to Dalvik bytecode and stored in Dalvik Executable (DEX) files, for the Dalvik Virtual Machine, which is a virtual machine supporting the execution of Dalvik Executable files. Since the release of Android 5.0 ”Lollipop” version, the Dalvik VM has been completely replaced by the Android Runtime (ART) that executes DEX files as well as the Dalvik VM. Unlike Dalvik VM, which used Just-in-Time (JIT) compilation, ART introduced Ahead-of-Time (AOT) compilation, and improved Garbage Collection (GC) mechanisms. JIT compilation is a technique added to the VMs, in order to compile code to native code dynamically at the execution time, and allow the optimization of the application based on its execution. AOT compilation does the same only once, when the application is installed.

The four main components of an Android application are activities, services, broadcast receivers and content providers. Activities are the entry points responsible for user interaction, by presenting a user interface to the user. Each activity is independent of the others, however many activities are bound together towards a better user experience. A service does not provide any user interface, and it is a component responsible for keeping an application running, while in the background. A broadcast receiver is a component via which events are delivered to the application from the system, allowing the application to respond to system-wide broadcast messages. A content provider, typically manages the application data that can be stored in any persistent storage location, and through it other applications can request to query or modify the data of another application. Generally, every application component is declared in the AndroidManifest.xml file.

Even though mobile storage is constantly expanding, there is always a requirement for more lightweight applications, in terms of APK file size and execution performance. In order to achieve this, many tools can be used, and one of these is the ProGuard optimizer for Java bytecode.

## 2.2 ProGuard

ProGuard is the open source Java bytecode optimizer, officially used in Android application development in order to create smaller and faster APKs. ProGuard is integrated in the Android SDK tools, and, in order to enable ProGuard usage in the building process of an Android application, one has to specify it in the corresponding Gradle build script. The Gradle [8] build tool is the default building mechanism used in Android application development, and is integrated in the Android Studio.

A ProGuard pipeline consists of four basic steps: the shrinking step, the optimization step, the obfuscation step and the preverification step, which is not of interest in the case of Android applications. Each step of the ProGuard pipeline is optional, and thus can be skipped by defining the appropriate rule in the configuration file.

In the shrinking step, ProGuard detects classes, fields, methods and attributes which are not used and there is no need for them to be kept, and removes them. The shrinking step is the most important, since the application size can be reduced by a lot, by the elimination of dead code.

The optimization step of ProGuard performs optimizations on the bytecode level of the methods, based on the results of the static analysis it performs. There is a list of optimizations that can be enabled, based on the above analyses. One can specify which optimizations can be performed or not, by providing an ”-optimizations” rule in the configuration file with a filter of optimizations, leading to a more efficient application in terms of APK size and execution performance. These optimizations are based on the results of control flow analysis, data flow analysis, partial evaluation, static single assignment, global value numbering and liveness analysis. Furthermore, the optimizations may lead to better results depending on the application code and the virtual machine that the application is executed on.

The obfuscation step, is where ProGuard transforms classes, fields, and methods names to trivial names. Other than reducing the size of the final APK to a greater extent, obfuscation is important because it leads to an APK which is harder to reverse engineer, making it difficult to get the initial application source code.

ProGuard decides what should be kept or discarded, based on the entry points declared on the respective keep specifications of the configuration files, where entry points are classes, class members, packages, etc. Each step takes into consideration the specified entry points, in order to perform its work. Entry points are not shrunk or obfuscated, unless explicitly declared. The entry points ”keep” specifications are the following:

```
1          -keep [,modifier, ...] class_specification : specifies the classes and
               their members to be kept as entry points.
2          -keepclassmembers [,modifier,...] class_specification : specifies the
               members to be preserved if their classes are preserved as well.
3          -keepclasseswithmembers [,modifier,...] class_specification :
               specifies the entry points to be preserved, if and only if the
               declared members are preserved.
4          -keepnames class_specification : Specifies class names not to be
               obfuscated, if they are not removed in the shrinking phase
5          -keepclassmembernames class_specification : Specifies class members'
               names not to be obfuscated if they are not removed in the
               shriniking phase
6          -keepclasseswithmembernames class_specification : Specifies classes'
               and class members' names to be preserved, on the condition that
               each specified class member is preserved
7              after the execution of the shrinking step
```

**Figure 2: ProGuard keep specifications**

The above specifications, mention a modifier and a class_specification. The keep modifiers are the following:

```
1       allowshrinking : This modifier specifies that the entry point specified
            in the -keep rule may be shrunk, even if it has to be preserved.
2       allowoptimization : This modifier specifies that the entry point
            specified in the -keep rule may be optimized.
3       allowobfuscation : This modifier specifies that the entry point specified
            in the -keep rule may be obfuscated.
```

**Figure 3: Keep modifiers**

A class_specification on the other hand, is a template used in the -keep rules in order to declare the entry points. The template syntax is quite similar to the Java syntax of a class or interface specification.

```
[@annotationtype] [[!]public|final|abstract|@ ...] [!]interface|class|enum
    classname
    [extends|implements [@annotationtype] classname]
    [{
        [@annotationtype]
            [[!]public|private|protected|static|volatile|transient ...]
            <fields> | (fieldtype fieldname);
        [@annotationtype]
            [[!]public|private|protected|static|synchronized|native|abstract|
        strictfp ...] <methods> | <init>(argumenttype,...) |
            classname(argumenttype,...) | (returntype
            methodname(argumenttype,...));
        [@annotationtype] [[!]public|private|protected|static ... ] *;
        ...
    }]
```

**Figure 4: ProGuard class specification**

In order to use ProGuard in Android application development, one has to declare it into the appropriate build type on the application's Gradle Build script, by adding the line minifyEnabled true. An example of enabling ProGuard in debug mode build, is the following:

```

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

**Figure 5: ProGuard gradle script sample**

The proguardFiles option declares the ProGuard rules to be used by ProGuard. The Android SDK comes with two files, the default ProGuard files, 'proguard-android' and 'proguard-android-optimize'. The default 'proguard-android' file consists of the following rules:

```
-dontusemixedcaseclassnames
-dontskipnonpubliclibraryclasses
-verbose

-dontoptimize
-dontpreverify

-keepattributes *Annotation*
-keep public class com.google.vending.licensing.ILicensingService
-keep public class com.android.vending.licensing.ILicensingService

-keepclasseswithmembernames class * {
   native <methods>;

}

-keepclassmembers public class * extends android.view.View {
   void set*(***);
     *** get*();

}

-keepclassmembers class * extends android.app.Activity {
   public void *(android.view.View);

}

-keepclassmembers enum * {
    public static **[] values();
       public static ** valueOf(java.lang.String);

}

-keepclassmembers class * implements android.os.Parcelable {
     public static final android.os.Parcelable$Creator CREATOR;

}

-keepclassmembers class **.R$* {
       public static <fields>;

}

-dontwarn android.support.**

-keep class android.support.annotation.Keep

-keep @android.support.annotation.Keep class * {*;}

-keepclasseswithmembers class * {
       @android.support.annotation.Keep <methods>;

}
```



**Figure 6: Default ProGuard Android configuration**

```
1
2     -keepclasseswithmembers class * {
3           @android.support.annotation.Keep <fields>;
4
5     }
6
7     -keepclasseswithmembers class * {
8           @android.support.annotation.Keep <init>(...);
9
10    }
```

**Figure 6: Default ProGuard Android configuration**

The provided default configuration of ProGuard for Android applications keeps annotations, the ILicensingService class, and every class that contains native methods. Furthermore, it keeps the setter and getter methods of every class extending the View class, as well as every method of an activity class with an input View parameter. It also keeps values and valueOf methods of enumeration classes, and the Creator fields of the Parcelable class. Finally, every static field of inner auto generated R classes and the Keep class as well as methods, fields and constructors annotated with Keep are not shrunk or obfuscated.

The default configuration file used in order to optimize applications consists of the same rules, but, instead of the -dontoptimize rule, the following are added:

```
1     -optimizations
          !code/simplification/arithmetic,!code/simplification/cast,!field/*,
2     !class/merging/*
3     -optimizationpasses 5
4     -allowaccessmodification
```

**Figure 7: Default Android optimization configuration**

In the figure above, it is declared that ProGuard should perform every optimization but for optimizations for arithmetic instructions, casting operations, field optimizations and class merging, and that the maximum passes of optimizations should be five. Finally, it is specified that access modifiers of every class and class members may be modified.

After each build, ProGuard generates four files, which are described as follows:

```
1     A dump file, which describes the structure of all the class files.
2     A mapping file, which maps each initial class, method and field name to the
          equivalent obfuscated one.
3     A seed file, listing the entry points.
4     A usage file, listing the removed code.
```

**Figure 8: ProGuard output files**

The default rules are usually sufficient, though each application may require further configuration, especially in the case that library code or dynamic features of the programming language are used within the application code. This is when application-specific rules may be required, and the developer has to provide ProGuard with extra configuration directives and be familiar with the application's source code. This is rather impossible in the case of closed-source libraries that do not provide their own library-specific ProGuard files, thus there is a need for analyzing the library code for better configuration.

## 2.3 Reflection

Reflection is a dynamic language feature that allows the modification of a program's structure and behavior dynamically. With reflection, someone can get information about a class, discover its public and private members, instantiate a new object, or invoke a method at runtime, even lacking static knowledge of its existence.

In Java, reflection can be used via the java.lang.reflect and java.lang.Class packages. Consider the following example, where, provided with a class name in java.lang.String format, we want to invoke a method named foo, if such exists. In order to accomplish this, one would get a java.lang.Class object, then create a new object of the given class, and finally get a java.lang.reflect.Method object representing the method, and invoke it with the newly created object. In terms of Java, this should be done as follows:

```
1  String className = ...; // possibly a constant string
2  Class myClass = Class.forName(className);
3  Object myObject = myClass.newInstance();
4  String methodName = ...; //possibly a constant string
5  Method myMethod = myClass.getMethod(methodName, ...);
6  myMethod.invoke(myObject, ...);
```

**Figure 9: Common reflection pattern**

Reflection use occurs in the case of Android applications as well. A solid example for the need of reflection use is version and device compatibility. After each Android update, a class could be removed, so, to check whether it still exists, one can use reflection.

Despite the fact that reflection is quite useful, it can also result in problems, when analyzing or optimizing a program. In such a scenario, the developer should specify a class, or a class member that is invoked dynamically, to be kept as an entry point, in the ProGuard configuration. ProGuard can handle some basic reflection usage, but the developer may have to manually specify what should be kept, which is a rather unpleasant task, especially when reflection is used within a library that the developer is not familiar with. That is where whole-program static analysis tools can help.

## 2.4 Points-to Analysis and the Doop Framework

Datalog is a declarative logic programming language, which among other, has found use as a query language for deductive databases. In Datalog, computation consists of monotonic logical inferences that apply to produce new results until a fixpoint is reached.

Doop is a static analysis framework for pointer or points to analysis of Java programs, that implements a range of algorithms for pointer analysis, including context insensitive, call-site sensitive and object sensitive analyses. The most defining feature of the Doop framework, is the use of Datalog for its analyses.

Datalog has proven to be a great fit for the domain of program analysis, and has been extensively used both for low-level and for high level analyses. The ability of Datalog to define recursive relations solves the problem of mutual recursion, which is the source of complexity in program analysis.

At first, Doop used a commercial Datalog engine, developed by LogicBlox Inc. Doop currently uses an open source Datalog implementation, called Soufflé [9], which compiles a Datalog program to a native C++ program.

Doop, at its core, uses the Soot framework for the preprocessing step that takes as an input the bytecode of a Java program, and generates the input facts to be used for an analysis, in the form of relations. Due to this, there is no need for source code for the framework to perform its analysis, which is important if we take into consideration that libraries whose code is not open source can be analyzed as well. The relations that are generated from the input program are also known as EDB (Extensional Database) predicates, in Datalog terminology.

Once the preprocessing step is finished, and the facts are generated, a simple pointer analysis can be expressed in Datalog as a transitive closure computation, in the following form:

```
1    VarPointsTo(?heap, ?var) <- AssignHeapAllocation(?heap, ?var).
2    VarPointsTo(?heap, ?to) <- Assign(?to, ?from), VarPointsTo(?heap, ?from).
```

**Figure 10: Datalog pointer analysis : VarPointsTo(?heap, ?var)**

The two rules shown in the above figure are known as IDB (Intensional Database) rules and are used for the derivation of new facts. The first rule constitutes the base of the computation, and states that upon the assignment of an allocated heap object to a variable, this may point to the heap object. The second rule states that, upon the assignment of a variable to another, the latter may point to any object that the former may point to.

Pointer analysis of static analysis frameworks attempts to compute a precise representation of a program's heap, by recursively computing the set of objects a variable may point to. Static handling of various language features, especially dynamic ones, like reflection,

may produce unsound results: the analysis will miss possible runtime behaviors. As a consequence, many static analysis frameworks consider such features to be absent for their analyses. Reflection statements can occur at any location of the program, thus creating the requirement for a whole program analysis in order to acquire an understanding of the program's heap, for a more precise reflection tracking. Doop's reflection handling is based on the fact that points-to and reflection analysis can be combined, as described in [4].

Static analysis frameworks of Java programs, usually start their computations from the main method of the program's main class. However, this is not the case for Android applications, where no such method exists. Doop supports the analysis of Android applications, by discovering the application's components and the UI elements described in the application's XML configuration files, in order to start its points-to computations from the multiple entry points those components may have [10].

# 3. KEEP SPECIFICATIONS GENERATION IN DOOP

The Doop framework's static analysis,computes the set of the Reachable methods of a program, among others, which constitutes the base of our Datalog KeepMethod rule for producing ”keep” method specifications for ProGuard, and is as follows:

```
1        .decl KeepMethod(?m:Method)
2        .output KeepMethod
3
4        KeepMethod(cat("-keepclassmembers class ", cat(?type, cat(" { ",
             cat(?retType, cat(" ", cat(?simpleName, cat(substr(?descriptor,
             strlen(?retType), strlen(?descriptor)), "; }")))))))) :-
5            Reachable(?m),
6            MethodLookup(?simpleName, _, _, ?m),
7            Method_Descriptor(?m, ?descriptor),
8            Method_SimpleName(?m, ?simpleName),
9            Method_DeclaringType(?m, ?type),
10           Method_ReturnType(?m, ?retType).
```

**Figure 11: Keep method specification rules**

The above rules state that for every reachable method in the program, a string constant of the form ”-keepclassmembers class ?type  ?retType ?descriptor; ” should be produced, leading to the creation of a KeepMethod file, listing all the -keepclassmembers rules for the reachable class methods. The MethodLookup, Method_Descriptor, Method_SimpleName, Method_DeclaringType and Method_ReturnType predicates are used for matching the method and get its name, its return type, its parameters and finally the class it belongs to. Within just a few lines of Datalog, we manage to have a complete ProGuard configuration file, with a solid set of keep rules, at the end of Doop's analysis. A sample of a KeepMethod file for the Material Calculator application, is the following:

```
1    -keepclassmembers class java.lang.reflect.Array { java.lang.Object
         newInstance(java.lang.Class,int[]); }
2    -keepclassmembers class java.lang.ThreadGroup { void <init>(); }
3    -keepclassmembers class java.lang.Thread { void
         <init>(java.lang.ThreadGroup,java.lang.Runnable); }
4    -keepclassmembers class java.security.PrivilegedActionException { void
         <init>(java.lang.Exception); }
5    -keepclassmembers class java.lang.Object { java.lang.Object clone(); }
```

**Figure 12: Calculator KeepMethod file**

The produced ”keep” method rules, are combined with a set of standard rules, proposed by Android, thus leading to the creation of concrete ProGuard configurations, ready to be integrated in the building process. The set of standard rules used are described in the following figure:

```
1         -dontusemixedcaseclassnames
2         -dontskipnonpubliclibraryclasses
3         -verbose
4
5         -dontoptimize
6         -dontpreverify
7
8         -dontnote
9         -ignorewarnings
```

**Figure 13: Doop configuration standard rules**

# 4. EXPERIMENTAL EVALUATION

In this section, we present our experimental evaluation of the analyses performed by the Doop framework for different open source APKs, available on GitHub[11], as well as the experiments performed for the various ProGuard configurations generated by the Doop framework, in contrast to the default Android configuration. Thus we consider the following four possible cases of ProGuard configuration for each application: ProGuard disabled, ProGuard default Android configuration, ProGuard Doop configuration and finally ProGuard Doop configuration with reflection handling. For each configuration, we test the application with the Android Monkey tester tool.

**Table 1: Doop keep method specification sizes in terms of keep rules**

| APKs | Doop default | Doop with reflection handling |
|---|---|---|
| Material Calculator | 22750 | 37996 |
| Gson Android Example | 21249 | 34017 |
| Mintube | 23546 | 34837 |
| Radiodroid | 27745 | 44317 |
| Reddinator | 28438 | 43286 |

The default Android configuration, as well as the Doop configurations with/without reflection handling were sufficient for the APK size to be reduced and to be able to be executed in most of the applications tested, though in some cases there was a need for ignoring the obfuscation step. Doop configurations outperformed the default one for most applications, but for a few that the APK size reduction was the same. As shown in Table 1, Doop's reflection handling adds way more keep rules to the configuration, considering that the set of all reachable methods is inluded in ”keep” clauses.

**Table 2: Doop analysis execution time (seconds) for keep method specifications generation**

| APKs | Doop default | Doop with reflection handling |
|---|---|---|
| Material Calculator | 214 | 730 |
| Gson Android Example | 161 | 491 |
| Mintube | 147 | 542 |
| Radiodroid | 180 | 1031 |
| Reddinator | 213 | 1040 |

For the generation of the keep method specifications, several analyses had to be executed with the Doop framework on a server consisting of Intel(R) Xeon(R) CPU E5-2687W v4 @ 3.00GHz CPUs, and 512GB RAM. Analyses were executed using the Android SDK 25 platform. In Table 2, Doop's analysis execution times are presented, where analyses handling reflection are way slower from those that do not, as expected. Finally, applications were tested with the Monkey tester with a setup of 1000 pseudorandom events, with an emphasis on activity launches.

We start our experiments presentation with the Material Calculator, Gson Android example and Mintube applications in the first subsection, followed by Radiodroid and Reddinator on separate subsections. For our evaluation, we compare the APK size reduction and the number of library and program classes, for each of the four configurations, and whether the APK could be executed after ProGuard shrinking/obfuscation.

## 4.1 Material Calculator, Gson Android example, Mintube cases

We start by presenting the APK sizes, number of classes with/without ProGuard and whether the execution of the three of these applications was successful for each configuration, in the following table.

**Table 3: Material Calculator**

| APK and Configuration | APK size | Library Classes | Program Classes | Monkey |
|---|---|---|---|---|
| Calculator no ProGuard | 2.8MB | 4237 | 2729 | OK |
| Calculator default configuration | 1.8MB | 1079 | 1189 | OK |
| Calculator Doop configuration | 1.7MB | 1079 | 1135 | OK |
| Calculator reflection configuration | 1.8MB | 1079 | 1338 | OK |

**Table 4: Gson Android example**

| APK and Configuration | APK size | Library Classes | Program Classes | Monkey |
|---|---|---|---|---|
| Gson ex. no ProGuard | 1.5MB | 4590 | 1717 | OK |
| Gson ex. default configuration | 804KB | 1068 | 551 | OK |
| Gson ex. Doop configuration | 567KB | 1068 | 500 | OK |
| Gson ex. reflection configuration | 833KB | 1068 | 695 | OK |

**Table 5: Mintube evaluation**

| APK and Configuration | APK size | Library Classes | Program Classes | Monkey |
|---|---|---|---|---|
| Mintube no ProGuard | 1.6MB | 3880 | 1977 | OK |
| Mintube default configuration | 930KB | 1024 | 1007 | OK |
| Mintube Doop configuration | 918KB | 1024 | 1065 | OK |
| Mintube Doop reflection | 935KB | 1024 | 1109 | OK |

The Doop configurations mentioned in Table 2 are as produced by the Doop framework's analysis, enriched with the standard rules defined in Figure 13. Doop normal configurations seem to be clearly better than the default configuration, but for the Doop reflection configuration which outputs slightly bigger APKs. For the case of Gson Example, the APK size reduction is significant with Doop's configuration, though this is an expected result since the application is quite simple.

## 4.2 Radiodroid case

The execution of Radiodroid using the Monkey tester failed for both the default and Doop configuration, as shown in Table 6, since the application crashed, though it was successful for the case of building and testing the application with the Doop reflection configuration file. The final APK was shrunk and obfuscated over the initial one with a significant reduction of its size, thus establishing the fact that Doop's reflection handling can be decisive for the analysis and optimization of an Android application with reflection usage, whereas Android's default configuration could not model it, leading to a failed execution of the application.

**Table 6: Radiodroid evaluation**

| APK and Configuration | APK size | Library Classes | Program Classes | Monkey |
|---|---|---|---|---|
| Radiodroid no ProGuard | 2.7MB | 4237 | 3229 | OK |
| Radiodroid default configuration | 1.7MB | 1217 | 1918 | CRASH |
| Radiodroid Doop configuration | 1.6MB | 1217 | 1979 | CRASH |
| Radiodroid reflection configuration | 1.7MB | 1217 | 2054 | OK |

## 4.3 Reddinator case

In the case of Reddinator, neither of the ProGuard configurations was sufficient, to generate an APK that would not crash, but for the case of Doop's reflection configuration with the obfuscation step disabled.

**Table 7: Reddinator evaluation**

| APK and Configuration | APK size | Library Classes | Program Classes | Monkey |
|---|---|---|---|---|
| Reddinator no ProGuard | 3.4MB | 4237 | 2661 | OK |
| Reddinator default configuration | 2.6MB | 1231 | 1302 | CRASH |
| Reddinator Doop configuration | 2.3MB | 1231 | 1322 | CRASH |
| Reddinator reflection configuration | 2.4MB | 1231 | 1586 | CRASH |
| Reddinator reflection no obfuscation | 2.7MB | 1231 | 1586 | OK |

The application would crash because of a class not found exception. Preventing the obfuscation step leads to the successful execution of the application with Doop's reflection configuration for ProGuard, with a small increase of the APK size to 2.7MB in contrast to the unobfuscated one, whereas the other two configurations produced APKs that still crash. Reddinator proves yet again the power of Doop's configuration with reflection handling for ProGuard, which with minor modification lead to the successful execution of Reddinator.

Based on our experimental results, we conclude that the Doop framework can be used to generate configurations that lead to the production of either smaller or more correct APKs in the sense of avoiding application crashes, with possibly minor or no modification.

# 5. CONCLUSIONS

Program optimization is a valuable step for developing faster and more efficient applications. Static analysis constitutes a valuable asset for superior optimizations. The ProGuard optimizer is a must-use tool towards this step, and just by enabling its shrinking step we can get significant results in terms of APK size reduction. However its poor reflection handling may result in application crashes, thus requiring the developer to manually declare a set of rules for what has to be kept for the application to be able to execute. The integration of more sophisticated static analysis with better reflection handling in ProGuard configurations may add to ProGuard's optimization power, while automating the generation of such configurations can be easily done with a few declarative rules, as we presented in this thesis. Doop's simple keep method configurations outperformed the Android default configuration in most of the applications tested, especially for the cases of reflection usage within the application.

# ACRONYMS AND ABBREVIATIONS

| | |
|---|---|
| SDK | Software Development Kit |
| NDK | Native Development Kit |
| APK | Application Package |
| JVM | Java Virtual Machine |
| VM | Virtual Machine |
| ART | Android Runtime |
| JIT | Just-in-Time |
| AOT | Ahead-of-Time |
| GC | Garbage Collection |
| API | Application Programming Interface |
| EDB | Extensional Database |
| IDB | Intensional Database |

# REFERENCES

[1] ”Pointer Analysis” [Online]
Available: http://yanniss.github.io/points-to-tutorial15.pdf

[2] ”Doop: Framework for Java Pointer Analysis” [Online]
Available: http://doop.program-analysis.org/

[3] ”ProGuard : The open source optimizer for Java bytecode” [Online]
Available: https://www.guardsquare.com/en/proguard

[4] Smaragdakis Y, Balatsouras G, Kastrinis G, Bravenboer M, ”More Sound Static Handling of Java Reflection.” 13th Asian Symposium on Programming Languages and Systems (APLAS 2015)

[5] ”Android Operating System” [Online]
Available: https://www.android.com

[6] ”Android Debug Bridge” [Online]
Available: https://developer.android.com/studio/command-line/adb.html

[7] ”UI/Application Exerciser Monkey” [Online]
Available: https://developer.android.com/studio/test/monkey.html

[8] ”Gradle Build Tool” [Online]
Available: https://gradle.org/

[9] https://github.com/oracle/souffle/wiki [Online]
Available: https://github.com/oracle/souffle/wiki

[10] Neville Grech and Yannis Smaragdakis. 2017. P/Taint: Unified Points-to and Taint Analysis. Proc. ACM Program. Lang. 1, OOPSLA, Article 102 (October 2017), 28 pages.

[11] ”GitHub” [Online]
Available: https://github.com