Universidad de los Andes

# Comparison and Analysis Between Automatic Exploration Tools for Android Applications

*Author:*
Michael OSORIO-RIAÑO

*Advisor:*
Mario LINARES-VÁSQUEZ

*A thesis submitted in fulfillment of the requirements for the degree of Bachelor in Software and Computer Engineering*

*in*

THE SW DESIGN LAB

Systems and Computing Engineering Department

June 24, 2020

# Abstract

Michael OSORIO-RIAÑO

*Comparison and Analysis Between Automatic Exploration Tools for Android Applications*

The amount of android applications is having a tremendous increasing trend, leading the mobile software market to exert pressure over practitioners and researchers about several topics like application quality, frequent releases, and quick fixing of bugs. Because of this, mobile app development process requires of improving the release cycles. Therefore, the automation of software engineering tasks has become a top research topic. As a result of this research interest, several automated approaches have been proposed to support software engineering tasks. However, most of those approaches that provide comprehensive results use source code as entry, which due to privacy factors imposes hard constraints on the implementation of those approaches by third-party services. Nevertheless, the market is leading practitioners to crowdsource/outsource software engineering tasks to third-parties that provide on-the-cloud infrastructures.

Solutions that rely on third-party services cannot use state-of-the-art automated software engineering approaches because practitioners only provide them with APK files. Therefore, approaches that work at APK level (i.e., do not require source code) are desirable to enable automated outsourced software engineering tasks. As an initial point, in this thesis we explore the possibility of performing automated software engineering tasks with APKs, and in particular we use mutation testing as a representative example. Our experiments show that mutation testing at APK level outperforms (in terms of time and amount of generates mutants) the same task when conducted at source code level.

# Acknowledgements

Primero agradecir a Mario por dejarme entrar al grupo, por aceptarme como pupilo, ayudarme con las nuevas experiencias, asesorar mi tesis, por ayudarme con las dudas y por hacer este estudio posible con la ayuda de su conocimiento

Al grupo de TSDL por ayudarme y compartirme sus experiencias y conocimientos, los cuales fueron importantes para desarrollar este trabajo.

Especial agradecimiento a Camilo Escobar por ayudarme a con el proceso de desarrollo de InstruAPK, las bases, el concepto y parte de su implementación además de ayudarme con feedback sobre las gráficas, proceso de escritura y resolviendo dudas sobre otros conceptos de su dominio

También quiero agradecer a mi mamá y mi hermana por mantenerme motivado hasta el último momento, por su apoyo inocndicional y por estar siempre cuando las necesité.

A mi amigos por compartirme sus conocimientos que también hicieron posible este estudio.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Parafrasear lo que dice aquí, decir que este gran número de aplicaciones por fortuna a llevado a un gran número de estudios para mejorar la cobertura de código que se logra durante las pruebas automáticas, y también cubir diferentes estados del celular como modo avión con carga sin carga y demás, dada la gran cantidad de herramientas, los desarrolladores pueden sentirse sobrecargados, pueden exitir muchas opciones y pueden que no se elija la mejor herramienta. Los desarrolladores necesitan formas de elegir la mejor herramienta que se adapte lo mejor posible a sus necesidades.

Mobile markets have pushed and promoted the raising of an interesting phenomenon that has permeated not only developers culture, but also human beings' daily life activities. Mobile devices, apps, and services are helping companies and organizations to make "digital transformation" possible through services and capabilities that are offered ubiquitously and closer to the users. Nowadays, mobile apps and devices are the most common way for accessing those services and capabilities; in addition, apps and devices are indispensable tools for allowing humans to have in their phones, computational capabilities that make life better and easier.

The mobile apps phenomenon has also changed drastically the way how practitioners design, code, and test apps. Mobile developers and testers face critical challenges on their daily life activities such as (i) continuous pressure from the market for frequent releases of high quality apps, (ii) platform fragmentation at device and OS levels, (iii) rapid platform/library evolution and API instability, and (iv) an evolving market with millions of apps available for being downloaded by ends users [**joorabchi2013real**, **palomba2018crowdsourcing**]. Tight release schedules, limited developer and hardware resources, and cross-platform delivery of apps, are common scenarios when developing mobile apps [**joorabchi2013real**].

Therefore, reducing the time and effort devoted to software engineering tasks while producing high quality mobile software is a "precious" goal.

Both practitioners and researchers, have contributed to achieve that goal, by proposing approaches, mechanisms, best practices, and tools that make the development process more agile. For instance, cross-platform languages and frameworks (e.g., Flutter, Ionic, Xamarin, React Native) contribute to reducing the development time by providing developers with a mechanism for building Android and iOS versions of apps in a write-one-run-anywhere way [**joorabchi2013real**, **fazzini2017automated**]. Automated testing approaches help testers to increase the apps' quality and reduce the detection/reporting time [**choudhary2015automated**, **kochhar2015understanding**, **linares2017continuous**]. Automated categorization of reviews also helps developers to select relevant information, issues, features and sentiments, from large volume of review that are posted by users [**palomba2018crowdsourcin** **villarroel2016release**, **di2016would**]. Moreover, approaches for static analysis, are helping developers to early detect different types of bugs and issues that without the automated support could be time consuming for developers — when doing the analysis manually [**li:IST2017**]. Both static and dynamic analyses have been used with the aforementioned approaches, with a special preference for static analysis on source code.

The developers community is quickly moving towards using cloud-services and crowd-sourced services for software engineering tasks [**Leicht2017IEEESoftware**, **stol2017crowdsourcing**]; using those services is becoming a common practice of mobile developers who want to reduce costs and the time devoted for an activity. For example, the Firebase Test Lab platform [**firebase**] provides automated testing services, in particular, it automatically executes/explores a given app (provided by the developer as an Android APK file), and reports crashes found on a devices matrix that is selected by the user. However, the lack of knowledge of source code internals imposes a limitation on the usefulness and completeness of the results reported back to the users.

## 1.1 Problem Statement

The power and usefulness of a large number of state-of-the-art approaches for automated software engineering of Android apps rely on the existence of source code for extracting intermediate representations or models that drive the analysis execution or the artifacts generation. For example, Zaeem *et al.*[**zaeem2014automated**] instrumentates the source code in order to record the view flow followed through a set of instructions in order to validate oracles. However, existing approaches that rely on source code for supporting automated software engineering tasks are untenable in a commercial environment where practitioners outsource software engineering tasks, but without releasing the source code (i.e., the services work directly on executable files).

Any type of analysis that relies only on executable files (i.e., dynamic analysis) is known to be limited when compared to static analysis that can be done directly on the code [**spathoulas2014assessing**]. For example, Firebase Test Lab enables automated tests based on Random GUI input generation. However, this test provides low usage case coverage. Additionally, for different legal and organizational reasons (e.g., source code contains a company's exclusive implementation of an algorithm, source code contains keys/secrets for services, etc.) the app's source code is often not available, making it difficult to enable cloud/crowd-based services that use state-of-the-art approaches.

Furthermore, as it will be shown in the Section 2, decompilation/compilation process is a cumbersome process due to the wide spectrum of developing possibilities that defines how each application is build. Software Engineering approaches that rely on source code modifications/instrumentation, and then require to build/compile the app are expensive; as is the case of mutation testing [**appelt2014automated**, **linares2017enabling**, **praphamontripong2016experimental**, **rodriguez2018mutode**].

## 1.2 Thesis Goals

//TODO

## 1.3 Thesis contribution

//TODO

## 1.4 Document Structure

// TODO Hacer al final porque no se sabe la estructura antes de

# Chapter 2

# Context

## 2.1 Mutation Testing

Mutation testing is a testing technique which consists on modifying an application (by injecting bugs) in order to enhance and evaluate the quality of the test suite that accompanies it. Each injected bug generates a new version of the application, and its called **mutant**. Each mutant differs from the original version in a simple modification, called **mutation**. As an example, if there is a compound logical operation in the original version, a valid mutation consists in replacing one and only one of the logical operators in the compound operation (i.e. if there is an "AND", the mutation changes it with "OR"). It is worth noticing, that the replacement of each operator in the compound operation generates one mutant.

Now, to determine how many mutants can be generated, mutation testing uses a set of rules called **mutation operators** that define common errors and practices that belong to programming rules (e.g., replace a math operator ), or to the specific programming language context (e.g., assign a null value to an Intent parameter ). Therefore, depending on the tool used to generate the mutants, the application to be mutated is analyzed to derive a Potential Fault Profile [**linares2017enabling**, **Moran:ICSE18**], i.e., a set of possible locations where a mutant operator can be applied. For each of this locations the mutation tool creates a copy of the original app, and this copy is a modification produced by applying a mutation operator. Several mutation operators have been proposed for different types of applications such as web apps [**praphamontripong2016experimental**], data-centric apps [**appelt2014automated**], NodeJS packages [**rodriguez2018mutode**] and Android apps [**linares2017enabling**]. Table 2.1 summarizes different implementations of mutation testing in several programming languages and for different types of applications.

For more information about mutation testing you can refer to Chapter 5 of Paul Ammann and Jeff Offutt book called Introduction to Software Testing [**ammann2016introduction**] or the survey by Jia and Harman [**Jia:TSE11**].

TABLE 2.1: Mutation testing papers for different application types.

| App type/Language | Paper |
|---|---|
| Java | Yu-Seung Ma, Yong Rae Kwon, and Jeff Offutt. 2002. ***Inter-Class Mutation Operators for Java.*** In 13th International Symposium on Software Reliability Engineering (ISSRE'02) |
| Python | Anna Derezinska and Konrad Halas. 2014. ***Analysis of Mutation Operators for the Python Language.*** Proceedings of the Ninth International Conference on Dependability and Complex Systems (DepCoS14) |
| Web Apps | Upsorn Praphamontripong, Je Outt, Lin Deng, and Jingjing Gu. ***An Experimental Evaluation of Web Mutation.*** ICSTW'16 |
| Data-Centric Apps | Dennis Appelt, Cu Duy Nguyen, Lionel C. Briand, and Nadia Alshahwan. ***Automated testing for SQL injection vulnerabilities: an input mutation approach.*** ISSTA'14 |
| GUIS | R. A. P. Oliveira, E. Alégroth, Z. Gao, and A. Memon. ***Definition and evaluation of mutation operators for GUI-level mutation analysis.*** ICSTW'15 |
| Test Data | Daniel Di Nardo, Fabrizio Pastore, and Lionel C. Briand. 2015. ***Generating Complex and Faulty Test Data through Model-Based Mutation Analysis.*** ICST'15 |
| Android | Mario Linares-Vásquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas and Denys Poshyvanyk. 2017. ***Enabling Mutation Testing for Android Apps***. ESEC/FSE 2017 |
| NodeJS | Diego Rodríguez-Baquero, and Mario Linares-Vásquez. 2018. ***Mutode: generic JavaScript and Node. js mutation testing tool***. ISSTA 2018 |

## 2.2 Android App Building Process

In order to understand some of the problems and design decisions in this thesis, it is important to know how android applications are built from the source code into an Android Application Package ( APK ), which is the file deployed to an Android device. It is worth noticing that Android applications can be developed in several ways, whether using first-class languages[1] as JAVA, Dart or Kotlin, or using frameworks for cross-platform development (e.g. Ionic, Phonegap, React4Native). Nevertheless, native approaches generates a project in a first-class language to be deployed in mobile devices. Therefore, for the purpose of this explanation lets assume the application to study was developed using a first-class (i.e., native) language.

Resource files (i.e., color.xml, strings.xml, etc.), application files (e.g, *.java) , libraries (i.e, jar files) and the Android manifest file are packaged in an APK file. The process starts with compiling the app source code to java bytecode, and then compiling bytecode and libraries to dex code. Dex code is the set of operations that are going to be executed by the VM in an Android device. Previously generated files along with the already compiled libraries are compiled using the *dx* tool into one or several .dex files[2]. Finally, packaged resource files and dex file(s) are packaged into an unsigned apk. So, while Java, Kotlin and Dart are high-level languages, dex operates as an intermediate representation that is closer to the machine.

## 2.3 Intermediate Representations (IR)

An intermediate representation is a representation of the source code aimed at being more expressive and easier of interpretation for a machine. An intermediate representation has the property that without losing app behavior, it presents the source code in a less complex format or in a less cumbersome context, from the point of view of a machine. In the specific case of Android apps, there are several intermediate representation; we will briefly describe 4 of the them, that have been widely used in research.: Java Bytecode, Dex, Smali, and Jimple. The first two are the closer representations to the starting and ending point of the Android application building process, while the other two (even being closer to the

[1]text

[2]For more information about multidex visit: `https://developer.android.com/studio/build/multidex?hl=es-419`

endpoint) have been created with the purpose of enabling analysis tasks on the source code.

One of the benefits of using intermediate representations (in the context of JAR and APK analysis) is avoiding the decompilation of the app to reach the original source code; in the specific case of Android apps, by using an intermediate representation we get rid of reversing the android building process.

### 2.3.1 JAVA Bytecode

This intermediate representation is used by the Java Virtual Machine (JVM). It is worth noting that JVM follows a stack-based architecture [**jvms**]. The most common way to get into this representation is through JAVA itself. However, there are compilers for other languages and frameworks into Java Bytecode. For example, SCALA, Clojure, Object Pascal, Kotlin and others [**pljvm**]. As presented previously, Java Bytecode is the first resulting representation when compiling an android application source code.

### 2.3.2 Dalvik Bytecode (DEX)

Originally designed to work with Dalvik VM (DVM) and the Android Runtime (ART), the dalvik bytecode was designed for systems as mobile devices that have restrictions in their computational power [**dvmi**, **jitcadvm**]. Actually, the Dalvik VM has been replaced by the Android Runtime (ART) keeping the same dalvik bytecode as representation to its execution. It is worth noting that both DVM and ART follow a register-based architecture that normally requires fewer, but typically more complex VM instructions.

### 2.3.3 JIMPLE

Jimple is an intermediate representation used by Soot [**soot**]. Jimple is based in 15 different operations that compared with the over 200 operations defined for Java bytecode makes it easier to optimize, however a lot of information can be lost when translated from java bytecode to Jimple.

### 2.3.4 SMALI

SMALI is an intermediate representation created by Ben Gruver [**smali**]. It offers a readable version of the Dalvik bytecode, due to the similar amount of operations supported for both representations. As it is closer to dalvik bytecode rather than Java bytecode it represent the code following a register-based architecture. This enhances our possibilities to analyze applications because all Java instructions tend to be separated from recursive calls. For example, if a new object is created inside the parameter field of a method, SMALI translates the object creation into one line and the method call into another as it can be seen in Figure **??**. Because of this, as it will be seen in next sections, we were able to recognize more potential fault injection points that are more complex to be recognized when analyzed in Java.

# Chapter 3

# Related work

## 3.1 Static Analysis of Android Packages

As part of the problem definition, we reviewed previous papers focused on static analysis of android apps at APK level, with the purpose of identifying the intermediate representations used by researchers. Therefore, we queried publications with the keywords: "smali code", "smali", "APK processing", "android apk" and "apk files", "android Smali". Most of the retrieved works focused mainly on security. As part of the results we found a paper called "Static Analysis of Android Apps A Systematic Literature Review" **[li:IaST2017]** wrote by Li *et. al.* Therefore, instead of conducting a new mapping study or literature review by ourselves, we relied on the work by Li *et. al.*.

The paper Li *et. al.*, in the authors words, *"provide a clear view of the state-of-the-art works that statically analyze Android apps, from which we [the authors] highlight the trends of static analysis approaches, pinpoint where the focus has been put, and enumerate the key aspects where future researchers are still needed"*. In particular, Li *et. al.*, conducted a systematic literature review using the search string shown in Table 3.1, which reports 124 papers and classifies them in 8 categories depicted in Figure **??**: (i) Private Data Leaks (46 papers), (ii) Vulnerabilities (40 papers), (iii) Permission Misuse (15 papers), (iv) Energy Consumption (9 papers), (v) Clone Detection (7 papers), (vi) Test Case Generation (6 papers), (vii) Cryptography Implementation Issues (3 papers) and (vii) Code verification (3 papers).

Note that the systematic literature review by Li *et. al.*, considers only papers up to 2015, and we recognize that since then several works could have been published. Our purpose with reviewing previous papers was to identify the existing intermediate representations and their characteristics. Therefore, not having a complete literature review until 2018 is not a limitation for our work. Future

TABLE 3.1: Keywords used by Li *et. al.*

| Line | Keywords |
|---|---|
| 1 | Analisis; Analyz*; Analys*; |
| 2 | Data-Flow; "Data Flow*"; Control-Flow; "Control Flow"; "Information-Flow*"; "Information Flow*"; Static*; Taint; |
| 3 | Android; Mobile; Smartphone*; "Smart Phone"; |

work, should be devoted to conduct a more up-to-date literature review that also includes previous papers that use dynamic analysis.

In the following, we briefly describe the task-related groups used by Li *et. al.*, and discuss some of the representative papers.

**Private Data Leaks.** This is the most frequent purpose reported in the papers categorized by Li *et. al.*,. In this group, FlowDroid is a representative example [**Arzt:2014**]; FlowDroid performs static taint analysis on android apps using flow-, context-, field-, object-sensitive and implicit flow-, lifecycle-, static-, alias-aware analysis. Therefore, FlowDroid has became a defacto tool used by researchers interested on finding privacy leaks in Android apps. For example, PCLeaks [**li:TrustCom2014**] goes one step further by performing sensitive data-flow analysis on top of component vulnerabilities, enabling not only issue identification but also data endangered. The most used intermediate representation for this category of papers is JIMPLE with 18 out of 46 papers, followed by SMALI with 8 papers.

**Vulnerabilities.** This category groups papers aiming at detecting vulnerabilities in Android apps. For instance, CHEX[**lu:CCS2012**] that detects potential component hijacking-based flows through reachability analysis on customized system dependence graphs and, Epicc [**octeau:Security2013**] and IC3 [**octeau:ICSE2015**] that implement static analysis techniques for implementing detection scenarios of inter-component vulnerabilities. The most used intermediate representation for this category of papers is SMALI with 15 out of 40 papers, followed by JIMPLE with 6 papers.

**Permission Misuse.** Permissions are one of the core elements of the Android security model. Malware applications try to use the permissions granted by the user to perform actions that do not correspond to the app features. Lin *et. al.* [**lin2014modeling**] conducted an study of permissions that users are most comfortable to grant, creating a set of privacy profiles, and in which way applications use those permissions. The most used intermediate representation for this category of papers is JIMPLE with 6 out of 15 papers, followed by SMALI with 4.

**Energy Consumption**. APIs and some hardware components have been demostrated as energy greedy elements in Android apps [**Linares-Vasquez:2014**, **Pathak:2011**], thus, analysis of energy consumption of mobile apps is becoming a hot topic. For instance, Li *et. al.* [**li:ISSTA2013**] present a tool to calculate source line level energy consumption through combining program analysis and statical modeling. The output of these analyses can then be leveraged to perform quantitative and qualitative empirical investigations into the categories of API calls and usage patterns that exhibit energy consumptions profiles. The most used intermediate representation for this category of papers is JAVA_CLASS with 6 out of 9 papers, followed by JIMPLE with 2.

**Clone Detection.** It is also well known that there are some circumstances that lead app users to use APK repositories different from Google Play, which generates a concern about the origin and provenance of Android apps in general. Therefore, approaches such as DNADroid[**crussell:ESORICS2012**] uses neural networks and dinamic-, static analysis to propose detection of ransomware before infection happens. At the same time, Crusell *et. al.*, [**crussell:ESORICS2013**] propose a scalable to detecting similar Android Apps based on their semantic information. The most used intermediate representation for this category of papers is SMALI with 3 out of 7 papers, followed by DEX_ASSEMBLER with 2.

**Test Case Generation.** A common way to perform analysis of an application is using systematic exploration, nevertheless running real world applications in real devices is cumbersome due to several problems like non-determinism, non-standard control flow, etc. Because of this, A3E[**azim:OOPSLA2013**] uses static, taint-style, dataflow analysis on the app bytecode to construct a higher level flow-graph that captures legal transition among activities, and can be used to explore the app in a user-like behavior. At the same time, Jensen *et al.*[**jensen:ISSTA2013**] propose a two-phase technique that uses concolic execution to build summaries of the event handlers of the application and builds event sequences backward from the target, enabling the testing of parts that require more complex event sequences. The most used intermediate representation for this category of papers is JIMPLE with 3 out of 6 papers, followed by WALA_IR and SMALI with 1 paper each one.

**Code Verification.** Code verification intends to ensure the correctness of a given app but without testing (*i.e.,* app execution). For instance, Cassandra [**lortz:SPSMD2014**] is proposed to check whether Android apps comply with their personal privacy requirements before installing an app. As another example, researchers have also extended the Julia [**payet:IST2012**] static analyzer to perform code verification through formal analyses of Android programs. The most used intermediate representation for this category is JAVA_CLASS with 2 papers.

**Cryptography Implementation Issues.** In addition to the aforementioned concerns, state-of-the-art works have also targeted cryptography implementation issues. As an example, CMA [**shuai2014modelling**] performs static analysis on Android apps and select the branches that invoke the cryptographic API. Then it runs the app following the target branch and records the cryptographic API calls. At last, the CMA identifies the cryptographic API misuse vulnerabilities from the records based on the pre-defined model. It is worth noticing that the top representations used by the reported papers are JIMPLE (38 papers), SMALI (26 papers) and JAVA_CLASS (22 papers), and the main concern covered is security with around 101 papers.

Therefore, as we wanted to use an intermediate representation that is closer to the compiled code, we have to choose between JIMPLE and SMALI. Because of this, we extended our research to find existing studies aimed at comparing these two intermediate representations. After a short review in google scholar, we found a paper by Arnatovich *et. al.*,[**arnatovich2014empirical**] called *Empirical Comparison of Intermediate Representations for Android Applications* in which they study the preserveness of program behavior, by *disassembling, assembling, signing, aligning and installing* 520 applications selected from the Google Play Store. The way they studied this was by running a random GUI-based input generation program (*i.e., monkey runner*) over each app before and after the designed process[1] and collecting the amount of apps that crashed. Using this result, they were able to identify the amount of apps that do not crash after this process. The results of the study are summarized in Table 3.2:

TABLE 3.2: Comparision of program behavior preserveness for SMALI, JIMPLE and JASMIN

| Intermediate Representation | Preserved Program Behaviors of Original Applications (%) |
|---|---|
| SMALI | 97.68 |
| JIMPLE | 85.58 |
| JASMIN | 81.92 |

[1]The monkey runner generates a seed that when given as parameter replicates the same events.

Therefore, knowing that SMALI is the intermediate representation that preserves more the program behavior, we decided to use it along with the tool studied to generate mutants that only get affected by the mutation process.

## 3.2 Mutation Testing for Android Apps

There are some previous work devoted to Mutation testing for Android apps. First, Linares-Vásquez *et. al.*,[**linares2017enabling**, **Moran:ICSE18**] (to the best of our knowledge) have implemented the most comprehensive tool for mutation testing at source code level, MDroid+. They empirically extracted a taxonomy of crashes/bugs in android apps, and based on that, then proposed a set of 38 mutation operators. At the same time, Deng *et al.*, [**deng2015towards**], presented a set of eight mutant operators oriented to mutate core components of android (*e.g., intents, event handlers, XML files and activity lifecycle*). Deng *et al.* ,presented in 2017 the implementation of their 8 mutation operators in their paper "Mutation Operators for Testing Android Apps"[**deng2017mutation**]. Finally, the last work found was muDroid[**mudroid**] a mutation testing tool that works at APK level, but it implements standard mutation operators. However, MDroid+ authors [**linares2017enabling**, **Moran:ICSE18**] found that muDroid generates around 53% of non-compilable mutants, that can be translated into a lost of half of the time invested on executing muDroid.

# Chapter 4

# Solution Design

## 4.1 General Approach

With the purpose of enabling the execution of software engineering tasks at APK level, we propose a three phase process depicted in Figure **??**. The first phase consists of APK processing in order to generate models of the application, and the second phase consists of using those models by a certain module that represent a software engineering task desired by users. The final phase consists in rebuilding the APK when required, as in the case of mutation testing or app instrumentation for dynamic analysis. Note that enabling automated software engineering tasks at APK level is our long term goal, therefore, rebuilding the APK (but whitout decompiling the app to get the source code) is a required step in our approach.

**APK processing.** The first step during the model generation is decoding an APK file to be able to extract the Android opcodes and resources. Note that an APK is a zip file that contains resources files and a classes file that include the opcodes in DEX format for all the classes belonging to an Android app (including libraries). Based on the wide usage of SMALI as intermediate representation (top 2 according to Li *et. al.,* [**li:IST2017**]) and that it keeps about 97% of the information in an APK [**arnatovich2014empirical**, **arnatovich2018comparison**], we propose it as the representation used for the models extraction. There are already parsers and lexers for SMALI which allows the extraction of abstract syntax trees (ASTs). Concerning the textual information available in resources files, it can be easily extracted because of the XML nature of those files in Android. Note that extracting SMALI code does not require de-compilation to original source code (*e.g.,* Java) which is time consuming and prone to de-compilation errors.

**Software Engineering Tasks modules.** Because SMALI can be used to extract representations and models such as ASTs, control flow graphs, among others, a plethora of analysis can be instantiated and without the need of original source code. Given the models of the application, implementing a software engineering task must be done on a separate module. These modules must be able to consume the models and provide a comprehensive result to the user. For example, a mutation testing module at APK level must use the AST models extracted from SMALI code to (i) identify the possible mutable snippets of code, and (ii) mutate the original app.

**Re-building/packaging the app**. The last phase of this process, consist of going back from decoded SMALI representation to an executable APK file, which does not require recompiling the modified code. However, note that this requires to be very careful when modifying the SMALI code to avoid injecting bugs into the app when the SMALI syntax is not properly used.This building process must be called by each software engineering task module, making sure all modules that require the execution of this process use the same format to deliver the internal result.

## 4.2 MutAPK

In order to validate the feasibility of our proposed approach we decided to implement it using mutation testing as a reference. As it was shown in the Section 3, mutation testing is still one of the software engineering tasks that has not been developed at APK level. Consequently, we implemented the proposed approach in a tool ( ***MutAPK*** ) that allowed us to compare the obtained results when working in the scenarios of having source code, and having only APK files.

As it was mention before, MutAPK must comply to some must-have rules for all mutation testing tools, it has a set of mutant operators, it provides the possibility to select the mutant operators that will be executed, it defines in detail the short process required for its extension and enables parallel execution. MutAPK is an Open Source project available at `https://github.com/TheSoftwareDesignLab/MutAPK`. In the following sections, we describe MutAPK according to its workflow described in Figure **??**

### 4.2.1 APK Processing

#### Unpackaging/Packaging APK

Recalling the study by Arnatovich *et al.* [**arnatovich2018comparison**], we use APKTool, which allows us to process an APK returning a folder with all the resource files decoded and the source files disassembled into SMALI files. Additionally, code-related files are presented in a useful project like file structure. Because of this, some source code analysis tasks that are based in file location can be easily translated to be executed over APKTool decoding result. At the same time, APKTool allows to build an unsigned apk from the previously mention files. Therefore, an application can be modified in its SMALI representation and then packaged again into an APK for its use.

#### Derivation of the Potential Fault Profile

We followed the same approach proposed by Linares-Vásquez *et al.* [**linares2017enabling, Moran:ICSE18**] Therefore, we detect mutation locations by extracting a Potential Fault Profile, and then we implement mutation operations on those locations. The ***Potential Fault Profile PFP*** is a set of code locations that represent potential points were a fault can be injected. These potential fault injection points are defined through the mutation operators shown in the Section 4.2.2. For the implementation of MutAPK, the PFP definition was inherited from MDroid+. First, both XML and SMALI files are statically analyzed searching for instructions that comply with the characteristics defined in the mutation operators. This previous process, also inherited from MDroid+, consist for XML files of going through the content looking for matches between the file tags and the different mutation operators potential fault injection points.

On the other hand, for SMALI files the process is based on the Abstract Syntax Tree that is obtained using the lexer and parser created by APKTool to perform the disassembling of an APK. In particular, MutAPK uses the visitor design pattern to identify the possible locations. Knowing this, the process can easily be extended to add new operators and to provide more comprehensive analysis of the app (*i.e.,* resource and SMALI files) if needed. The final result of the PFP derivation process is a list that joins the potential fault injection points with the mutation operators that can be applied to those locations.

### 4.2.2 Mutation Testing Module

#### Operators

We built upon the 38 operators proposed by Linares-Vásquez *et al.* [**linares2017enabling, Moran:ICSE18**] , which are representative of potential fault in Android apps and can be found either on source code statements, XML tags, or locations in other resource files. In MutAPK we implemented (i) the 33 operators implemented in

MDroid+ that do not lead to compilation errors, and (ii) two additional operators not available in MDroid+.

Therefore, our work was to translate the operators implementation from the original source code-based implementations to the corresponding implementations in the SMALI representation. In order to do this, we manually selected 11 apps that had potential locations for implementing the operators. Given those applications, we built the APKs using Android Studio and disassembled manually each one to recognize the direct translation of each mutation original statement. After this dictionary is created, we proceeded to mutate manually the source code of these 11 applications and proceed to generate again the APK files. Finally we performed a diff comparison between the SMALI representation of the original version against the SMALI representation of the mutated version. Because of this, we were able to translate successfully each mutation operator from source code to SMALI representation. With this procedure we derived the list of operators implementation at APK level; the details of each operator are available in our online appendix [**MutAPK**].

### Mutant Creation

We start by unpackaging the apk into a temporal folder. After that, the PFP is derived and a list of potential fault injection locations is created. We now can use the defined mutation rules to generate the mutants, however, in order to make this process as efficiently as possible, MutAPK provides the option to parallelize the mutant creation. For each location in the PFP, a copy of the disassembled apk is created. After the copying ends, based on the mutation operator associated, a new process that translate the mutation rule into an actual change is executed over the exact location inside the associated folder. Next, a compilation process is triggered in order to generate as result an APK. As it was said before, this process can be parallelized and each task consist of: copying the dissambled apk, mutating either a resource file or a SMALI file, and finally compiling the result to obtain an APK. Nota that while MDroid+ only generates the source code of the mutants, MutAPK is able to generate APKs ready to install and test.

### Extensibility

Due to the fast change of the android framework, MutAPK must provide the possibility to add new mutation operators easily. Therefore, in order to enable a new mutation operator some changes must be implemented: (i) create a new detector/locator that is capable of finding the correct position that provides all the information needed to create a *Mutation Location* defined in MutAPK; (ii) a mutator, that is capable of using the previously identified location information to mutate the code or resource file; (iii) update the *operator-types.properties* file found under the "src/uniandes/tsdl/mutapk" folder to add the new mutation operator file path with its defined id; (iv) modify *OperatorBundle.java* (in case the new operator is text-based) to add the new text detector and (v) update the *operators.properties* file.

At the same time, MutAPK counts with an *extra* folder where the external libraries are located. Therefore, if user wants to improve the file analysis process or wants to execute a more specialized process over the application, he can save the library files in this folder and manage them easily. MutAPK has in this *extra* folder the *jar* file provided by APKTool. Note that here there is a big difference with MDroid+ implementation; because in MDroid+ the source code must be compiled, then it requires in the extra folder all the libraries required to compile the source, code. This is not required in MutAPK because we are already working with "compiled" code.

### 4.2.3 Tool Usage

MutAPK has been designed to work as a command line tool. In order to use it, the user must have installed Maven and Java. The MutAPK repository[**MutAPK**] must be cloned and then packaged using the following commands

LISTING 4.1: Git and Maven commands to build MutAPK

```
git clone https://github.com/TheSoftwareDesignLab/MutAPK.git
mvn clean
mvn package
```

After that, the jar file called MutAPK-<version>.jar will be located in the *target* folder. That file can be relocated and used in other places.

LISTING 4.2: Console command to run MutAPK

```
java -jar MutAPK-<version>.jar <APKPath> <AppPackage> <OutputFolder>
    <ExtraComponentFolder> <operatorsDir> <multithread>
```

To run it, the previous command must be used (Listing 4.2), where

1. **<APKPath>** is the path to the app's APK

2. **<APKPackage>** is the app package used to identify the code that belongs to the app (not the libraries)
3. **<OutputFolder>** is the path to the folder where all the mutants will be generated.
4. **<ExtraComponentFolder>** is the path to the folder that has the extra libraries used by MutAPK
5. **<operatorsDir>** is the path to the *operators.properties* folder, that describes the operators that must be used to generate mutants
6. **<multithread>** boolean value, defines if MutAPK must be executed using multiple threads

LISTING 4.3: Example Output of MutAPK for PhotoStream app

```
Amount Mutants   Mutation Operator
1                OOM_LARGE_IMAGE
3                NULL_INTENT
5                NULL_OUTPUT_STREAM
1                INVALID_FILE_PATH
5                INVALID_LABEL
19               NULL_VALUE_INTENT_PUT_EXTRA
7                INVALID_COLOR
9                FINDVIEWBYID_RETURNS_NULL
19               INVALID_KEY_INTENT_PUT_EXTRA
5                LENGTHY_GUI_CREATION
8                VIEW_COMPONENT_NOT_VISIBLE
3                NULL_INPUT_STREAM
0                SDK_VERSION
7                INVALID_ACTIVITY_PATH
8                INVALID_VIEW_FOCUS
2                CLOSING_NULL_CURSOR
39               WRONG_STRING_RESOURCE
3                WRONG_MAIN_ACTIVITY
1072             NULL_METHOD_CALL_ARGUMENT
4                NULL_BACKEND_SERVICE_RETURN
2                LENGTHY_GUI_LISTENER
9                INVALID_ID_FINDVIEW
7                ACTIVITY_NOT_DEFINED
5                MISSING_PERMISSION_MANIFEST
2                LENGTHY_BACKEND_SERVICE
3                DIFFERENT_ACTIVITY_INTENT_DEFINITION
Total Locations: 1248
```

When the command is executed in the console , the selected operators and the amount of mutants that are going to be generated for each operator (Listing 4.3) are logged. Additionally, when all mutants are generated a log of the mutation process can be found at the Output Folder defined in the command. This log allows testers to identify what was the mutation applied on each mutant.

As an extension, for testing purposes MutAPK creates 2 csv files: (i) mutants that were successfully compiled, and (ii) summary of the time consumed to mutate and to compile each mutant. It is worth noting that even if the mutant do not compile correctly the second file register the time it took the compilation to fail.

# Chapter 5

# Empirical Study

## 5.1 Study Design

In order to measure the method coverage reached by an exploration tool in one application, there is the need to know how many methods there are in the application and how many methods were called during the exploration. To achieve that, the application developers could count the number of methods in their project and write log lines at the beginning of every method. After that, they will need to compile, run the exploration tool against their application, measure the number of methods that were called during the exploration and compute statistics.

With that in mind, this study consists of nearly the same stages, i. instrumentation, ii. exploration, iii. coverage analysis, iv. summarize, and v. compute statistics. The stages ii. and iii. were repeated 10 times for every application that was selected, that leaded to the iv. stage.

The different stages were completed as follow:

Stage i: The applications' instrumentation was made by using InstruAPK. It is an instrumentation tool developed mainly for this study. This tool uses APKTool, a known Java application that allows inverse engineering in Android apps, allowing applications' instrumentation without the need of recompiling their source code. APKTool decodes the apk and the result is the smali representation of the app source code, These smali files are analysed in order to find all the methods to be instrumented and then, the log code is injected at the very beginning of each method. Its important to notice that no external libraries methods are instrumented. InstruAPK only search for methods following the android project structure that uses the application package name to store the application source code.

Stage ii: The exploration was made by four different automatic exploration tools. two from the industry and two from the academic side. The first tool was Firebase Test Lab. it was selected for being widely used in industry and for also being a Google product. The second one, Monkey, was selected for being the most basic one and because it is also included in the SDK for developing Android Apps. The third one, Droidbot, was selected from the academic side. Droidbot has been a point of study for many researches. Many others tools have based their functionality on this tool. The last one is RIP, this tool was selected for being of special interest for us. It is our own exploration tool and is is currently an active project inside the Software Design Lab at University of Los Andes.

Every tool was executed ten times per application, and every execution with a maximum time of 30 minutes. Some tools ended its exploration before the max time. The number of executions and the maximum time were arbitrary decisions that were made because of time limitations for the study. Although, during the study was notice that most of the tools ended the exploration or reached their maximum coverage within the first 15 minutes. Which means that the maximum time for exploration was more than enough in almost all cases.

stage iii: The coverage measurement was made by CoverageAnalyzer. It is a Java Application created mainly for this study. This tool analyses the resulting logcat of an Android phone, when executing an application that was instrumented by InstruAPK. It extracts different data such as, number of methods called, number of methods never called, number of error traces of the application being analysed, most called methods, less called methods, as well as the time stamp of all calls of every method. It is important to notice that the possibility of extracting all those information is because InstruAPK provides it.

stage iv: Due to the multiple executions of every APK per tool, it is important to summarize de data. The final report contains the total number of unique methods called during all ten executions, reporting only the first time they were called. The exploration reports are analysed and filtered as well.

stage v: This is the final stage of the study that involves, understanding data, computing statistics, creating graphs, extract insights and conclusions.

Besides that, for this study, a set of 11 applications was used. This set is a subset of a set of open source applications utilised inside The Software Design Lab research group for other studies and tests, including RIP. Every APK in the subset should be successfully instrumented by InstruAPK, it should compile without any problem after instrumentation and it should be launch in an emulator without any

TABLE 5.1: Applications used for the study

| App ID | Package Name | # Methods Reported by APKAnalyzer | # Methods Instrumented by InstruAPK |
|---|---|---|---|
| 1 | appinventor.ai_nels0n0s0ri0.MiRutina | 61993 | 9351 |
| 2 | com.evancharlton.mileage | 4000 | 1162 |
| 3 | com.fsck.k9 | 18799 | 7003 |
| 4 | com.ichi2.anki | 32370 | 2209 |
| 5 | com.workingagenda.devinettes | 19274 | 66 |
| 6 | de.vanitasvitae.enigmandroid | 13083 | 574 |
| 7 | info.guardianproject.ripple | 19429 | 100 |
| 8 | org.connectbot | 20606 | 1145 |
| 9 | org.gnucash.android | 75473 | 504 |
| 10 | org.libreoffice.impressremote | 14691 | 649 |
| 11 | org.lumicall.android | 45784 | 540 |

issue after instrumentation.

## 5.2 Context of the Study

In order to present a fair comparison between MutAPK and MDroid+, we have used the same apps MDroid+ used for their experiments. This 54 applications presented in Table 5.1 belong to 16 different categories of the Google Play Store. It is worth noticing that these 54 applications are open source and allows us to study the way code statements are translated from JAVA to SMALI.

In order to collect data that allow us to answer the research question, we compared MutAPK to an existing tool for mutation testing that works at source code level ( MDroid+ [**linares2017enabling**] ). The experiments were executed on a class-server machine. Note that in MutAPK, we implemented only 35 of 38 operators listed in Table 5.2 because the other 3 operators lead to non-compilable results. In order to analyze the impact of mutant generation process in MutAPK, we collect: (i) number of mutants generated per mutation operator per application; (ii) number of mutants that compile after mutation; (iii) mutant generation time (*i.e.,* the time required to generate each mutant) and (iv) mutant building times (*i.e.,* the time required to compile each APK file)

## 5.3 Results: Impact of generating mutants at APK level

$RQ_{1.1}$: To study our results, we present them in two stages, first we show a comparison where only the 33 mutants in both MDroid+ and MutAPK are taken into account. In Figure **??** we show the total amount of generated mutants per app. MutAPK generates around 30 more mutants per app (17% more than MDroid+). However, if all operators are taken into account, the difference between the amount of mutants get bigger. Figure **??** shows the amount of generated mutants per app. As it can be seen, MutAPK outperforms MDroid+ generating in average 1211 more mutants per app, this corresponds to 7.3 times more mutants. For further analysis of the results at app level, we added the Tables **??** and **??**, where all info collected is summarized around apps (See Appendix A). Also, we show in Figure **??** that the amount of mutants generated per mutant operator are very similar between MutAPK and MDroid+. It is worth nothing that this figure does not take into account the 63441 mutants generated by one of the operators implemented only in MutAPK.

$RQ_{1.2}$: If we consider again only the 33 shared mutants, in Figure **??** we can see that MutAPK generates around 16% of non-compilable mutants while MDroid+ generates only 0.5%. Nevertheless, when using all operators MutAPK generates around 2.36% of non-compilable mutants while MDroid+ lightly increase its rate to 0.6%. At the same time, Figure **??** shows the percentage of non-compilable mutants in terms of the mutant operators, from this we can see that there is also a similar behavior for both. Specifically, MutAPK generates in average 0.1% non-compilable mutants while MDroid generates 0.05%.

$RQ_{1.3}$: The most important result is the execution time. MutAPK takes only 3% of the time ( 144,66ms ) required by MDroid+ ( 4,6 seconds) to mutate a copy of the app. Therefore, due to the infraestructure used to run our study, MutAPK takes 9 seconds to generate all mutants for an app (on average), while MDroid takes 19 seconds.

$RQ_{1.4}$: For compilation, MutAPK spends only 6.3% of the time required by MDroid+ to compile a mutant. Consequently, MutAPK takes 11 min to compile all mutants for an app (on average) while MDroid+ takes 13 min.

Finally, if all mutant operators are selected, MutAPK takes around 9.63 hours to complete the mutation and compilation process for the 54 apps while MDroid+ takes 12 hours. It is worth remembering that MutAPK generates around 7.3 times more mutants than MDroid+. Therefore, the remaining time could be used by developers, practitioners, and servers to other software engineering activities. Additionally, as MutAPK generates more mutants, the generated search/bugs space might be more comprehensive, which means that the quality of the test suite can be tested in a more wide sense.

## 5.4 Analysis of non-compilable mutants

In order to understand the reasons for non-compilable mutants, we analyzed 3 mutants for each one of the mutant operators that generated non-compilable. It is worth noting that this process must be iterative and after finding and fixing the errors, the mutation process must be executed again.

### 5.4.1 31 - InvalidIDFindView

This operator generated more non-compilable mutants than others. For this operator we found there is an implementation error when the mutation was performed. The correct implementation should be to include *const <constVarName>, 0x<randomlyGeneratedHexa>* before the view was created to assign a random generated value to the key used as view ID. However, we injected *const/16 <constVarName>, 0x<randomlyGeneratedHexa>* that generated a packaging error due to specific instructions that must accompanying *const/16* and not *const*.

After this error was fixed the percentage of non-compilable mutants at app level without taking into account non-shared operators decreases to 4%.

### 5.4.2 27 - FindViewByIdReturnsNull

This operator presents two cases we did not consider. Listing 5.1 presents the SMALI representation for finding an Android view; the mutation rule asks to convert the result of the search into a null object. Therefore, Listing 5.2 presents the SMALI instruction that must be injected instead of the previous one to assign a null value to the result. Nevertheless, after the mutation is performed when the compilation process is launched, an error is displayed on the console (Listing 5.3), saying that all available registers are between 0 and 15. After a deeper analysis, we found that registers after 16 inclusive are used only for referencing values and a null value could not be assigned. Therefore, we found that a cumbersome process most be made and a verification of the value of the 16th available register must be performed to save the value while the result of the mutation is used, and then the original value can be reassigned to the used register.

This behavior was found in several mutants.

LISTING 5.1: SMALI representation of findByViewID method call

```
invoke-virtual {v0, v2},
    La2dp/Vol/main;->findViewById(I)Landroid/view/View;

move-result-object v21

check-cast v21, Landroid/widget/Button;
```

LISTING 5.2: SMALI representation of a null value being assigned

```
const/4 v21, 0x0
```

LISTING 5.3: APKTool console response

```
I: Using Apktool 2.3.2
I: Checking whether sources has changed...
I: Smaling smali folder into classes.dex...
test\smali\a2dp\Vol\main.smali[4027,4] Invalid register: v21. Must be
    between v0 and v15, inclusive.
Could not smali file: a2dp/Vol/main.smali
```

We found that last line of Listing 5.1 that is in charge of checking the type of the result, is not necessary and can be removed in some cases as it can be seen in Listing 5.4 . Therefore, our implementation search for that instruction to recognize the complete set of instructions that will be replaced. Therefore, MutAPK throws an error when trying to match this expression with next line.

LISTING 5.4: APKTool console response

```
invoke-virtual {v7, v9},
    Lcom/angrydoughnuts/android/alarmclock/ActivityAlarmNotification;->
          findViewById(I)Landroid/view/View;

move-result-object v7

invoke-virtual {v7, v12}, Landroid/view/View;->setVisibility(I)V
```

### 5.4.3 4 - InvalidKeyIntentPutExtra

Listing 5.5 shows the result of executing the compilation process over half of the mutants from this mutation operator that are non-compilable. As it can be seen in the listing, the process ends succesfully but no apk file is generated. At this point we think that we might be facing an error within APKTool (*i.e., the tool used for assembling/disassembling an APK*).

LISTING 5.5: Example Output of MutAPK for PhotoStream app

```
I: Using Apktool 2.3.2
I: Checking whether sources has changed...
I: Smaling smali folder into classes.dex...
I: Checking whether resources has changed...
I: Building resources...
S: WARNING: Could not write to
    (C:\Users\Camilo\AppData\Local\apktool\framework), using
    C:\Users\Camilo\AppData\Local\Temp\ instead...
S: Please be aware this is a volatile directory and frameworks could
    go missing, please utilize --frame-path if the default storage
    directory is unavailable
I: Building apk file...
I: Copying unknown files/dir...
I: Built apk...
```

If these 4 mutant operators are updated and they do not generate non-compilable mutants, the percentage of non-compilable mutants at APK level (without taking into account the non-shared operators) should be dropped to 0.1%.

TABLE 5.2: Comparision at Mutation Operator Level

| | | | Amount Mutants Generated | | Amount Mutants Compiled | |
|---|---|---|---|---|---|---|
| ID | Mutation Type | Operator Name | MutAPK | MDroid+ | MutAPK | MDroid+ |
| 1 | Text | ActivityNotDefined | 385 | 384 | 385 | 383 |
| 2 | AST | DifferentActivityIntentDefinition | 482 | 358 | 120 | 356 |
| 3 | Text | InvalidActivityName | 383 | 382 | 383 | 382 |
| 4 | AST | InvalidKeyIntentPutExtra | 477 | 459 | 62 | 456 |
| 5 | Text | InvalidLabel | 214 | 214 | 214 | 214 |
| 6 | AST | NullIntent | 482 | 559 | 413 | 556 |
| 7 | AST | NullValueIntentPutExtra | 477 | 459 | 452 | 459 |
| 8 | Text | WrongMainActivity | 56 | 56 | 56 | 56 |
| 9 | Text | MissingPermissionManifest | 227 | 229 | 227 | 229 |
| 10 | Text | WrongStringResource | 3432 | 3394 | 3430 | 3394 |
| 11 | AST | NotParcelable | 0 | 7 | 0 | 1 |
| 12 | Text | SDKVersion | 0 | 66 | 0 | 66 |
| 13 | AST | LengthyBackEndService | 15 | 8 | 15 | 8 |
| 14 | AST | LongConnectionTimeOut | 0 | 0 | 0 | 0 |
| 15 | AST | BluetoothAdapterAlwaysEnabled | 0 | 4 | 0 | 4 |
| 16 | AST | NullBluetoothAdapter | 9 | 9 | 9 | 9 |
| 17 | AST | InvalidURI | 2 | 2 | 1 | 2 |
| 18 | AST | NullGPSLocation | 2 | 1 | 2 | 1 |
| 19 | AST | InvalidDate | 20 | 40 | 20 | 40 |
| 20 | AST | NullBackEndServiceReturn | 34 | 8 | 25 | 7 |
| 21 | AST | InvalidMethodCallArgument | 0 | 0 | 0 | 0 |
| 22 | AST | NullMethodCallArgument | 63441 | 0 | 63437 | 0 |
| 23 | AST | ClosingNullCursor | 222 | 179 | 221 | 166 |
| 24 | AST | InvalidIndexQueryParameter | 82 | 7 | 82 | 6 |
| 25 | AST | InvalidSQLQuery | 82 | 33 | 82 | 33 |
| 26 | AST | ViewComponentNotVisible | 398 | 347 | 396 | 342 |
| 27 | AST | FindViewByIdReturnsNull | 915 | 413 | 803 | 413 |
| 28 | Text | InvalidColor | 47 | 52 | 44 | 52 |
| 29 | AST | InvalidViewFocus | 397 | 0 | 393 | 0 |
| 30 | AST | BuggyGUIListener | 0 | 122 | 0 | 122 |
| 31 | AST | InvalidIDFindView | 915 | 456 | 0 | 452 |
| 32 | AST | InvalidFilePath | 228 | 220 | 226 | 220 |
| 33 | AST | NullInputStream | 90 | 61 | 88 | 61 |
| 34 | AST | NotSerializable | 0 | 15 | 0 | 8 |
| 35 | AST | OOMLargeImage | 7 | 7 | 7 | 3 |
| 36 | AST | LengthyGUIListener | 339 | 122 | 335 | 122 |
| 37 | AST | NullOutputStream | 59 | 45 | 59 | 45 |
| 38 | AST | LengthyGUICreation | 336 | 129 | 330 | 129 |
| | | Total | 74255 | 8847 | 72317 | 8797 |

# Chapter 6

# Conclusion

We presented in this thesis a novel framework to enable automation of software engineering tasks at APK level through a proposed architecture presented in Section 4.1. Additionally, we validate its feasibility by implementing a Mutation Testing tool called MutAPK [**MutAPK**]. We evaluate the performance of MutAPK by comparing it with MDroid+, a Mutation Testing tool that works over source code. Our results show that MutAPK outperforms MDroid+ in terms of execution time, generating a testeable APK in a 6.28% of the time took by MDroid+. In terms of mutant generation MutAPK has a similar behavior to MDroid+ for the shared mutation operators generating about 17% more mutants (*i.e.,* around 30 more mutants per app). Nevertheless, MutAPK has implemented 2 operators not implemented yet by MDroid+, which enable the generation of about 85% of the mutants created. Therefore, MutAPK using this operators increases the difference to 739% more mutants (*i.e.,* around 1211 extra mutants per app).

Nevertheless, the mutation process done by MutAPK needs an improvement due to high rate of non-compilable mutants generated. In average, when using only the shared operators, 16% of the generated mutants by MutAPK are non-compilable and when all operators are used there are 2.36%. In this metric, MDroid outperforms MutAPK with only around 0.6% non-compilable mutants for both cases. Therefore, there is room for improvement because MutAPK should generate only compilable mutants, because it works on already compiled code from source code.

Finally, our results of the initial study with mutation testing suggest that in fact software engineering tasks can be enabled at APK level, and in the particular case of mutation testing we showed that working at APK level improves mutation testing times.

# Chapter 7

# Future Work

In this chapter we propose improvements and specialized tasks that could be done after this first stage of the research. First, a more comprehensive search of related work must be done to identify software engineering tasks that has been addressed using static analysis of android apps since 2016. Additionally, this further research can provide more information about the next to be implemented software engineering task (at APK level) in our pipeline, which could be either test cases generation, on-demand documentation, or another one.

At the same time, some effort must be dedicated to fully study the bug taxonomy generated by MDroid+ authors, in order to define more mutation operators or to propose other approaches to identify new possible bugs that could be translated into new mutation operators. Even more important, effort should be devoter to fix the high rate of non-compilable mutants that is generated by MutAPK.

Also, it will be helpful to build a wrapper for MutAPK ( or a new tool ) that is capable of orchestrating the execution of a test suite over the generated mutants. It is important for that solution to offer the possibility of deploying multiple AVD or similar representations and manage them taking into account different challenges as fragmentation, test flakiness, cold starts, etc. [**8094439**]

As an extension of the research question addressed in this thesis, an extensive study must be done using top applications of the different categories from the Google Play Store, to validate the behavior of MutAPK for more complex applications. Finally in terms of the implementation of MutAPK, some research effort can be invested in designing a model that improves the location recognition and provides enough information to continue mutating the SMALI representation in the registered times.