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

Systems and Computing Engineering Department

July 8, 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

First, I want to express my deepest thanks to Professor Mario Linares-Vásquez for helping me with the development of this final work, giving me the necessary feedback for getting this project to this final version. Giving me new ideas and ways to solve the presented problems while executing this research.

Second, I would like to give my thanks to all the members of The Software Design Lab, for sharing with me their experiences and knowledge which were very important for developing this thesis. Especially to Camilo Escobar for his great help giving the main concept of InstruAPK, for helping with its implementation, besides giving me feedback about the figures in this text as well as solving some extra questions and doubts that I had during the process of developing this thesis.

Third, I want to say thanks to my mother and sister for keeping me motivated within all my major, till the last moment of it. For their unconditional support and for being there in the moments I needed them the most.

At last, but not least important, I want to say thanks to all my friends for sharing their knowledge with me and contributing with that to the final product of this thesis.

Without the help of the people mentioned, this work would not be possible.

I wish to clarify that the order in the mention does not reflect the level of thankfully I feel for the people mentioned in this statement. All of them supported this work in different ways, and under their capabilities, and helping me in one way or another to reach this results. For that reason, all of them deserves the same feelings from my. One more time, thanks to all of them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Mobile applications market is a continuous growing market. According to Statista, the number of available application, by the 2020, in the two main markets together is 4407000. The great amount of applications means that there is a vast number of users willing to download them, but also means that they can change an application, or even the platform, whenever they desire to, or when something does not fulfil their needs or expectations. Therefore, every day more and more complex applications are launch in the market as well as the users becomes more demanding. As a result, every new application or new functionality should have a good quality, they should work as expected in all different scenarios the distinct users will put them in, they have to be developed very fast and also, they have to be cheap in order to be sustainable for the company who develops them.

The great amount, the complexity, and also the more demanding users make this commerce a very competitive one.

In order to reach users quality expectations, the frequently releases time of the market and the sustainability needed for the companies, many things have been developed, among them, automated testing. Automated testing has been of high interest for researchers and companies because it lowers the costs of production and it allows better quality products. One of the branches of automated testing is the automatic exploration tools, these tools aim to explore the application as deep as possible and find as many errors as possible. There is a plethora of exploration tools. Every year there are more of them, each one using different exploring strategies. Some of them use random inputs, others use image analysis, others a mixture of these two, and new researches are starting using AI. It is easy to think that the more deep exploration the more errors will be found, but, as will be shown in this text, that is not always the case. The number of errors detected can vary due to the exploration strategy because of the nature of the

errors. Furthermore, most of this exploration tools are developed for Android applications.

Examples of the automatic exploration tools are Monkey, this tool uses a semi-random generation events strategy to explore the applications, leading to different exploration results unless the same seed is given for the generation of the random events. This tool is the default one provided by Google; Firebase Test Lab, this tools is the only one which is online. Accordingly to its documentation, it analyzes the UI of the applications and explores it simulating users events. They claim to always explore the tool in the same order. Another exploration tool is RIP, an active project inside the University of Los Andes at the research group The Software Design Lab, it explores applications using a model-based GUI testing with multiple comparison criteria. It can lead to different exploration results due to its current implementation as well as to the comparison criteria.

For developers is important to know the difference between these tools. They need to know the tools that will match the best in the new incoming project to make a good decision. Besides, the project bugged, and application complexity among others things can also affect the decision. The information available to make the decision is most of the time based in what the tools claim to do and if the developers have enough working with more than one then they have some extra information. Though, this information is not completely reliable, it is not objective and can be slanted.

1.1 Problem Statement

Accordingly to the previous section, the number of automatic exploration tools is increasing annually. Thus, with no objective information about them, developers will need to explore new tools and compare them to find the best one. It can turns easily in spending valuable time and at the end in making the wrong decision, resulting in poor quality final products. In short, the decision of the right automatic exploration tool should be easy, quickly and rely in objective data, such as coverage reached, and number of errors found.

1.2 Thesis Goals

The main objective of this thesis, is to provide quantitative and qualitative information of the most widely used automatic exploration tools for Android mobile applications, to facilitate developers in the selection of the right tool that suits

their needs. Under those circumstances, the next specific objectives were proposed.

1. Compare the tools by their exploration coverage
2. Compare the tools by the number of unique error traces discovered while exploring an application.
3. Compare the tools using qualitative aspects such as, is the tool a open source project? Is the tool free? Is the tool allowing introduce login values? how useful is the tool report for developer to reproduce, find and fix bugs?

1.3 Thesis contribution

The main contribution of this thesis is to provide developers with enough and objective information, to decide which automatic exploration tool suits their projects' needs the most, making this process easy and less time consuming.

// TODO aquí

1.4 Document Structure

//TODO aquí

Chapter 2

Related work

2.1 Static Analysis of Android Packages

// TODO Agregar la sección por cada related work

Chapter 3

Solution Design

3.1 General Approach

With the aim of complete the objectives mentioned in Sec.1.2, a workflow was designed. This work flow contains five stages.

1. Instrumentation of the applications
2. Exploration
3. Coverage measurement
4. Summarize data
5. Data Analysis

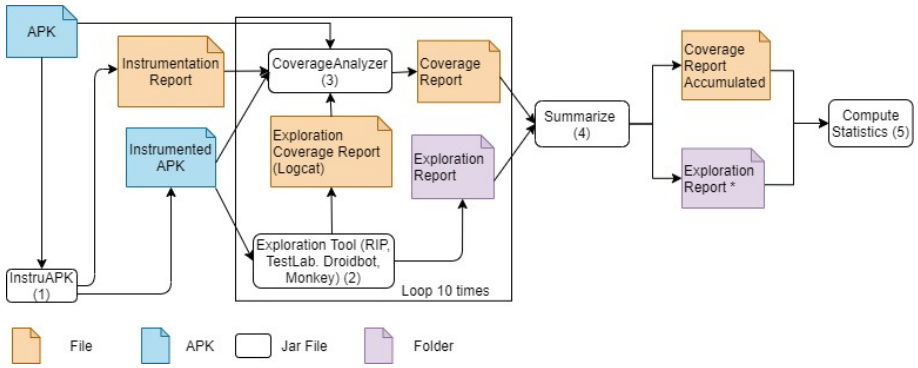

FIGURE 3.1: Main Workflow

For the first stage, InstruAPK was used to make the instrumentation of the applications. This tool only takes into account the methods under the package name of the application that is being analysed. As a result, methods from different libraries are not being taken into count. The input for this stage is the original APK file, and the output is the instrumented APK together with the instrumentation report containing general information such as the file path, method's name, file

name, and the method arguments, as well as a sequential number that will help us to know the total amount of instrumented methods and will work as their unique identifier.

The exploration was made by four different exploration tools, i. Droidbot, ii. Monkey, iii. Firebase Test Lab, and iv. RIP. Every tool was run to explore with a maximum time of 30 minutes. It is important to notice that, even when the max execution time seems to be short, it was enough for most of the application. This was because of the size of the applications, if the application is small, then the coverage will increase rapidly, because a bug was found during the exploration, or even because the tool marks the exploration as done.

This stage input is only the instrumented APK file, and its output is the exploration report that every tool provides. I took the logcat from the exploration report, but when it was not available in it, which was the case of Monkey, it was extracted by using the adb shell command. The other three tools got the information at the end of their exploration.

Some of these tools, Droidbot, Monkey and Test Lab, were selected because of their high use in the industry and the remaining one was selected because of personal interest owing to it is an active project at the University of Los Andes inside the research group The Software Design Lab were I make part of.

In stage 3, **CoverageAnalyzer (CA)** was used to make the coverage measurement and search for error lines. This stage inputs are the original APK, the instrumentation report and the instrumented APK, both from stage 1, and the logcat from stage 2. Its output is a report containing two method coverage measurements, the first one, calculated using the number of methods reported by APKAnalyzer, a tool provided for Google, and the second one with the number of instrumented methods reported by InstruAPK.

Stages 2. and 3. were repeated 10 times for every application that was selected, that leaded to the stage 4. The multiple executions are intending to get average values as well as comparable results along the different exploration tools. The input for this stage are all the method coverage reports from of the stage 3 as well as the exploration report from stage 2. The output are the accumulated method coverage by each tool for each application, which was calculated taking the unique methods called over all the ten executions, the average accumulated method coverage over time, as well as the number of errors and its average found per application.

The final stage encompasses data understanding, graphs creation, comparison using the graph and analysis of different qualitative aspects of every exploration tool. Thus,

Any person can reproduce this work flow who desires to compare different exploration tools, even can make use of the same tools for the instrumentation and the coverage measurement, allowing easy and fast comparisons. Consequently, the decision-making starts to be easier for developers and researchers. Also,gives the possibility to researchers of compare their own exploration tools in a effortless and quick way.

3.2 InstruAPK

This tool was developed mainly for this study. It 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. It is 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.

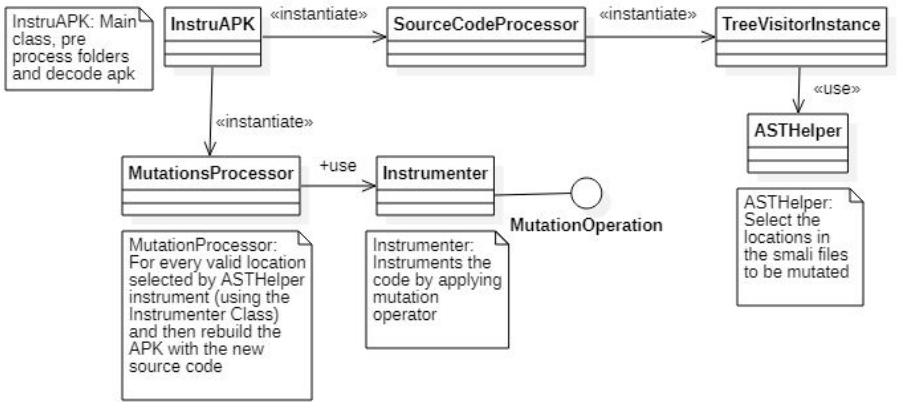

FIGURE 3.2: Class Diagram InstruAPK

3.3 Coverage Analyser (CA)

This tool is also a Java Application created mainly for this study. It extracts all data from the log lines injected by InstruAPK and also the errors found under

the application package name. When an instrumented application is ran, the logcat will contain the log lines with all the data. The logcat is stored in a txt file and that is what CA uses as its input filtering the results using the package name of the application being analysed. For the coverage measurement the tool uses the number of instrumented methods included in the InstruAPK instrumentation report As it was mentioned before, CA depends totally on the information provided by InstruAPK, CA can be seen as a complement of it, rather than an separate application.

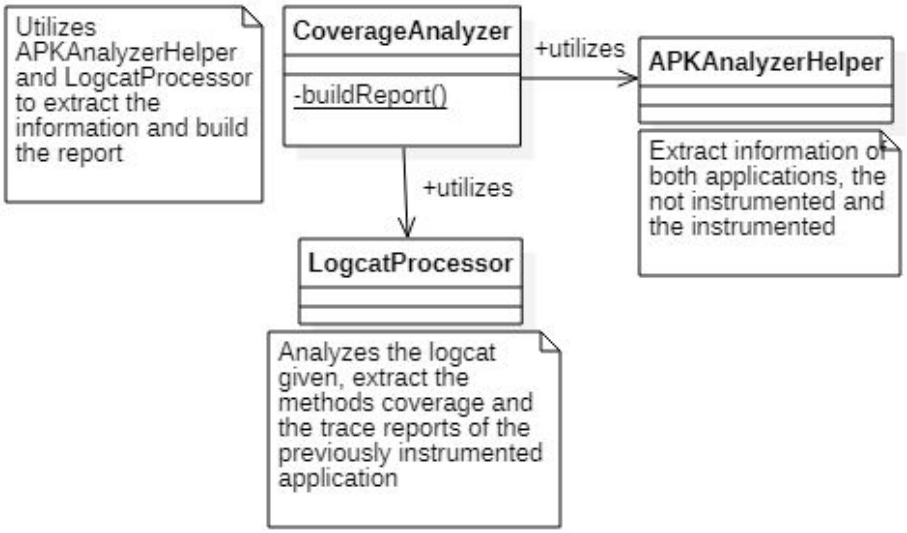

FIGURE 3.3: Class Diagram Coverage Analyser

Chapter 4

Empirical Study

4.1 Study Design

As expected, para cumplir con el objetivo general de esta tesis se deben cumplir con los objetivos especificos, una vez estos sean completados a cabalidad entonces se tiene el objetivos general completado.

Explicar por qué se seleccionaron las herramientas (RIP, TestLab, etc) Explicar las apps utilizadas en el estudio.

La idea es que estos objetivos especificos lleven o ayuden a llegar al cumplimiento de este objetivo general.

Se debe explicar cómo se cumplió con cada uno de ellos y al final explicar cómo se llegó a cumplir con el objetivo general

mostrar los resultados y analizarlos.

//TODO 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

TABLE 4.1: 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

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.

//TODO 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 issue after instrumentation.

4.2 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 4.1 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 **??** 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)

4.3 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.

4.4 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.

4.4.1 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%.

4.4.2 27 - FindViewByIdReturnsNull

This operator presents two cases we did not consider. Listing 4.1 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 4.2 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 4.3), 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 4.1: 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 4.2: SMALI representation of a null value being assigned

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

LISTING 4.3: 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 4.1 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 4.4 . 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 4.4: 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
```

4.4.3 4 - InvalidKeyIntentPutExtra

Listing 4.5 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 4.5: 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%.

Chapter 5

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 3.1. 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 6

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.