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 30, 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

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

// TODO explicar el problema que se quiere solucionar. Para mi es el comparar las diferentes herramientas puede tomar tiempo y elegir la correcta para un proyecto o para una aplicación puede tomar tiempo valioso

As it was said in the introduction, the number of automatic exploration tools is increasing daily. With no objective information about them, is necessary developers to explore new tools and compare them with no basis in order to find the best one. It can turns easily in spending valuable time and at the end 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, 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 main objective mentioned in Sec. **??**, a workflow was designed. This work flow contains five stages, some of these were executed multiple times intending to get average values as well as better comparable results along the different exploration tools.

If a developer or researcher wants to compare a new exploration tool by its coverage he or she should know how many methods there are in the application and how many methods were called during the exploration. To achieve that, the person 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.**

Stage ii: The exploration was made by four different automatic exploration tools. further information about the tools is given in Sec.**??**. //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 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 (CA).**

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.

//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.

explicar el workflow de cómo se obtuvieron los resultados.

APK processing.

3.2 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.

3.3 Coverage Analyser (CA)

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.

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.

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

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

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.