Fault Injection in Android Applications

Adélia Helena Valentim Gonçalves

Mestrado em Engenharia de Software

Supervisor: Ana Cristina Ramada Paiva

October 23, 2019
Fault Injection in Android Applications

Adélia Helena Valentim Gonçalves

Mestrado em Engenharia de Software

October 23, 2019
Abstract

The use of mobile phones is widespread, and more and more time is spent on mobile phones. With the widespread use of mobile phones, the mobile applications available in app stores also increase. There is a growing reliance on mobile applications.

The availability of these applications is made at a fast pace and seeking to make development cost-effective. The consequence of this is often the disregard for the quality of the final product.

Ensuring the quality of the applications allows to ensure the satisfaction of the end customer and their loyalty. It can avoid serious financial and human consequences. To promote the quality of the software, it is necessary to test the software ensuring that it does what is expected, working properly with a high level of quality.

Mutation testing is a technique for injecting faults into code implementation. Each of the faults produced represents a mutant. The execution of these tests makes it possible, in a reliable way, to guarantee the quality and effectiveness of a test suite or testing tool. The goal is to assess the quality of a test suite (or testing tool) by checking if it allows to distinguish the result of the original code compared with the result of each mutant (if it kills the mutants).

The present work aims to define some mutation operators able to reproduce real failures in real Android mobile applications. It extends an existing tool that allows to automate the injection of some mutants.

After generating the mutants it will be possible to assess the quality of a testing tool calculating the mutation score (percentage of mutants killed by the testing tool). As a final goal, ensure, therefore, the increase in the quality of the mobile applications made available.

Keywords: Mutation Testing, Software Testing, Android Testing
Resumo

A utilização do telemóvel encontra-se massificada, sendo que cada vez se passa mais tempo ao telemóvel. Com a utilização massificada dos telemóveis, também as aplicações móveis disponíveis nas app stores não param de aumentar. Há cada vez uma maior dependência das aplicações móveis.

A disponibilização destas aplicações é feita a um ritmo acelerado e procurando que o desenvolvimento seja a custo baixo. A consequência disto é muitas vezes o menosprezo pela qualidade do produto final.

Garantir a qualidade das aplicações permite assegurar a satisfação do cliente final e a sua fidelização. Permite evitar consequências graves tanto a nível financeiro como humano. Para promover a qualidade do software, é necessário testar o software garantindo que este faz aquilo que é esperado, trabalhando correctamente com um nível de qualidade elevado.

Os testes de mutação são uma técnica de injeção de faltas na implementação do código. Cada uma das faltas produzidas representa um mutante. A execução destes testes possibilita, de uma forma fiável garantir a qualidade e eficácia dos testes corridos, permitindo testar a robustez de um conjunto de casos de teste com base nessas falhas.

O presente trabalho tem como objectivo definir alguns operadores de mutação capazes de garantir faltas reais em aplicações móveis Android reais, extendendo uma ferramenta que permite automatizar a injeção desses mutantes para posteriormente serem manualmente validados, comparando os resultados obtidos nos testes ao código original e no código mutado. Com os resultados alcançados, será exequível obter a pontuação dos testes de mutação e detectar se os mutantes injectados são mortos.

Como meta final, garantir, assim o aumento da qualidade das aplicações móveis disponibilizadas.
Acknowledgements

A realização desta dissertação de mestrado contou com revelantes apoios e incentivos sem os quais este objectivo não teria sido concretizado.

À professora Ana Paiva, pela sua orientação, apoio e pragmatismo ao longo de todo o processo.

Aos docentes do MESW pela altruísmo de partilharem conhecimento aos longo destes 2 anos.

Aos meus colegas do MESW que tornaram este percurso uma alegre aventura.

À empresa onde trabalho, Critical Software que investiu em mim, dando-me a oportunidade de aprender e melhorar as minhas competências.

Aos meus amigos que se viram privados da minha companhia mas que me apoiaram e incentivaram a nunca baixar os braços.

À minha família, avós, pais, sogros, irmão, cunhados, tios, sobrinhos-primos, que pacientemente esperaram que eu alcançasse esta meta.

Por último, ao meu marido, que tanto sofreu nestes 2 anos, mas que se manteve sempre ao meu lado, dando-me a coragem e a ajuda necessária para conseguir alcançar o meu objectivo.

A todos os que passaram pela minha vida aos longo destes 2 anos, o meu bem haja.

Adélia Gonçalves
“There is no harm in being sometimes wrong - especially if one is promptly found out.”

John Maynard Keynes
# Contents

1 Introduction .......................................................... 1
   1.1 Context .............................................................. 1
   1.2 Motivation and Goals ............................................... 2
   1.3 Structure of the document ......................................... 3

2 State of the Art ......................................................... 5
   2.1 Android Programming and Android Application ................. 5
      2.1.1 Android Activities ........................................... 6
   2.2 Testing ............................................................ 8
      2.2.1 Testing Mobile Applications ............................... 9
   2.3 Mutation Testing ................................................. 14
      2.3.1 The Process of Mutation Analysis ......................... 15
      2.3.2 Mutation Operators for Android .......................... 17
      2.3.3 Mutation Operators Tools .................................. 17
      2.3.4 Previous work ............................................... 23
   2.4 Summary .......................................................... 25

3 Definition of Android Mutation Operators ......................... 27
   3.1 Selection of Android Applications .............................. 27
      3.1.1 Manual Testing .............................................. 29
   3.2 Definition of Mutation Operators ................................ 30
   3.3 Mutation Operators automation .................................. 33
      3.3.1 Liabilities ................................................... 37
   3.4 Conclusion ....................................................... 38

4 Case Study ............................................................ 39
   4.1 Automated Mutation Injection ................................... 39
   4.2 Manual Validation of the Automated Mutation Operators .... 41
   4.3 Conclusion ....................................................... 43

5 Conclusions and Future Work ........................................ 45
   5.1 Goal Satisfaction ................................................. 46
   5.2 Future Work ..................................................... 46

References ............................................................... 47
<table>
<thead>
<tr>
<th>CONTENTS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A</strong> Android applications dataset</td>
<td>51</td>
</tr>
<tr>
<td><strong>B</strong> Number of mutants generated per application</td>
<td>53</td>
</tr>
<tr>
<td><strong>C</strong> Manual testing result</td>
<td>55</td>
</tr>
</tbody>
</table>
List of Figures

2.1 A simplified illustration of the activity lifecycle........................................ 7
2.2 Block Diagram of the Architecture of the Approach [25].................................. 12
2.3 Comparison between the Studied Mutation Testing Tools [34].............................. 14
2.4 Generic Process of Mutation Analysis [29].................................................. 16
2.5 The defined taxonomy of Android bugs [36].................................................. 18
2.6 Mutation Operators [36]............................................................................ 19
2.7 Energy-aware mutation testing framework [20]................................................ 20
2.8 MDroid+ System Architecture Diagram [23]................................................... 21
2.9 MuDroid System Architecture Diagram [37]................................................... 22
3.1 Applications distribution according Google Play Stores categories.................... 28
3.2 Example of screen orientation behaviour for AlertDialog, before and after rotation (OpenTasks)............................................................... 29
3.3 Example of a show method called on Dialog or its Builder Fault and a fix in an app. [13]......................................................................................... 31
3.4 From Portrait to Landscape - ScrollView position is lost in the Android Web Opac: 1,000+ libraries application.......................................................... 32
3.5 Example of a missing Id in XML layout Fault and a fix in an app. [13].................. 32
3.6 Example of a Aged target SDK version Fault and a fix in an app. [13].................. 32
3.7 Applications distribution according Google Play Stores categories.................... 33
3.8 Mutation Tool............................................................................................... 34
3.9 Tool’s output when is not possible to inject mutant of certain type..................... 34
3.10 Tool’s output when is possible to inject mutant of certain type.......................... 34
3.11 Tool’s output example.................................................................................... 35
3.12 Regex and Mutant Operator code for Show method called on Dialog or its Builder.......................................................... 35
3.13 Regex and Mutant Operator code for Missing Id in XML layout.................... 35
3.14 Regex, constant replaced and Mutant Operator code for Aged Target SDK Version. 36
3.15 Regex Mutant Operator 2.............................................................................. 36
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Criteria for the selection of the application</td>
<td>28</td>
</tr>
<tr>
<td>3.2</td>
<td>Application’s Manual Testing Results against the event of changing the screen orientation behaviour</td>
<td>30</td>
</tr>
<tr>
<td>4.1</td>
<td>Number of Mutants Generated for each Mutation Operator</td>
<td>39</td>
</tr>
<tr>
<td>4.2</td>
<td>Number of Applications where was possible to inject each type of mutation operator</td>
<td>41</td>
</tr>
<tr>
<td>4.3</td>
<td>Mutated Application Manual Testing Results</td>
<td>42</td>
</tr>
<tr>
<td>A.1</td>
<td>Android applications dataset</td>
<td>52</td>
</tr>
<tr>
<td>B.1</td>
<td>Number of mutant applications generated by the tool developed per mutation operator and application</td>
<td>54</td>
</tr>
<tr>
<td>C.1</td>
<td>Dataset applications general information and results of Manual Testing</td>
<td>56</td>
</tr>
</tbody>
</table>
## Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>App</td>
<td>Application</td>
</tr>
<tr>
<td>AUT</td>
<td>Application Under Test</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>APK</td>
<td>Android Application Package</td>
</tr>
<tr>
<td>SMS</td>
<td>Short Message Service</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>AST</td>
<td>Abstract Syntax Tree</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Tree</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Development Kit</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The use of the smartphone is increasingly widespread. Each time a larger number of people use their smartphone to search on the most diverse themes, or to see their email and they are getting to spend about 5 hours per day in the smartphone, so 50% is the time individuals spend on digital media on mobile applications. In August 2017, there were over 3.5 billion unique mobile internet users [7].

A mobile application (app) is a software program that runs on a mobile device such as a smartphone or a tablet. The number of mobile applications is growing non stop. Over 8 million apps available on the Google Play store and the total number of Android app downloads in 2016 was 90 billion [4].

As the applications available in Google Play store grow, quality became a serious and growing problem [22]. There has also an increase of business critical mobile applications, such as mobile banking applications [24]. Many apps reach the market containing significant faults, which often result in failures during use [22]. Google recognizes this issue and has created a core app quality page where developers can find some basic aspect to evaluate the application quality and invested in an initiative to improve the stability and performance of Android devices called Android Vitals [8].

1.1 Context

The world’s most popular mobile OS is Android, as the statistics reveals [7] and Android advertises it [5]. It can be find on TV’s, phones, tablets, watch or even cars.
With the growth in popularity it is crucial to ensure the quality of the applications that can be downloaded for Android. On a daily basis, people use various applications in some cases as entertainment but in others cases with more critical functions.

Fastest releases and new features often contribute to the degradation of the quality of Android applications.

Taking into consideration the current reality, it is fundamental that companies and developers invest in testing to guarantee the quality of the applications developed in Android, avoiding failures and discontentment by the users. And with the speed of development, automating mobile application testing is one of the most appropriate option.

1.2 Motivation and Goals

The Google Play Store offers several apps. In many cases for the same context the user can choose between app A and app B. One of the motivations of the choice may be the quality of the same when in use. That is, the user downloads one of the applications and starts interacting with it. If the interaction starts to find many difficulties of use, because it is always crashing or behavior different than expected, the user will end up deleting this app and looking for a new one that best suits his needs.

The advantage of having a choice is that you can quickly move from A to B, and the disadvantage of the application having lots of issues is that users discard their use and make negative reviews of them, for example.

In this way, Google and the companies that develop the applications have a growing concern with the quality of its applications [6]. They invest in tests and recognize the importance of making them part of the development process early on.

The growth in the number of applications and the concern for their quality has resulted in a multitude of new techniques for app testing. Each of the techniques has its weaknesses and strengths, and it is essential to understand them so that one chooses the one that best fits the needs.

As will be described opportunely, there are several non-specific Android testing tools that can be used but often fail to get specific Android issues or require additional non-compensating effort [12]. Some work has been developed in the area to compare existing Android app testing techniques. These seek to evaluate the quality of the testing tools available, compare features, making it easier to use in the developer’s day-to-day.

Mutation testing is used to evaluate the quality of the software tests and it consists of modifying small pieces of software code. Each modification is called a mutant and the test injects mutants modifying the behavior of the original version. When it happens, that the behavior of the original differs from the mutant, it is called killing the mutant and the effectiveness is
measure by the percentage of mutants killed, the higher, the better quality of the test suite. New iterations can be designed to kill additional mutants. Mutation testing helps to locate weaknesses and omissions in the unit tests, testing and improving code quality and tests.

In this sense, this study, an extension of previous research work done in this field, is focused on mutation testing in Android applications. That is, to find real failures resulting from the non-use of good development practices in Android, through the simulation of these same failures using mutation operators. The mutation operators allow to inject these failures into applications and to evaluate them. The mutant injected in the source code of the application under test (AUT) reveals if the manual tests are able to detect and kill the mutants and if this mutant operators become an asset for mutation testing in Android.

If these mutant operators allow to simulate real faults, then developers test suits can anticipate bugs and corrected them in the source code and increase the trust of the quality of their applications. One of the main goals of this research is to define and automate a set of mutation operators related to rotation screen of Android Applications, simulating faults identified by Amalfitano et al. [13].

This project is divided into these phases:

1. Review state of art;
2. Analyze typical Android programming error by studying the Android apps available;
3. Define mutation operators;
4. Evaluate the effectiveness of the mutation operators by manual testing;

The first phase is important for understanding the progress that have been done. Then the focus is to analyze the source code of Android application and identify the errors associated with bad practices of Android programming. Using that information is possible to define the mutation operators for finally evaluate and validate the mutation operators manually. For manual testing, mutant operators are going to be generated using an automation tool. The mutation source code is going to be compiled and manually handling mutant app and original app compare if there are any differences of behavior between them.

1.3 Structure of the document

After the introduction, chapter 2 presents the current state of the art of Android programming, testing mobile apps and mutation testing. On chapter 3 is defined the mutant operators, the dataset is described and the methodology of the research. Chapter 4 presents an experiment to assess the quality of the mutant operators, i.e., check if they reproduce real failures and if they can be detected by manual tests.
Chapter 2

State of the Art

This chapter is divided into three main sections: in section 2.1 is presented some aspects about Android and Android Application, in section 2.2 is summarized the current state of testing including mobile testing and in section 2.3 the state of the art of the mutation testing and tools for mutation testing for Android. There is a final section presenting conclusions.

2.1 Android Programming and Android Application

Android operating system is a multi-user Linux system with a unique Linux User ID who sets permissions for all the files in an application and the application code runs in isolation from other applications because of the Virtual Machine (VM) within an isolate process.

Android application (Android app) can be developed in different languages such as Kotlin, Java and C++. The code, data and resources files are compiled by the Android SDK tools into an APK. The APK file is the file use to install the application. Android app uses a concurrent event-driven model and only the main thread has access to GUI objects.

It is possible for an application to share the user ID with a different application facilitating the access to the files by sharing the same certificate and VM. It is also possible to an application request permissions to access to data such as SMS messages, camera, contacts, etc.

Android is a component based language and an app consist of four kind of components: activities, services, broadcast receives and content providers. The components are going to be describe further.
2.1.1 Android Activities

Android app usually consists of a suite of Activities. An activity represents a single screen with a user interface and works as the entry point for interact with the user. Every activity can have different states: Created, Started, Resumed, Paused, Stopped or Destroyed [2].

The navigation on an application is the transition through different states in the lifecycle, where the callbacks allow the activity to know that the state has changed. Each callback allows to perform specific work and avoid crash or consuming values without needing.

Depending on the complexity of your activity, developer probably doesn’t need to implement all the lifecycle methods but it is important to understand how transitions between states can be handled.

The conceptual and implementation information about the callback methods used during the activity lifecycle is going to be explained [18]:

- **onCreate()** - This callback is fired when the system first creates the activity, performing startup logic that happens only once for the entire life of the activity. The use of method annotation ensures that any step code is perform. After the `onCreate()` method finishes execution, the activity enters the Started state, and the system calls the `onStart()` and `onResume()` methods in quick succession [2].

- **onStart()** - The `onStart()`, a very quickly method, shows the activity to the user and all the components receive the ON_START event. Once this callback finishes, the activity enters the Resumed state, and the system invokes the `onResume()` method.

- **onResume()** - This method is invoked when the activity is on the Resumed state. It is here where the app interacts with the user. It will stay in this state until something happens to take the focus from the app, like receiving a phone call or navigating to another application. When such an interruptive event occurs, the activity is set to the Paused state.

- **onPause()** – The system invokes this callback when the activity is in the Paused state, so user is leaving the application but it possible that the activity is being destroyed and it is still running in the background.

- **onStop()** – This method is fired when the activity is no longer visible to the user and enters the Stopped state.

- **onDestroy()** – This callback is invoked before the activity is destroyed. If there are any resources that are yet to be released, they should be so here.

- **onRestart()** – After the activity is stopped and before being started again this method is called and it is followed by the `onStart()` callback.
The misunderstanding of the lifecycle of an activity may lead to loss of information and memory leaks, that may cause the application to crash.

Android has the following application components:

1. **Services**

   A service is a general-purpose entry point for keeping an app running in the background for all kinds of reasons. It is a component that runs in the background to perform long-running operations or to perform work for remote processes [3].

2. **Broadcast receiver**
A broadcast receiver is a component that enables the system to deliver events to the app outside of a regular user flow, allowing the app to respond to system-wide broadcast announcements [3]. They are an entry into the application without being currently running, like notifications.

3. **Content provider**

A content provider manages a shared set of app data that you can store in the file system, in a SQLite database, on the web, or on any other persistent storage location that your app can access. Through the content provider, other apps can query or modify the data if the content provider allows it [3].

When the system starts a component, it starts the process for the application but it is not possible to activate a component from another application because of the file permissions.

The components activities, services, broadcast receiver and content provider are activated by an intent. An intent is an asynchronous message.

Before the Android system can start an application component, the system must know that the component exist by reading the `AndroidManifest.xml` where all its components are declared.

The Android application requires resources as images or audio files. SDK build tools define a unique integer ID that is used as reference from code to the resource, which is very helpful for development.

### 2.2 Testing

Testing is the process used in software development to guarantee the quality of the solution in terms of functional and non functional requirements. Test cases require a mechanism to determine the test outcome, the expected result. Test cases can be designed from different points of view:

**White-box testing**: focus on the internal structure of the application (source code level). Test cases involve assigning values to the variables and measuring what lines of code get executed. There are two techniques to derive test cases: statement coverage (if the test case executes every line of code in the program, it is called 100% statement coverage) or decision coverage (if the test cases execute both the decisions, it is called 100% decision coverage).;
Black-box testing: focus on the external behavior of the application without considering implementation. The testers have no knowledge of how the system or component is structured inside the box. In black-box testing the tester is concentrating on what the software does, not how it does it;

Grey-box testing: combination of white box and back box testing. Test are defined at the user level but with the knowledge of the internal structure of the application. Test case will be defined concern with the interaction of the main components;

There are usually four levels on witch test cases can be defined:

Unit Testing – focus on a specific unit of the program. Usually it follows a white-box testing approach.

Integration Testing – tests the interaction between components of the application;

System Testing – consider the program as a whole to see if meets all the requirements and quality standards.

Acceptance Testing – final test that decides if the application is complete and ready to be deployed.

2.2.1 Testing Mobile Applications

Mobile apps testing refers to “testing activities for native and Web applications on mobile devices using well-defined software test methods and tools to ensure quality in functions, behaviors, performance, and quality of service, as well as features, such as mobility, usability, interoperability, connectivity, security, and privacy” [28].

There are several unique requirements discern mobile applications from conventional software testing:

• Mobile apps must function anytime, anywhere;

• Mobile apps must work properly across platforms (phone, watch, tablet);

• Mobile apps must include multiple input channels (voice, touch, keyboard);

• Mobile apps must function in diverse network connectivity contexts.

There different mobile testing approaches that can be used for testing different activities:

• Emulation-based testing – using a mobile device emulator which creates a virtual machine version of a mobile device
• **Device-based testing** – requires setting up a testing laboratory and purchasing real mobile devices;

• **Cloud testing** – build a mobile device cloud for testing;

• **Crowd-based testing** – a crowd-based testing infrastructure and a service management server to support diverse users, where users test ad-hoc.

Mobile apps can be native apps that are deployed and executed on mobile devices and usually depend on native APIs and web apps that consist of an app server and client software executed over Web browsers though which users can access application services or hybrid app (part native apps and part web apps).

Testing this types must take into account their own differences. For example, the concern for web apps is to validate the quality on different browsers, and for native apps is to validate the quality of mobile apps downloaded and executed on select mobile platforms on different mobile devices. There are different types of testing strategies that can be applied to mobile applications [19]:

• **Performance and reliability testing** - mobile device resources defines performance and reliability of mobile applications. For preventing degradation resources and connectivity state should be monitored.

• **Memory and Energy testing** - Limited resources as battery and memory can generate memory leaks and abusive consumption that must be avoid for mobile applications working properly.

• **Security testing** - Mobiles can be connected on different networks so they became much more vulnerable to attacks. User should trust that their data, contacts, schedule, private date are protected and not vulnerable to attacks so it must be assured the mobile security.

• **GUI testing** - Graphical User Interface (GUI) testing of mobile applications should test whether different devices provide an adequate rendering of data, and whether native applications are correctly displayed on different devices. The idea is to capture user interaction with the application and verify the behavior of the application. The iMPAcT tool is an example of a GUI testing tool, that it is going to be presented latter but there are others crash testing and regression testing of Android applications.

• **Product line testing** - Mobile applications can be used in different size of screen depending on the mobile or tablet or even watch. Tests must be conducted to cover the multitude of mobile devices.
2.2 Testing

2.2.1.1 Testing Mobile Applications Main Issues

Since the behaviour of an Android application is event-driven, most of the approaches already available for Event-Driven Software testing are still applicable for Android. However, it is necessary to assess how these techniques can be adopted to carry out cost-effective testing process in the Android platform [14].

The main issues about testing Mobile applications, especially Android are the diversity of screens size and density in total of 24093 distinct Android devices available on the market at 2015 [1]. Mobile apps can run into watch, phone or tablet and now there are available touch screens that need to be responsiveness.

There are also plenty of operating systems still in use for mobile apps. Mobile connectivity is another issue because the mobiles can be connect to different networks with different speed, security and reliability, so different scenarios of testing need to be considered [16].

Android mobiles can have different levels of energy consumption and autonomy which influence the android apps performance.

Besides that, testers should also take into account regional trails (internationalization issue) and think about keeping the interaction clean and simple for the user, and at the same time display all the necessary information with no crash (usability issue).

2.2.1.2 iMPAcT Tool

"The iMPAcT tool automates the testing of recurring behaviour (UI patterns) present on Android mobile applications" [25] [27] [26]. The Android applications should be tested with the objective of finding UI patterns automatically and to test them by using the test strategies associated with patterns. The four main characteristics for testing approach supported by the iMPAcT tool are:

1. The goal is to test recurring behavior (UI patterns);
2. The whole process is completely automated;
3. The process is iterative combining automatic exploration [15], reverse engineering and testing;
4. The reverse engineering process is fully dynamic;

According to [33] and [30] the tool has these patterns implemented:

- **Side or Navigation drawer pattern** – This is a UI panel that shows your apps main navigation menu, and it is hidden when not in use, but appears when the user swipes a finger from the left border or on click of the icon.

  When it is open, occupies the full height of the screen and this is what is tested.
Orientation Pattern — The mobile devices has two orientations: landscape or portrait. The rotation from one to another involve the change of layout, and information cannot be lost. The pattern tests if no information inputted is lost and no widget disappears. The iMPAcT tool compares the screen before and after rotation for verifying if the screens are identical and an error is detected when a widget is not shown, the user input is presented in the first screen and not in the other, one screen is pop up but the other is not, the side bar is only present in one of the screens.

Tab Patterns — Facilitate the navigation between different views. iMPAcT Tool test this pattern against some guidelines to ensure the correct implementation. When detecting a tab, the tool test the implementation by verifying if [25]:

- There is only one set of tabs per activity;
- The tabs are correctly positioned on the upper part of the screen;
- The horizontal swipe notion on the screen changes the selected tab and nothing else.

Resource Dependency Pattern — External resources are needed for several applications to work properly. So it is necessary to check if they are available or not when the resources are called and when the resources are not available it is important to check if the app doesn’t crash.

Figure 2.2: Block Diagram of the Architecture of the Approach [25].
The tool checks if the app is using a certain resource and if so, turns resource off. Then the app state is verified to attest whether the unavailability of resources crashed the app or caused some error.

- **Background pattern** — Pressing the home button when using an application should show the home menu saving status for the application still running. For user is possible to go back to the app and the screen presented should be the same and the app should be at the same state as before. This is what is tested on the tool: simulates the click in the home button, sending the AUT to the background. Then opens the app again and compares the screen sent to background to the one reopened looking for the same state.

- **Action Bar Pattern** — This structure provide interactive elements to user with main functions: a dedicated space to give app an identity and indicates the user’s location, access to import action in a predictable way and support navigation and preview switching. The action bar is placed at the top of the screen and presents the app title and floating menu with the most important actions. The goal of this test is confirming the correct position of the Action Bar on screen and it flows. iMPAcT tool does not check formally if action bar exists.

- **Up Pattern** — It should be possible to go to the pattern screen in the hierarchy by using the up button present in Action Bar for become easy to go to the main screen. The test verifies the existence of the Up button at the Action Bar and when it is clicked it sends the app to the current screen logic parent on the hierarchy.

- **Back Pattern** — Back button helps user to move backward through the history of screens previously visited. The AUT checks if the AUT uses the back button provided and not a personalized one and checks if it goes back to the previous visited screen when the back button is clicked.

- **Call Pattern** — A mobile device is also a mobile phone and for that gets mobile calls. The Call Pattern checks if an app does not crash when getting an incoming Call. Also, it checks if the state of the screen before the incoming call is equal to the state of the screen after such incoming call.

There are three phases for the iMPAcT tool approach an execution:

1. **Exploration** - Exploring the application and identifying which events can be fired randomly and fires it;

2. **Pattern Matching** - Analyzing the current screen after the event is fired to verified if the pattern is present;
3. **Testing** - Applying the test strategy if a pattern is detected. If the test is succeeded the pattern is correctly implemented. There are two main artifacts produced at the end of the exploration: the report of the exploration with the log of the exploration done on the AUT and a model of the behavior observed during the exploration. The set of events available for execution depends on the exploration mode defined by user: **execute once** (the possible events are fired only once), **prioritize not executed** (the events not fired have priority over the ones that have already been fired), **prioritize not executed and list items** (events associated with lists have higher priority) and **all events** (every event will be fired). An improvement of the exploration process is presented in [15]. The iMPAcT tool is a valuable Android testing framework because it does not require access to source code of the application, multiple patterns are implemented and different exploration algorithms are available.

One way to assess the quality of a testing tool (or test suite generated by such tool) is through mutation testing. In [31], an experiment tries to assess if iMPAcT tool is able to detect failures related to the wrong implementation of the background/foreground behavior of Mobile Apps. It was an experiment performed over 50 apps publicly available.

The results of the experiment pointed out some possible improvements for iMPAcT tool.

### 2.3 Mutation Testing

Mutation Testing is a fault-based testing technique which provides a testing criterion called the “mutation adequacy score”. The mutation adequacy score can be used to measure the effectiveness of a test set in terms of its ability to detect faults.

In this white box testing technique, the tester makes copies of the system under test with some syntactic changes (faults).

There are different testing tools for mutation testing for different languages:

![Comparison between the Studied Mutation Testing Tools](image)

Figure 2.3: Comparison between the Studied Mutation Testing Tools [34].
Mutation testing promises to be effective in identifying adequate test data which can be used to find real faults. Mutation Testing benefits are better fault exposing capability compare to other test coverage criteria and a good alternative to real faults which can provide a good indication of the fault detection ability of a test case.

One main issue about mutation testing is that it is impossible to generate mutants representing all of the potential faults because it involves a high computational cost of executing the enormous number of mutants against a test set and takes an enormous amount of time in practice.

The other issue is the amount of human effort involved and is difficult to fully automatic the equivalent mutant, known as the Equivalent Mutant Problem.

As a result, reducing the number of generated mutants without significant loss of test effectiveness has become a popular research problem:

1. Mutant Sampling – Proposed by Acree and Budd, in this approach all possible mutants are generated first as in traditional Mutation testing but then randomly chooses a small subset of mutants from the entire set.

2. Mutant Clustering – This approach chooses a subset of mutants using clustering algorithms. A clustering algorithm is then applied to classify the first order mutants into different clusters based on the killable test cases. Each mutant in the same cluster is guaranteed to be killed by a similar set of test cases. Only a small number of mutants are selected from each cluster to be used in Mutation Testing.

3. Select Mutation – reduction in the number of mutants can also be achieved by reducing the number of mutation operators applied. This is the basic idea, underpinning Selective Mutation, which seeks to find a small set of mutation operators that generate a subset of all possible mutants without significant loss of test effectiveness [32].

2.3.1 The Process of Mutation Analysis

In mutation analysis, from a program p, a set of faulty programs p’ called mutants, is generated by a few single syntactic changes to the original program p [21].

A mutation operator is transformation rule that generates a mutant from the original and are design to modify variables and expressions by replacement, insertion or deletion operators.

One off the generic classification of mutation operators is the follows:

1. Specification mutation — applied at the software design level

2. Program mutation (expression level, statement-level or other like SQL-specific or concurrent mutation) — applied to unit level and integration level;
Figure 2.4: Generic Process of Mutation Analysis [29].

- From the original program create Mutants P’;
- The test set is executed against the original program p to check the test case;
- If p is incorrect, should be fix before running other mutants;
- It is possible to improve the test set T for killing more mutants but it is not possible to kill all the mutants (Equivalent Mutants).

A High quality test case is that which can find more number of faulty versions of system under test. The Mutation score associated with a test suite ‘T’ and mutants ‘M’ is simply given by equation

\[ \text{Mutation score} = \frac{\text{Number of killed mutants}}{\text{Total number of mutants} - \text{Equivalent mutants}} \times 100 \]

or

\[ \text{Mutation score} = \frac{\text{Number of killed mutants}}{\text{Number of nonequivalent mutants}} \]

The result of this fraction ranges from 0 to 1. The result equals to 1 is corresponding to all mutants killed by the test suite (best case) and result equals to 0 corresponds to none mutants killed (worst case). The closest the result to 1 the better is the test suite.
2.3.2 Mutation Operators for Android

Mutation analysis cannot be performed the same way for Android apps as for traditional Java programs. One reason why the process must be different is that, whereas Java mutation analysis tools mutate only Java files, we have designed Android operators that also apply to XML layout and configuration files. A second reason is because Android apps require additional processing before being deployed. Traditional Java mutation analysis tools typically compile mutated Java source files to bytecode Java class files. The Java bytecode files are then dynamically linked by the language system during execution.

Android apps have the additional requirement that each Android mutant must be compiled as an Android application package (APK) file so that it can be installed and executed on mobile devices and emulators. This has a significant impact on how mutation analysis tools run.

Deng et al. try to define good mutants operators that can lead to very effective test. For that, it was analyze every systematic element of the language being mutated, and designs mutants to modify the syntax in ways that typical programmers might make mistakes [22].

The mutant operators defined:

- **Intent Mutation Operators** - operation performed on the android components. Used for launch an activity. Intent Payload Replacement and Intent Target Replacement are the two Mutation Operators defined.

- **Event Handler Mutation Operator** – Recognize and reply to events. `OnClick` Event Replacement, `OnTouch` Event Replacement are the two Mutation Operators defined.

- **An activity Lifecycle Mutation Operator** – Related to state transitions. Lifecycle Method Deletion is one of the Mutation Operators defined.

- **XML Mutation Operator** – Android uses XML files. Button Widget Deletion, EditText Widget Deletion and Activity Permission Deletion are the three Mutation Operators defined.

Vasquez et al. after defining the taxonomy of Android bugs by studying several sources, defined 38 mutant operators.

They evaluate these same mutant operators with various mutation testing tools including Java and Android [36].

2.3.3 Mutation Operators Tools

The literature presents some tools developed in this area, such as the case of µDroid, MDroid+, and MuDroid specific for Android that are going to be describe further.
However, there are mutation testing Java tools that can be used in the unit level testing that can be tailoring to Android, e.g. Major, MuJava and PIT but studies show that this tools don’t get as good results as the ones specific for Android because Android applications are not executed on the standard Java virtual machine.

There are also some GUI Testing tools for Android based on Mutant Operators such as EvoDroid, MobiGUITTAR, AndroFrame, Sapienz, Monkey or DynoDroid which are black box testing tools that detect a large number of crashes defects among Android applications.
2.3 Mutation Testing

2.3.3.1 $\mu$Droid

$\mu$Droid is a framework for energy-aware mutation testing for Android applications. This tool was developed with the intention of helping the removal of the energy defects of the applications, so helping developers on the evaluation the quality of their tests for revealing energy defects. $\mu$Droid, for energy-aware mutation testing of Android apps, consisting of three major components:

- Eclipse Plugin that implements the mutation operators and creates a mutant from the original app;
- Runner/Profiler component that runs the test suite over both the mutated and original versions of the program, profiles the power consumption of the device during execution of tests, and generates the corresponding power traces (i.e., time series of profiled power values);
- Analysis Engine that compares the power traces of tests in the original and mutated versions to determine if a mutant can be killed by tests or not [20].

Mutation operators are designed based on an energy defect model, and were design by studying different repositories looking for energy defects. The mutation operators defined are:

<table>
<thead>
<tr>
<th>Mutation Operator</th>
<th>Det.</th>
<th>Cat.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActivityNotDefined</td>
<td>Text</td>
<td>A/I</td>
<td>Delete an activity `<a href="">android:name=&quot;Activity&quot;/</a> in the Manifest file</td>
</tr>
<tr>
<td>DifferentActivityIntentDefinition</td>
<td>AST</td>
<td>A/I</td>
<td>Replace the Activity class argument in an Intent instantiation</td>
</tr>
<tr>
<td>InvalidActivityName</td>
<td>Text</td>
<td>A/I</td>
<td>Randomly insert typo in the path of an activity defined in the Manifest file</td>
</tr>
<tr>
<td>InvalidKeyIntentPutExtra</td>
<td>Text</td>
<td>A/I</td>
<td>Randomly generate a different key in an Intent putExtra(key, value) call</td>
</tr>
<tr>
<td>InvalidXmlLabel</td>
<td>Text</td>
<td>A/I</td>
<td>Replace the attribute &quot;android:label&quot; in the Manifest file with a random string</td>
</tr>
<tr>
<td>NullIntent</td>
<td>Text</td>
<td>A/I</td>
<td>Replace an Intent instantiation with null</td>
</tr>
<tr>
<td>NullValueIntentPutExtra</td>
<td>Text</td>
<td>A/I</td>
<td>Replace the value argument in an Intent.putExtra(key, value) call with new Parcelable[0]</td>
</tr>
<tr>
<td>WrongMainActivity</td>
<td>Text</td>
<td>A/I</td>
<td>Randomly replace the main activity definition with a different activity</td>
</tr>
<tr>
<td>MissingPermissionManifest</td>
<td>Test</td>
<td>AP</td>
<td>Select and remove an &lt;uses-permission /&gt; entry in the Manifest file</td>
</tr>
<tr>
<td>NotParcelable</td>
<td>AST</td>
<td>AP</td>
<td>Select a nonParcelable class, remove implementations Parcelable and the (override annotations)</td>
</tr>
<tr>
<td>NullGPSLocation</td>
<td>AST</td>
<td>AP</td>
<td>Inject a null GPS location in the location services</td>
</tr>
<tr>
<td>SGVersion</td>
<td>Test</td>
<td>AP</td>
<td>Randomly mutate the integer values in the id/version-related attributes</td>
</tr>
<tr>
<td>WrongStringResource</td>
<td>Test</td>
<td>AP</td>
<td>Select a &lt;string /&gt; entry in res/values/strings.xml and mutate the string value</td>
</tr>
<tr>
<td>Null_BackgroundServiceReturn</td>
<td>BES</td>
<td>Assign null to a response variable from a back-end service</td>
<td></td>
</tr>
<tr>
<td>BluetoothAdapterWakeUpEnabled</td>
<td>C</td>
<td>Replace a BluetoothAdapter instance with null</td>
<td></td>
</tr>
<tr>
<td>BluetoothAdapterAdapter</td>
<td>C</td>
<td>Replace a BluetoothAdapter instance with null</td>
<td></td>
</tr>
<tr>
<td>InvalidIDIdentifier</td>
<td>AST</td>
<td>GP</td>
<td>Assign a cursor to null if it is closed</td>
</tr>
<tr>
<td>InvalidMethodCallArgument</td>
<td>AST</td>
<td>GP</td>
<td>Randomly mutate a method call argument of a basic type</td>
</tr>
<tr>
<td>NullTerminable</td>
<td>AST</td>
<td>GP</td>
<td>Select a terminable class, remove implementations Terminable</td>
</tr>
<tr>
<td>NullMethodCallArgument</td>
<td>AST</td>
<td>GP</td>
<td>Randomly set null to a method argument</td>
</tr>
<tr>
<td>InvalidInstance</td>
<td>AST</td>
<td>GUI</td>
<td>Set a random instance to a date object</td>
</tr>
<tr>
<td>InvalidOnClickListener</td>
<td>AST</td>
<td>GUI</td>
<td>Remove the text content in a layout File</td>
</tr>
<tr>
<td>InvalidOnClickListener</td>
<td>AST</td>
<td>GUI</td>
<td>Replace the id argument in an Activity.findViewById(CharSequence) call</td>
</tr>
<tr>
<td>InvalidOnClickListener</td>
<td>AST</td>
<td>GUI</td>
<td>Set a view attribute from a View to false</td>
</tr>
<tr>
<td>InvalidOnClickListener</td>
<td>AST</td>
<td>GUI</td>
<td>Set a view attribute from a View to false</td>
</tr>
<tr>
<td>InvalidOnClickListener</td>
<td>AST</td>
<td>GUI</td>
<td>Set a view attribute from a View to false</td>
</tr>
<tr>
<td>InvalidOnClickListener</td>
<td>AST</td>
<td>GUI</td>
<td>Set a view attribute from a View to false</td>
</tr>
<tr>
<td>InvalidOnClickListener</td>
<td>AST</td>
<td>GUI</td>
<td>Set a view attribute from a View to false</td>
</tr>
<tr>
<td>InvalidOnClickListener</td>
<td>AST</td>
<td>GUI</td>
<td>Set a view attribute from a View to false</td>
</tr>
<tr>
<td>InvalidOnClickListener</td>
<td>AST</td>
<td>GUI</td>
<td>Set a view attribute from a View to false</td>
</tr>
<tr>
<td>InvalidOnClickListener</td>
<td>AST</td>
<td>GUI</td>
<td>Set a view attribute from a View to false</td>
</tr>
<tr>
<td>InvalidOnClickListener</td>
<td>AST</td>
<td>GUI</td>
<td>Set a view attribute from a View to false</td>
</tr>
<tr>
<td>InvalidOnClickListener</td>
<td>AST</td>
<td>GUI</td>
<td>Set a view attribute from a View to false</td>
</tr>
<tr>
<td>InvalidOnClickListener</td>
<td>AST</td>
<td>GUI</td>
<td>Set a view attribute from a View to false</td>
</tr>
</tbody>
</table>

Figure 2.6: Mutation Operators [36].
Figure 2.7: Energy-aware mutation testing framework [20].

- Location Mutation Operator – location aware apps;
- Connectivity Mutation Operator – for example network or Bluetooth;
- Wakelock Mutation Operator – mechanisms to maintain the application awake;
- Display Mutation Operator – screen awake;
- Recurring callback and loop Mutation Operator – for repeating tasks;
- Sensor Mutation Operator - Wakeup sensor;

2.3.3.2 MDroid+

MDroid+ is a Mutation Testing tool, implemented in Java command line utility where user can select the Mutation Operator to use.

Mutation Operator should be defined according to the flaws of the projects and for this a study was made to several repositories of bug fixing, open source projects, Stackoverflow, reviews in Google Play, in order to identify the most impacting failures.

This enabled the extraction of 38 mutation operators, divided in 10 categories:

- Activity/Intents
- Android Programming
- Back-End Services
- Connectivity
2.3 Mutation Testing

- Data
- Database
- General Programming
- GUI
- I/O
- Non-Functional Requirements

![MDroid+ System Architecture Diagram](image)

Figure 2.8: MDroid+ System Architecture Diagram [23].

For implemented operators targeting one of the Android resource XML files, the structure of each XML file is analyzed and a pattern matching process for different attributes within the XML is used. However, for operators that are applied to the Java source code, a two-phase AST-based and text-based analysis is utilized that is capable of identifying the location of target API calls.

The identification of API call sites is implemented utilizing the visitor design pattern, allowing for extensible, decoupled operations to be performed on the AST of a target app. This helps to ensure that MDroid+ is capable of supporting additional operators in the future that may require more advanced AST analysis.
After the AST-based location of specific API calls, a fine-grained, text-based pattern matching is performed on identified API calls to derive the precise textual location where the mutation operator transformation will be applied. The end result of the PFP derivation process is a list stipulating all potential injection points in code of the Android-specific mutation operators. The MDroid+ has the advantage of being easy to add new Mutation Operators.

2.3.3.3 MuDroid

MuDroid is a mutation testing tool for Android apps developed by Yuan Wei. MuDroid contains three parts:

- Mutant generator – generate APK mutants from the APK file. The mutant generator consists three components: Apktool, a mutation analyzer and a mutant selector;

- Interaction simulator - using the APK mutants generated by Mutant Generator and produces screenshots as output. The result of these events is recorded in screenshots. Interaction simulator consist components: adb and interaction generator.

- Result analyzer checks if the mutant is killed. Result Analyzer generates a report that contains mutation score and mutants information for a given APK by comparing screenshots. Result Analyzer consist on the components image checker and Report Generator.

![Figure 2.9: MuDroid System Architecture Diagram [37].](image)

One of the limitation on this framework is that this approach could not be involved as a part of Android APK compilation since it does not required the source code. The other one is since targets at Smali code instead of manipulating Android bytecode, that is less efficient.
2.3 Mutation Testing

The experience was conducted in a small number of subjects and implements six general mutation operators that is not enough for the Android complex features.

2.3.3.4 Mutation Operators automation

There are several techniques to automate the injection of mutation operators:

- **Bytecode**
  
  Code from a computer program written in the Java language is compiled into an intermediate form of code called bytecode, which is interpreted by Java Virtual Machines (JVMs), allowing them to run on any system that has a JVM (.class files). Generating bytecode has a better performance than interpreting abstract syntax tree (AST) [9][35]. It is possible to transform classes without the java source code.
  
  The bytecodes are difficult to modify once they are emitted [35].

- **Abstract Syntax Tree (AST)**
  
  For interpreting AST it is need to traverse the tree, inspect nodes, determine the type of nodes, check the type of any operands, verify legality and decide which AST operator applies before performing any action.
  
  By creating an AST on a Java file, control is greater over browsing and modifying a Java program. For complex analysis or modification of Java files, the AST is a good technique. ASTs are more malleable than bytecodes [35].

- **Source Code**
  
  If the java source code is available, it is easier to analyse the source code and faster to introduce some modifications directly into code. A code editor is enough to do it.

  All techniques are valid to inject mutation operators into Android app and each technique has different capabilities and trade-offs, depending what it is necessary to be done. For example, as described above, MDroid+ operations are performed on the AST of a target app.
  
  Reading bytecode is not so easy or intuitive than source code. And, bytecode has better performance than AST. Looking at the tradeoffs of the various techniques, it seems that the source code becomes the most advantageous.

2.3.4 Previous work

In Liliana Ribeiro [33] and João Gouveia [17] thesis and João Gouveia [31] article, the same problem was proposed and researched. Both choose the application using several criteria such
as availability at Google Store and Portugal, have a Google store rating $\geq 3.5$, have a number of ratings $\geq 100$, use gradle, be Android native, have a GUI and be in a Western European Language.

The mutation operators, defined for all research, were based on the study of good programming practices in Android, including the with special focus on the lifecycle of Android apps. The applications to use for testing must have source code accessible and a GUI interface available for being tested by the iMPAct tool.

The patterns tested by [33] were Side or Navigation Drawer pattern, Orientation pattern and Tab pattern and two problems were found in the iMPAct tool: the tool is not able to detect errors in the search widget and cannot reach all the possible screens of an application.

On the study [17], four different mutation operators capable of affecting the background behavior of Android application were defined by taking into account Android programming best practices: onSaveInstanceState, EditText, Spinner, Intent.

The mutation operators were automatized and then run against the background pattern defined at iMPAct tool where most of the mutant apps were detected. By doing that, two problems were found: the iMPAct tool able to reach all the activities/screens in an application, which may lead to undetected incorrect behaviours and some input widgets is not detected the loss of information when the application goes to background and comes back to foreground [17].

In the article [31], three mutation operators were presented to test a specific behavior of mobile applications related to the preservation of uses transient UI state when apps are sent to back and than return to foreground, that is, each time an application comes back from foreground the user must find the application as it was sent to background.

The three mutation operators were manually or automatically applied to a set of applications and the quality of the tests validated by the iMPAct tool.

The first mutant operator defined is related to the onSaveInstanceState(Bundle) method that is called when activity begins to stop. The key-value pair added to the Bundle object saves instance state information for the activity.

onRestoreInstanceState(Bundle) or onCreate() methods are called to restore the saved state by onSaveInstanceState(Bundle) when application backs from foreground. The key-value pair added to the Bundle object is saved in the case of unexpected error. By adding to this method a .clear(), the Bundle is cleared and when the application calls onRestoreInstanceState(Bundle) or onCreate() method, Bundle is going to be empty.

When the developer persists data in local storage using shared preferences, if it is empty, when the application tries to read shared preferences no information is available to be presented. This was the second defined mutation operator where .clear().commit() by erasing the shared preferences will cause the user will find a different state from the previous.
The third mutant operator changes the `onPause()` method by creating an intent to other activity and starts the activity inside the `onPause()` method. `onPause()` method is updated or created when non-existing. Another activity is started when the app is paused. This happens when activity A starts and covers the activity B, sending activity B to background and `onPause()` method is called.

In conclusion, it was possible to generate automatically mutants and apply them to real apps. Mutant operator 3 was the one that generated more mutants. It was also possible to detect some new features that can improve iMPAct tool.

2.4 Summary

While Android applications do not stop increasing in quantity, it is essential to ensure that it also happens in quality. Testing is critical to ensure the success of applications.

Mutation testing in Android is an area with a lot of research potential and taking its first steps. The few tools available as MuDroid, MDroid+ or µDroid implements specific mutation operators for Android programming.

The iMPAct tool is an easy tool to test specific characteristics of Android applications through their GUI. This tool may be an inspiration to create new mutation operators. Also it can be used to try killing mutants resulting from mutation testing. And finally, the quality of the iMPAct tool itself can be assessed by mutation testing (trying to check if the tool is able to kill the mutants generated by mutation testing tools).
Chapter 3

Definition of Android Mutation Operators

For defining Android mutation operators, a process was conducted to achieve the objectives of research. Android applications were selected based on the future work from the Amalfitano et al. article [13]. Android Mutation Operators were defined, the mutation operators were injected and the results evaluated.

3.1 Selection of Android Applications

The data set used for this study has also been used in [17] and [33] work. The completed list can be reviewed in A.1. The biggest concern for defining the dataset was the possibility to evaluate the correctness implementation of the guidelines to the event of changing the screen orientation and being possible to exploit the Android-specific fault classes to develop new mutation operators for testing of Android applications defined in the Amalfitano et al article [13].

Some manual experimentation was made on that data set and some of applications used by [17] were removed because they were not available anymore or it was not possible to compile them, in order to find applications that could be used for this study.

The final criteria used in this study is detailed below in table 3.1.

The applications selected, a total of 40 applications, have the categories distribution according to Google Play Stores in accordance with fig. 3.1. The dataset selected take in consideration the diversity of categories for a broad purpose and use.

The dataset includes only applications that could be compiled using Android Studio 3.1 and ran both in an emulator (Galaxy Nexus API 27 Android 8.1.0) and a real device (Huawei P20
Available in Google Play Store | The application can be downloaded in the Google Play Store
Available in Portugal | Application must be available in Portugal
Be open Source | The application code must be available to insert mutant and be compiled.
Have a Google Play Store ranking >=3.5 | For assurance some degree of quality, the application’s minimum rating is 3.5
Have Google Play Store downloads >=10000 | The application’s minimum downloads is 10000 in order to ensure that at least multiple amount of people are using it
Be Android Native | To define mutation operators for evaluating the correctness Android code implementation, the application code must be Android native.
Use Gradle | The application must use Gradle to simplify the build of the application.
Have a GUI | The application must have a Graphical User Interface.
Be in an Western European Language | The application must be in an Western European Language so its UI can be understand.

Table 3.1: Criteria for the selection of the application.

Figure 3.1: Applications distribution according Google Play Stores categories.
3.1 Selection of Android Applications

lite with Android 8.0). Some of the initial applications were removed from the dataset because of existing compiling issues as generic build failed errors, gradle sync failed.

3.1.1 Manual Testing

Regarding the guidelines for changing the screen orientation in Android applications, it is important to define what is a correct or incorrect behaviour. During the execution time, some configuration can change and the re-start behaviour is called, projected for automatically recharge application.

For the correct re-start, before destroying the activity, `onSaveInstanceState()` is called for saving the data. Then, the previous state is restored by calling `onCreate()` or `onRestoreInstanceState()`.

The application should restart without losing data or state. Figure 3.2 shows an alert dialog before and after rotation.

![Figure 3.2: Example of screen orientation behaviour for AlertDialog, before and after rotation (OpenTasks).](image)

As an example of an incorrect changing of the screen orientation in Android applications occurs when an information inputted in landscape it is lost after rotating to portrait.
3.1.1.1 Manual event of changing the screen orientation behaviour Testing

For the completed dataset and to exploit this failure, every activity was tested rotating the screen, by using the following process:

1. Open new activity
2. Rotate activity to landscape
3. Rotate activity to portrait
4. Check if all the widgets that existed before are still present and the information and data is still the same
5. Rotate activity to landscape
6. Check if all the widgets that existed before are still present and the information and data is still the same
7. Go to step 1 until no more activities can be explored.

If during the process any errors or malfunction was detected the application was considered to have an incorrect mobile-specific event of changing the screen orientation behaviour, in accordance with the Android Guidelines.

<table>
<thead>
<tr>
<th>Type</th>
<th>Number of applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct</td>
<td>34</td>
</tr>
<tr>
<td>Incorrect</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 3.2: Application’s Manual Testing Results against the event of changing the screen orientation behaviour.

The feedback from these manual testings, was mainly to verify the common faults presented in the next section, based on Amalfitano et al research [13].

3.2 Definition of Mutation Operators

Orientation change on mobile applications is a peculiar event. Android guideline recommend that the application should adapt itself to the new layout if a orientation change event occurs, preserving state and without losing any data.

Amalfitano et all article [13] aimed at investigating "the failures exposed in mobile applications by mobile-specific event of changing the screen orientation" and focus in "GUI failures
resulting in unexpected GUI states that should be avoid to improve the applications quality and to ensure better user experience. To validate event of changing the screen orientation, two consecutive orientation change events are required and compares the initial state before event with the final state after the two consecutive orientation change. If the initial state and final state are equal, no failure happened.

From their research, six classes of common faults were defined. From these six, because of time constraints, it was decided to apply mutants to three of the common faults defined:

- **Show method called on Dialog or its Builder**

  Show method called on Dialog or its Builder refers to the Dialog behavior when the orientation changes, so the Dialog is going to disappear when the activity is destroyed by calling `onDestroy()` method and recreated after orientation change event. The fault happens when it is not possible to adapt to a new layout configuration without loosing data when `onCreate()` method is called.

```
private void dialog()
{
    DialogDialogBuilder builder = new DialogDialogBuilder(this);
    // The Builder class is used for convenient dialog construction...
    builder.show();
    DialogFragment backgplert = new DialogDialogFragment();
    backgplert.show(getActivityFragmentManager(), "backg");
}

public class DialogDialogFragment extends DialogFragment
{
    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState)
    {
        AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
        // The Builder class is used for convenient dialog construction...
        builder.create();}
```

Figure 3.3: Example of a show method called on Dialog or its Builder Fault and a fix in an app. [13]

- **Missing Id in XML layout**

  Missing Id in XML layout refers to an unique ID to restore the state of the views, so, if it is not unique, it can cause the off state when the user rotates the mobile.

  The failure produced is the wrong `scrollView` behavior and for introducing the failure, we can introduce a fault by removing the `@id/scrollview`.

- **Aged target SDK version**

  Aged target SDK version refers to `android:targetSdkVersion` version that lower than 19 is responsible for the loss of `ScrollView` position after rotation. The failure produced is lost of position for the `scrollview` and for introducing the failure, we can introduce a fault by change the target SDK version.

  The mutant operators described above and presented as the three classes of common faults by Amalfitano et all are the ones used and defined for this research.
At last, from the following article [31], it was defined and described as "**Mutation Operator 2**: This mutation operator is applied when developers persist data in local storage using shared preferences", where the mutation operator clears and commits an empty editor in the `onPause()` method, so when the application reads shared preference they are empty.

This mutant operator was already defined but it was not automate the insertion of the mutant into the code, so, this research is going to improve that.
3.3 Mutation Operators automation

João Gouveia [17], for his thesis, developed a command line tool using Node JS to automatically inject the mutation operators. The command line tool developed by [17] was extended to injected the mutation operators defined in the above subsection, by adding new mutation operators but also by searching in both JAVA and XML files to inject the mutation operators.

For replacing the original code by a mutant operator, a regex was defined. So for each mutant operator, a regex is used to search for the piece of code that should be replaced by the mutant operator.

The tools works as a two staged process:

- Exploration
  The tool getting the path to the mobile application folder looks in the directory where you inject the mutants.

- Injection
  After finding where to inject the mutation operator, injects the mutant and generate a new folder.

The flow for the two staged process is described below:

1. After running the tool by using the command `node index.js`, the directory to the mobile application is prompted.

2. Insert the path for the directory and hit enter.

---

```java
...  
SharedPreferences v0 =  
    getSharedPreferences(file_name, mode);  
...

(Mutant 2)

public void onPause() {
    super.onPause();
    ...
    getSharedPreferences(file_name,
        mode).edit().clear().commit();
}
```

Figure 3.7: Applications distribution according Google Play Stores categories.
3. The tool searches the directory for any matching files (XML and JAVA) to inject the mutation operators.

- If the directory has no files to be injected, the tool finishes its process and notifies the user;

![Figure 3.8: Mutation Tool.](image)

Figure 3.8: Mutation Tool.

- Otherwise, it proceed for the injection stage.

4. For every file where the mutant can be injected:

- The tool generates a new folder with the original content plus the mutant file injected in the output project folder.

- The tool prints in the command line the path to the file where was possible to inject the mutant, the type of mutant injected and the number of mutants for each type

![Figure 3.9: Tool’s output when is not possible to inject mutant of certain type.](image)

Figure 3.9: Tool’s output when is not possible to inject mutant of certain type.

![Figure 3.10: Tool’s output when is possible to inject mutant of certain type.](image)

Figure 3.10: Tool’s output when is possible to inject mutant of certain type.

The output generated by the tool can be manually validated, verifying if the mutant was killed and generating metrics. For that, it is necessary to generate the mutant APK by replacing the original directory given with the mutant folder.

Each mutant needs its own implementation and how the tool interacts with the files also depends:

- **Show method called on Dialog or its Builder**

  The tool searches for instances of `AlertDialog` via a regex expression by replacing the parameter passed to the function. When it finds one, it creates a new folder with the
3.3 Mutation Operators automation

original content directory and a new folder with the mutant code. If no mutant is injected, it outputs that no mutant AlertDialog was injected as shown in 3.11

Figure 3.11: Tool’s output example.

Figure 3.12: Regex and Mutant Operator code for Show method called on Dialog or its Builder.

- **Missing Id in XML layout**

Amalfitano et al article [13] used as example for Missing Id in XML layout the component scrollview. For automation purpose, the implementation was to removed for the scrollview where the id is defined in the XML android:id="@id/scrollView1", seeking to prevent Android system to restore the state of the views contained in the activity.

The tools searches for the tag <ScrollView and replaces the android:id="@+id/" for an empty line.

Figure 3.13: Regex and Mutant Operator code for Missing Id in XML layout.
Definition of Android Mutation Operators

The tool outputs the path to the file where the mutant was injected and if it does not find one file to inject the mutant, no mutants is injected as shown in 3.11.

- **Aged target SDK version**

  The tool does a search for an `AndroidManifest.xml` file where the following tag exists: `uses-sdk`. When the tag is found, the tag is replaced by a `targetSdkVersion` lower than the 19 or an `overrideLibrary` lower than the version 19.

  Since both situations are common, `tools:overrideLibrary` and `android:targetSdkVersion`, a new regex was created for finding both tags and generating a file containing the mutant operator.

  ```cpp
  const BACKGROUND_SDK_REXEG = `/uses-sdk.+tools:overrideLibrary\="[\\w\\s]*\"/\`;  
  const BACKGROUND_SDKVERSION_REXEG = `/uses-sdk\[\\s]*android:targetSdkVersion\="[\\w\\s]*\"/\`;  

  const BACKGROUND_SDK_MUTANT = "android.support.v17.leanback";  
  const BACKGROUND_SDKVERSION_MUTANT = "17";  
  ```

  Figure 3.14: Regex, constant replaced and Mutant Operator code for Aged Target SDK Version.

  The tool outputs the path to the file where the mutant was injected and if it does not find one file to inject the mutant, no mutants is injected as shown in 3.11.

- **Mutant Operator 2 - onPauseSharedPreferences**

  The tool searches for occurrences of `onPause` method. When it finds one, override the method and creates a new folder with the content original and the injected mutant in the new folder. The tool outputs the path to the file where the mutant was injected.

  ```cpp
  const BACKGROUND_ONPAUSE_REXEG = `/\.*onPause\..+\s\$\[\s*\]/\`;  
  const BACKGROUND_ONPAUSESharedPreferences_REXEG = `/\.*onPauseSharedPreferences\..+\s\$\[\s*\]/\`;  
  const BACKGROUND_ONPAUSESharedPreferences_MUTANT = "super.onPause(); SharedPreferences.Editor put mSharedPreferences.FILE_NAME, MBC_MANUFACT.EDITOR put mSharedPreferences.FILE_NAME, MBC_MANUFACT.next(); super.onResume();";  

  ```

  Figure 3.15: Regex Mutant Operator 2.

  If no mutant is injected, it outputs that no mutant `onPauseSharedPreferences` was injected as shown in 3.11.

  The tool is very straightforward to use and extend. More mutation operators can be added and some of the existing ones can be readjusted with some minor modifications and this is very important.
3.3 Mutation Operators automation

3.3.1 Liabilities

Aged target SDK version and Missing Id in XML layout require manipulation of XML files and the tool uses regex for that. A regex can search for a specific pattern but, as XML is not a regular language, with structures that can nest arbitrarily deep, written in different ways by developers, so it is almost impossible to ensure that all standards are respected and cover all the scenarios in which developers write their XML.

By manual validation in a large number of applications, was tried to find the best possible pattern and multiple Regexes to cover different possibilities.

There is also another issue from using Regex with XML files: Catastrophic Backtracking. Catastrophic Backtracking is a condition that can occur when we are checking a (usually long) string against a complex regular expression. "The problem is that there’s two ways the string could match at every stage: using "[a-zA-Z0-9]+", or using "[a-zA-Z0-9]+s". For a string of length 30 this would mean approx. 1 billion — possibilities!" [10].

The solution to the problem, that might prevent blocking the thread when we are looking for the code that needs to be replaced, is to use a regex that requires the minimum of iterations, aiming to improve the regex to its maximum potential with the minimum of iterations, like leveraging non-capturing groups instead of redundant capturing ones.

Taking into account these 2 points presented, both mutants were correctly implemented and it was possible to find the location to inject them. It should be noted that if there is more than one, it will only be able to inject the mutant in the first one found, and this is a rule for all four mutants.

There are some other limitations of the tool. The tool does not have a log implemented, where the result can be written after the command is closed, being necessary to copy the information of the command line to a file to save later.

When the tool is run, all mutants are injected and it is not possible to define which mutants we specifically intend to inject. This improvement may direct the tool to better manage the needs of developers.

Finally, as known, NodeJS has limitations in making scalable applications and the adoption of the asynchronous programming model could be a solution but is harder to implement in comparison to the linear blocking I/O programming. The non adoption of adoption of the asynchronous programming model makes the operations slower since it prevents from injecting multiple mutants at the same time.

Despite the limitations presented, it seems that the advantages of this automation go far beyond its limitations and give the necessary guarantee of the injection of mutants in a simple, efficient, effortless and relatively fast way in the Android applications.
3.4 Conclusion

In this chapter, 40 different Android applications were studied, analyzed and tested the correctness implementation of guidelines to the event of changing the screen orientation. Six of them, showed, by manual tests, signs of the violation of guidelines.

Four mutants operators were defined in accordance with the Amalfitano et. all article [13] and its automation delineated and implemented efficiently. The mutant operators were generated automatically with success.

In the next chapter, after compiling all the generated files, manual tests will be performed in order to infer if the mutants may be killed and calculate the Mutation Score per mutation operator.

The results obtained will allow to verify the quality of the work developed throughout this project and obtain conclusions.
Chapter 4

Case Study

This chapter presents the results of the research. For each application, mutant operators were generated using the automation tool described in section 3.3. The generated mutants were manually ran to attest its reliability and accuracy.

4.1 Automated Mutation Injection

All applications were ran through the automated mutation injection tool described in section 3.3. The results are presented in table 4.1 and all the results per application can be over viewed in B.1.

<table>
<thead>
<tr>
<th>Mutation Operators</th>
<th>Mutants Generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Show method called on Dialog or its Builder</td>
<td>141</td>
</tr>
<tr>
<td>Aged target SDK version</td>
<td>6</td>
</tr>
<tr>
<td>Missing Id in XML layout</td>
<td>70</td>
</tr>
<tr>
<td>Mutant Operator 2 - onPauseSharedPreferences</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 4.1: Number of Mutants Generated for each Mutation Operator.

Show method called on Dialog or its Builder and Missing Id in XML layout generate higher number of mutants for the 40 application in the dataset than the Mutant Operator 2 - onPauseSharedPreferences and Aged target SDK version. There are some explanations that cause these differences:

- **Show method called on Dialog or its Builder**
Although it was possible to inject mutants into the `Dialog` method, it was not possible to inject one to validate the behavior when the application is rotated. However, it was possible to inject a mutation to be validated.

- **Aged target SDK version**
  
  This mutant is inserted in the `AndroidManifest.xml` file that it is unique in each project with a single reference to `uses-sdk` tag. But in spite of being only one file, in addition, not all developers declares the tag.

  This happens because, since the applications use Gradle, then most of the developers prefers to update the `build.gradle` file rather than the manifest, since Gradle overrides the manifest values. Although not advised by the good practices of Android.

  Gradle overrides the manifest values, and I prefer to update the `build.gradle` file rather than the manifest.

- **Missing Id in XML layout**
  
  This mutant has been specifically injected into the `ScrollView` but can easily be adapted to other components.

  It was injected into all `ScrollView` tag by removing the `Android:id` line.

- **Mutant Operator 2 - `onPauseSharedPreferences`**
  
  The Mutant Operator 2 - `onPauseSharedPreferences` is applicable to all activities that are overriding the `onPause` method, and developers uses `SharedPreferences` to save state and properties before the application calls the `onPause` method, for example when it goes to background.

  The `onPause` method exists in all applications but only a very small number of the used sample of applications invokes the interface `SharedPreferences` for accessing and modifying preference data.

  In addition, in three of the applications used it was not possible to confirm the injection of the mutant.

Despite the very reduced results of some mutant operators obtained, they are important according to the articles that served as the basis for the definition of mutant operators. They will not be disregarded in the manual analysis to be performed and described in the following section.

In the total sample of applications used, only in uCrop, Photo Affix, Material Dialogs Library Demo, Lottie the mutants could not be injected. Blokish does not rotate and Timber only rotates in one of its screens. Both applications, ended up not contributing to the research.
4.2 Manual Validation of the Automated Mutation Operators

<table>
<thead>
<tr>
<th>Mutation Operators</th>
<th>Number of Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Show method called on Dialog or its Builder</td>
<td>31</td>
</tr>
<tr>
<td>Aged target SDK version</td>
<td>6</td>
</tr>
<tr>
<td>Missing Id in XML layout</td>
<td>23</td>
</tr>
<tr>
<td>Mutant Operator 2 - onPauseSharedPreferences</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 4.2: Number of Applications where was possible to inject each type of mutation operator.

In table 4.2 is shown the number of applications where it was possible to inject for each type of mutant. The Show method called on Dialog or its Builder was injected in a higher number of applications. This situation had already been explained above. The sample appears to be significant enough to be used as well as the mutation operators defined.

It remains to evaluate whether the generated mutants are manually detected. This will be the challenge described in the next section and evaluate the results by calculating the Mutation Score.

4.2 Manual Validation of the Automated Mutation Operators

Through the mutant injection tool, we obtain a folder with the file of the mutant injected. This file should then be replaced in the original folder and the application be compiled with that source code change.

With the compiled application with an injected mutant, manual tests were run to analyze whether the behavior entered by the tool (mutant app) differs from the behavior of the original application.

According to the possible behavior to be obtained in manual tests, three categories of faults injected were defined:

- **Incorrect** - Not possible to compile and run the mutant application.
- **Detected** - The failure was detected by testing manually (test case capable of detecting different behavior between mutant application and original).
- **Not Detected** - The failure was not detected by testing manually (test case not capable of detecting different behavior between mutant application and original).

For the execution of manual tests was constructed a test case to be executed for both injected mutant operator application and the original.

For Show method called on Dialog or its Builder, Aged target SDK version and Missing Id in XML layout, the exact same steps were performed on the original and on the mutant.
Android application and then the application was rotated from portrait to landscape and back to portrait. The result of both tests was compared and defined the category of the faults injected. For Mutant Operator 2 - onPauseSharedPreferences, the same steps were performed on the original and on the mutant Android application making the application goes to background and back to foreground. Once again, the result of both tests was compared and defined the category of the faults injected.

The results for the manual testing are presented in 4.3 and all the results per application can be overviewed in C.1. Due to time constraints, only 40% of the mutant apps were tested against the original.

<table>
<thead>
<tr>
<th></th>
<th>Show method called on Dialog or its Builder</th>
<th>Aged target SDK version</th>
<th>Missing Id in XML layout</th>
<th>Mutant Operator 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detected</td>
<td>8</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Not Detected</td>
<td>17</td>
<td>2</td>
<td>22</td>
<td>1</td>
</tr>
<tr>
<td>Incorrect</td>
<td>1</td>
<td>2</td>
<td>21</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>26</td>
<td>5</td>
<td>45</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 4.3: Mutated Application Manual Testing Results

Regarding the Missing Id in XML layout, some applications cannot compile without the missing element.

Given the results from 4.3 it is possible to overall efficiency in detecting incorrect behaviour by calculating Mutation Score in accordance with:

\[
\text{Mutation Score (MS)} = \frac{\text{Mutants Killed}}{\text{Total Mutants} - \text{Equivalent Mutants}}
\]

where the Mutants Killed are the amount of the detected for each mutant operator, Total mutants are the sum of the Detected, Not Detected and Incorrect for each mutant operator and Equivalent Mutants are the amount of Not Detected for each mutant operator.

\[
\text{Show method called on Dialog or its Builder MS}= \frac{8}{(26 - 17)} = 0.89
\]

\[
\text{Aged target SDK version MS}= \frac{1}{(5 - 2)} = 0.33
\]

\[
\text{Missing Id in XML layout MS}= \frac{2}{(45 - 22)} = 0.09
\]

\[
\text{Mutant Operator 2 - onPauseSharedPreferences MS}= \frac{1}{(2 - 1)} = 1
\]

\[
\text{All MS}= \frac{12}{(78 - 42)} = 0.33
\]
4.3 Conclusion

The Mutation Score results are the result of the dataset used and the manual validation. Since many applications do not compile with missing id element in xml, the results obtained for Missing Id in XML layout mutation score are the reflection of this behavior.

4.3 Conclusion

During this chapter, mutation operators defined in section 3 were injected using the automation tool extend and the results were ran against the original applications.

The automation of the mutation operators occurred efficiently and allowed the creation of several code changes (mutants). Show method called on Dialog or its Builder and Missing Id in XML layout generated most of the mutants since they are widgets widely used in Android applications. Mutant Operator 2 - onPauseSharedPreferences depends on developer use share preferences to ensure the preference values remain in a consistent state and control when they are committed to storage. Aged target SDK version exists only on AndroidManifest.xml, and, although guidelines recommend specification, developers do not use it.

Tests were conducted manually which took some time and the results of the manual tests were not conclusive enough.
Chapter 5

Conclusions and Future Work

In this project, four different mutant operators were defined, three of them affecting the rotation of Android Applications and one affecting the background-foreground behaviour of Android Application. These mutants were defined based on the Android programming guidelines. The programmer does not always follow the best practices in programming and the result is the reduction of the quality of its Applications and other consequences that follow. By injecting this faults, bad practices can be prevented.

After selecting a set of Android Applications, it was manually investigated wrong behaviors resulting from poor Application of the guidelines to the event of changing the screen orientation, based on research developed Amalfitano et. all article [13] where some Android-specific fault classes were defined.

These mutation operators were injected into various Android Applications through an automation tool. This automation tool was developed in Node.js. The use of certain regex pattern is searched in code and replaced by the new one. This substitution in the code creates a new file that, when being replaced in the original folder, allows you to run against the original and compared and calculate the mutation score based on the results.

Despite of the good performance of automation tool, potential improvements were also discuss and presented for future.

All mutation operators generated a significant amount of mutant application. Two of the mutation operators did not generate a significant amount of mutants, not because the mutant Application was incorrect, but because the dataset used did not contain as many cases as would be desired, for example, more Applications in which it would be possible to rotate screen.

The mutant applications generated and original applications were manually run and the behaviors obtained for were compared for calculation of the score. A test case was defined and manually tested on the mutant applications generated and original application using the simulator from Android Studio.
Conclusions and Future Work

Given the results obtained, it was possible to analyse the efficiency of the mutant operators defined to inject faults simulating real failures in the event of changing the screen orientation and incorrect background-foreground behaviour by calculating the Mutation Score and evaluate the work developed. For some cases it was not possible to detect loss of information or a different behaviour between the two.

5.1 Goal Satisfaction

The main goal of this research was to automate a set of mutation operators that affects rotation of Android Applications defined in Amalfitano et al. article [13]. For defining the mutation operators, the Android programming guidelines were taken into account.

The automation of the mutation operator was achieved by extending an existing automation tool in Node JS which injected automatically the mutant operators on code.

Manually it was possible to verify the quality of the mutation operators defined by injecting them in real applications. Comparing the results produced manually regarding the original code and the mutated code, it was possible to verify if the test case executed manually was efficient to detect the injected faults and kill them.

5.2 Future Work

As future work it would be interesting to test the mutation operators defined with an GUI tool and compare with the results obtained manually.

Since two of the defined operators had little results, it would be interesting to increase the sample in order to validate these same mutants.

It would also be important to improve the Node JS tool to manage asynchronous behaviour and evaluate how the tool behaves.

And, at last, it should be interesting to investigate some faults in Android Applications developed in Kotlin, as it is gaining ground as a development language for Android Applications since Google started supporting it [11].
References


[37] Yuan Wei. Mudroid: Mutation testing for android apps.
Appendix A

Android applications dataset
### Table A.1: Android applications dataset.

<table>
<thead>
<tr>
<th>Name</th>
<th>Category</th>
<th>Repository</th>
<th>Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amaze File Manager</td>
<td>Tools</td>
<td><a href="https://github.com/TeamAmaze/AmazeFileManager">https://github.com/TeamAmaze/AmazeFileManager</a></td>
<td>4.3</td>
</tr>
<tr>
<td>aMetro</td>
<td>Maps and Navigation</td>
<td><a href="https://github.com/RomanKolovanov/ametro">https://github.com/RomanKolovanov/ametro</a></td>
<td>4.4</td>
</tr>
<tr>
<td>And Bible</td>
<td>Books and Reference</td>
<td><a href="https://github.com/mjdenham/and-bible">https://github.com/mjdenham/and-bible</a></td>
<td>4.6</td>
</tr>
<tr>
<td>AnkiDroid Flashcards</td>
<td>Education</td>
<td><a href="https://github.com/ankidroid/Anki-Droid">https://github.com/ankidroid/Anki-Droid</a></td>
<td>4.5</td>
</tr>
<tr>
<td>AntennaPod</td>
<td>Video Players &amp; Editors</td>
<td><a href="https://github.com/AntennaPod/AntennaPod">https://github.com/AntennaPod/AntennaPod</a></td>
<td>4.6</td>
</tr>
<tr>
<td>Anuto TD</td>
<td>Strategy</td>
<td><a href="https://github.com/reloZid/android-anuto">https://github.com/reloZid/android-anuto</a></td>
<td>4.3</td>
</tr>
<tr>
<td>Bliskish</td>
<td>Puzzle</td>
<td><a href="https://github.com/scoutant/bliskish">https://github.com/scoutant/bliskish</a></td>
<td>4.3</td>
</tr>
<tr>
<td>Chroma Doze</td>
<td>Music and Audio</td>
<td><a href="https://github.com/pmarks-net/chromadoze">https://github.com/pmarks-net/chromadoze</a></td>
<td>4.5</td>
</tr>
<tr>
<td>ConnectBot</td>
<td>Communication</td>
<td><a href="https://github.com/connectbot/connectbot">https://github.com/connectbot/connectbot</a></td>
<td>4.6</td>
</tr>
<tr>
<td>Debatekeeper</td>
<td>Tools</td>
<td><a href="https://github.com/czlee/debatekeeper">https://github.com/czlee/debatekeeper</a></td>
<td>4.6</td>
</tr>
<tr>
<td>Forkhub for Github</td>
<td>Productivity</td>
<td><a href="https://github.com/jonan/ForkHub">https://github.com/jonan/ForkHub</a></td>
<td>4.4</td>
</tr>
<tr>
<td>GPS Logger for Android</td>
<td>Travel and Local</td>
<td><a href="https://github.com/BasicArtData/GPSLogger">https://github.com/BasicArtData/GPSLogger</a></td>
<td>4.2</td>
</tr>
<tr>
<td>k-9 mail</td>
<td>Communication</td>
<td><a href="https://github.com/k9mail/k-9">https://github.com/k9mail/k-9</a></td>
<td>4.2</td>
</tr>
<tr>
<td>Libre Torrent</td>
<td>Video Players &amp; Editors</td>
<td><a href="https://github.com/promnyaaroslav/libretorrent">https://github.com/promnyaaroslav/libretorrent</a></td>
<td>4.3</td>
</tr>
<tr>
<td>Lottie</td>
<td>Library and Demo</td>
<td><a href="https://github.com/arthur/lottie-android">https://github.com/arthur/lottie-android</a></td>
<td>4.7</td>
</tr>
<tr>
<td>Material Dialogs Library Demo</td>
<td>Library and Demo</td>
<td><a href="https://github.com/afollestad/material-dialogs">https://github.com/afollestad/material-dialogs</a></td>
<td>4.8</td>
</tr>
<tr>
<td>MHGen Database</td>
<td>Books and Reference</td>
<td><a href="https://github.com/gatheringhallstudios/MHGenDatabase">https://github.com/gatheringhallstudios/MHGenDatabase</a></td>
<td>4.8</td>
</tr>
<tr>
<td>My Diary (unofficial)</td>
<td>Lifestyle</td>
<td><a href="https://github.com/DaxiaK/MyDiary">https://github.com/DaxiaK/MyDiary</a></td>
<td>4.8</td>
</tr>
<tr>
<td>Omni Notes</td>
<td>Productivity</td>
<td><a href="https://github.com/federicoitsou/Omni-Notes">https://github.com/federicoitsou/Omni-Notes</a></td>
<td>4.4</td>
</tr>
<tr>
<td>OpenBikeSharing</td>
<td>Travel and Local</td>
<td><a href="https://github.com/bparmentier/OpenBikeSharing">https://github.com/bparmentier/OpenBikeSharing</a></td>
<td>4.2</td>
</tr>
<tr>
<td>OpenTasks</td>
<td>Productivity</td>
<td><a href="https://github.com/dmix/opentasks">https://github.com/dmix/opentasks</a></td>
<td>4.2</td>
</tr>
<tr>
<td>Photo Allix</td>
<td>Tools</td>
<td><a href="https://github.com/afollestad/photo-allix">https://github.com/afollestad/photo-allix</a></td>
<td>4.5</td>
</tr>
<tr>
<td>pMetro for Android</td>
<td>Travel and Local</td>
<td><a href="https://github.com/uliyf/pMetro">https://github.com/uliyf/pMetro</a></td>
<td>4.1</td>
</tr>
<tr>
<td>qBittorrent Controller</td>
<td>Tools</td>
<td><a href="https://github.com/Galliard/qBittorrent-Controller">https://github.com/Galliard/qBittorrent-Controller</a></td>
<td>3.8</td>
</tr>
<tr>
<td>RedReader</td>
<td>News and Magazines</td>
<td><a href="https://github.com/QuantumBudget/RedReader">https://github.com/QuantumBudget/RedReader</a></td>
<td>4.6</td>
</tr>
<tr>
<td>Rgb Tool</td>
<td>Tools</td>
<td><a href="https://github.com/fasteque/rgb-tool">https://github.com/fasteque/rgb-tool</a></td>
<td>4.3</td>
</tr>
<tr>
<td>Shader Editor</td>
<td>Tools</td>
<td><a href="https://github.com/markusvlak/ShaderEditor">https://github.com/markusvlak/ShaderEditor</a></td>
<td>4.7</td>
</tr>
<tr>
<td>Swiftnotes</td>
<td>Productivity</td>
<td><a href="https://github.com/adrianchifor/Swiftnotes">https://github.com/adrianchifor/Swiftnotes</a></td>
<td>4.3</td>
</tr>
<tr>
<td>Timber</td>
<td>Music and Audio</td>
<td><a href="https://github.com/naman14/Timber">https://github.com/naman14/Timber</a></td>
<td>4.3</td>
</tr>
<tr>
<td>uCrop</td>
<td>Photography</td>
<td><a href="https://github.com/Yalantis/uCrop">https://github.com/Yalantis/uCrop</a></td>
<td>4.0</td>
</tr>
<tr>
<td>Unit Converter Ultimate</td>
<td>Tools</td>
<td><a href="https://github.com/physphil/UnitConverterUltimate">https://github.com/physphil/UnitConverterUltimate</a></td>
<td>4.5</td>
</tr>
<tr>
<td>Vilex Checker</td>
<td>Maps and Navigation</td>
<td><a href="https://github.com/ojacque/mart/vilex-Checker">https://github.com/ojacque/mart/vilex-Checker</a></td>
<td>4.6</td>
</tr>
<tr>
<td>Web Opac: 1,000+ libraries</td>
<td>Education</td>
<td><a href="https://github.com/opacapp/opaccient">https://github.com/opacapp/opaccient</a></td>
<td>4.3</td>
</tr>
<tr>
<td>Weechat Android</td>
<td>Communication</td>
<td><a href="https://github.com/uberjek/2/weechat-android/">https://github.com/uberjek/2/weechat-android/</a></td>
<td>4.3</td>
</tr>
<tr>
<td>WiFi Automatic</td>
<td>Tools</td>
<td><a href="https://github/j4velin/WiFi-Automatic">https://github/j4velin/WiFi-Automatic</a></td>
<td>4.1</td>
</tr>
<tr>
<td>WiFi Analyser (open-source)</td>
<td>Tools</td>
<td><a href="https://github.com/VREMSoftwareDevelopment/WiFiAnalyzer">https://github.com/VREMSoftwareDevelopment/WiFiAnalyzer</a></td>
<td>4.4</td>
</tr>
<tr>
<td>WiGLE WiFi Wardriving</td>
<td>Tools</td>
<td><a href="https://github.com/wiglenet/wigle-wifwardriving/">https://github.com/wiglenet/wigle-wifwardriving/</a></td>
<td>4.2</td>
</tr>
</tbody>
</table>
Appendix B

Number of mutants generated per application
Table B.1: Number of mutant applications generated by the tool developed per mutation operator and application.
Appendix C

Manual testing result
<table>
<thead>
<tr>
<th>Name</th>
<th>Show method called on Dialog or its Builder</th>
<th>Aged target SDK version</th>
<th>Missing Id in XML layout</th>
<th>Mutant 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amaze File Manager</td>
<td>1 Detected</td>
<td>-</td>
<td>1 Detected</td>
<td>-</td>
</tr>
<tr>
<td>abMetro</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Amulet</td>
<td>Not tested</td>
<td>-</td>
<td>Not tested</td>
<td>-</td>
</tr>
<tr>
<td>AnkiDroid Flashcards</td>
<td>Rotation not possible</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Anato TD</td>
<td>Rotation not possible</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>BlokTalk</td>
<td>Rotation not possible</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Chromia Zoo</td>
<td></td>
<td>-</td>
<td>Not Detected</td>
<td>1 Detected</td>
</tr>
<tr>
<td>Clementine Remote</td>
<td>Not tested</td>
<td>-</td>
<td>Not tested</td>
<td>-</td>
</tr>
<tr>
<td>ConnectHot</td>
<td>Not Detected</td>
<td>-</td>
<td>5 Incorrect</td>
<td>-</td>
</tr>
<tr>
<td>Debanskeeper</td>
<td>Not tested</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Forkhub for Github</td>
<td>Not tested</td>
<td>-</td>
<td>5 Incorrect</td>
<td>-</td>
</tr>
<tr>
<td>GPS Logger for Android</td>
<td>2 Detected/3 Not Detected</td>
<td>-</td>
<td>Not Detected</td>
<td>-</td>
</tr>
<tr>
<td>i-O mail</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Lottie</td>
<td>Incorrect</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Material Dialogs Library Demo</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>MHRGen Database</td>
<td>Not tested</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>My Diary (unofficial)</td>
<td>Incorrect</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Omia Notes</td>
<td>Incorrect</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>OneBusAway</td>
<td>Not tested</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>OpenAudioSharing</td>
<td>Not Detected</td>
<td>-</td>
<td>Not Detected</td>
<td>-</td>
</tr>
<tr>
<td>OpenTasks</td>
<td>Not tested</td>
<td>-</td>
<td>Not Detected</td>
<td>-</td>
</tr>
<tr>
<td>Photo Affix</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>pMetro for Android</td>
<td></td>
<td>-</td>
<td>Incorrect</td>
<td>-</td>
</tr>
<tr>
<td>Primitive FTPD</td>
<td>Not tested</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>qBittorrent Controller</td>
<td>Not tested</td>
<td>-</td>
<td>Not Detected</td>
<td>-</td>
</tr>
<tr>
<td>RedReader</td>
<td>Not tested</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>RGit Tool</td>
<td>Not tested</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Shader Editor</td>
<td>Not detected</td>
<td>-</td>
<td>Not Detected</td>
<td>-</td>
</tr>
<tr>
<td>ShutUp!</td>
<td>1 Detected</td>
<td>-</td>
<td>1 Detected</td>
<td>1 Detected</td>
</tr>
<tr>
<td>Swiftnotes</td>
<td>1 Detected</td>
<td>-</td>
<td>Not Detected</td>
<td>-</td>
</tr>
<tr>
<td>Timber</td>
<td>1 Detected</td>
<td>-</td>
<td>Incorrect</td>
<td>Incorrect</td>
</tr>
<tr>
<td>uCrop</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Unit Converter Ultimate</td>
<td>Not tested</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Vlille Checker</td>
<td>1 Detected</td>
<td>Not Detected</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>WebOpac: 1,000+ libraries</td>
<td>Not tested</td>
<td>-</td>
<td>Not tested</td>
<td>-</td>
</tr>
<tr>
<td>WeChat Android</td>
<td>Rotation not possible</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>WiFi Automatic</td>
<td>1 Detected</td>
<td>-</td>
<td>1 Detected</td>
<td>-</td>
</tr>
<tr>
<td>WiFiAnalyser (open-source)</td>
<td>-</td>
<td>-</td>
<td>Incorrect</td>
<td>-</td>
</tr>
<tr>
<td>WorldWide WiFi Wardriving</td>
<td>Not tested</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Total Detected</td>
<td>8</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Total Not Detected</td>
<td>17</td>
<td>2</td>
<td>22</td>
<td>1</td>
</tr>
<tr>
<td>Total Incorrect</td>
<td>1</td>
<td>2</td>
<td>21</td>
<td>0</td>
</tr>
<tr>
<td>Total Run</td>
<td>26</td>
<td>5</td>
<td>55</td>
<td>2</td>
</tr>
</tbody>
</table>

Table C.1: Dataset applications general information and results of Manual Testing.