FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Mutation Operators for Android Apps

Ana Rita Veiga

Master's in Software Engineering

Supervisor: Ana Cristina Ramada Paiva

Second Supervisor: João Carlos Viegas Martins Bispo

June 26, 2023

Mutation Operators for Android Apps

Ana Rita Veiga

Master's in Software Engineering

June 26, 2023

Resumo

Os dispositivos móveis evoluíram exponencialmente na última década, tornando-se uma ferramenta determinante no nosso quotidiano, desempenhando um papel fulcral na comunicação, acesso à informação e entretenimento.

Dentro das aplicações existentes no mercado, as aplicações Android apresentam-se como as mais populares, com números elevados de lançamentos, compras e transferências.

Dado o impacto significativo que os dispositivos móveis têm nas nossas vidas, torna-se de extrema importância garantir a qualidade das aplicações android.

Com este trabalho, pretendemos contribuir para o desenvolvimento de aplicações Android de maior qualidade, através da identificação de estratégias para minimizar os custos associados a *Mutation testing*.

Mutation testing consiste em aplicar pequenas alterações ao código-fonte do sistema, em teste, com o intuito de gerar cópias incorretas do programa, conhecidas como mutantes, e compará-las com a versão original. Todavia, *mutation testing* requer recursos computacionais significativos, o que inviabiliza a sua aplicação projetos de média e larga escala.

A fim de reduzir o custo de *mutation testing* implementámos operadores de mutação capazes de produzir mutantes com a estrutura do *Mutant Schemata*, também chamado de *metamutants*, no qual todos os mutantes são agrupados numa única versão do software. Esta técnica reduz o consumo de recursos: memória e tempo.

Foram ainda propostas duas abordagens para suprimir duas limitações encontradas durante o processo de implementação de operadores de mutação em *metamutants*. Primeiramente, foi proposta uma reestruturação da *Abstract Syntax Tree*, permitindo a aplicação destes operadores em declarações e instanciações. Paralelamente, foi introduzida uma abordagem inovadora aplicada sobre ficheiros *XML*, que expande o leque de operadores aplicáveis num cenário de teste.

Por fim, avaliámos a correção da implementação do *metamutante* em relação à implementação tradicional e comparámos o consumo de recursos (tempo e memória) das duas implementações.

Os resultados revelaram que os mutantes implementados utilizando a estrutura de *Mutant Schemata*, apresentaram vantagens consideráveis em termos de eficiência de tempo e utilização de espaço em disco, superando a abordagem tradicional na geração de mutantes.

Abstract

Mobile devices have evolved exponentially in the last decade, becoming a key tool in our daily lives and playing a central role in communication, access to information and entertainment.

Of all the applications on the market, Android applications (apps) are the most popular, with large numbers of releases, purchases and downloads. Given the significant impact that mobile devices have on our daily lives, ensuring the quality of mobile apps becomes of utmost importance.

With this work, we aim to contribute to the development of higher-quality Android apps by identifying strategies to minimize the expenses associated with mutation testing effectively.

Mutation testing has as a premise to make small changes to the source code of the system under test in order to generate incorrect copies of the program, known as mutants, and compare them with the original version. However, mutation tests require significant computational and financial resources, which makes their application in medium and large-scale projects unfeasible.

In order to reduce the cost of mutation testing, we implemented mutation operators capable of producing mutants with the structure of *Mutant Schemata*, also called *metamutants*, in which all mutants are packed into a single version of the software application. This technique reduces resource consumption: memory (just one version of the System Under Test – SUT – instead of one version for each mutant) and time (reducing the deployment time because we need to deploy just once).

Two approaches were also proposed to remove two limitations encountered during the process of implementing mutation operators in *metamutants*. Firstly, a restructuring of the *Abstract Syntax Tree* was proposed allowing the application of these operators in declarations and instantiations. In parallel, an innovative approach was introduced on *XML* files, expanding the range of applicable operators in a test scenario.

Finally, we evaluated the correctness of the *metamutant* implementation in relation to the traditional implementation. We also compared the resource consumption (time and memory) of the two implementations.

The results revealed that mutants implemented using the structure of "Mutant Schemata" exhibited considerable advantages in terms of both time efficiency and disk space utilization, surpassing the traditional approach for generating mutants.

Acknowledgements

Completing this thesis would not have been possible without the collaboration and support of numerous individuals, and I am thankful for their support. First and foremost, I would like to express my sincere appreciation to Prof. Ana Paiva, my supervisor, and Prof. João Bispo, my co-supervisor, for their invaluable guidance throughout the entire process of this dissertation.

I would also like to give special recognition to my twin sister, Sofia Veiga and my family, and also my friend Bruno Rodrigues, who have given me plenty of support. Their unconditional love, dedication, and efforts have provided me with the necessary tools to succeed in the future.

Ana Rita Veiga

“The best way to predict your future
is to create it.”

Abraham Lincoln

Contents

List of Figures

List of Tables

List of Acronyms

AOP	Aspect Oriented programming
API	Application Programming Interface
APK	Android Application Package
APP	Application
AST	Abstract Syntax Tree
CLI	Command Line Interface
DSL	Domain-specific Language
GUI	Graphical User Interface
IPC	Interprocess communication
MSG	Mutant Schema Generation Method
SDK	Software Development Kit
SUT	System Under Test
UMS	Untch Mutant Schema
Wi-Fi	Wireless Fidelity
XML	eXtensible Markup Language

Chapter 1

Introdution

1.1 Context

Over the last decade, the use of Android devices has grown substantially. Mobile devices, such as smartphones and tablets, have become an integral part of people's daily life and are now crucial tools for communication, access to information, and entertainment worldwide. Recently Statista reported that, in 2022, the total number of global mobile devices reached 15.96 billion, exceeding the world population (7.87 billion) in the same year [11].

According to International Data Corporation (IDC), Android devices dominate the global market due to the diversity of brands and gadgets devices and a wider range of prices compared to other products [4]. The growing popularity of these devices is primarily due to a robust ecosystem of applications known as mobile applications (or mobile apps), which are software programs specifically developed for mobile devices [12]. More than a million apps are available for Android users on the Google Play store [13], the most widely used Android app store, with thousands being added every day.

These apps involve several new programming features, including change-prone APIs and platform fragmentation, with little to no knowledge of how to test them. There are no standard adequate automated testing tools due to the fact that only recently, in the last five years [10], did researchers start showing interest in mobile testing [10]. This results in weak and ineffective testing.

Given the enormous impact that mobile devices have in our daily life, it is of utmost importance to ensure the quality of mobile apps in all their facets (e.g., functionality, usability, security), and one way to increase the quality of the software is through testing.

However, the effectiveness of software testing depends on the test suite's quality [14]. One way to assess the test suite's quality is through mutation testing.

1.2 Objectives

The primary objective is to contribute to the development of higher-quality Android apps by identifying strategies to minimize the expenses associated with mutation testing effectively. Reducing the costs will enable a thorough evaluation of Android apps, ensuring they meet the highest quality and performance standards.

Android apps have specific characteristics that existing generic mutation operators do not address. Consequently, certain app functionalities are usually overlooked by this technique. Therefore, there is a need for specific mutation operators to cover these unique characteristics and ensure comprehensive testing.

1.3 Thesis Structure

The structure of the thesis is organized by chapters and their respective sections. Chapter 1 introduces the context of the work, presenting the motivation that led us to the development of this thesis and the objectives expected to be accomplished. The chapter 2 contains the Background of the problem, briefly outlining the mutation analysis process and describing the Android operating system. Chapter 3 refers to the State of the Art, focusing on how mutation testing is applied to Android applications and the existent cost reduction techniques, specifically Mutation testing using Mutant Schemata in Android. Additionally, in this chapter, it is also presented the differences between Mutation testing and Mutation testing using Mutant Schemata. Furthermore, it contains the description of Android mutation operators and existent Android Specific Mutation Tools. Chapter 4 encompasses the presentation of the tool, Kadabra Tool, and the architecture used to implement the Operators, the Mutant generation process, and the Mutant execution process utilized. In Chapter 5, all the mutation operators implemented are briefly described, along with a demonstration of the mutation to be applied. The results of the mutation insertions into each selected application are disclosed in Chapter 6. Finally, Chapter 7 presents the conclusion of this thesis, where we discuss the study's findings and our view on the necessity of further investigation and future work plans.

Chapter 2

Background

This research work falls under the purview of the field of software engineering, a branch of computer science associated with the development and maintenance of software.

In software engineering, a software process "is the model chosen for managing the creation of software from initial customer inception to the release of the finished product" [15].

Any software development process must comprise the four activities listed below. The initial activity is *Software Specification (or requirements engineering)*, which defines all the primary functionality and limitations. The following activity is *Software design and implementation*, which involves designing and programming software. Then we have our focus activity, **Software verification and validation**, which checks if the system meets requirements and standards and serves the intended purpose, and the last activity is *Software evolution (or software maintenance)* where the program is altered to match changes in customer and market needs.

Software testing is a crucial step within the software development life cycle, as it helps ensure its quality since it is a set of activities used to validate and verify that a software system is developed according to its requirements. "Software testing is the process of evaluating and verifying that a software product or application does what it is supposed to do. The benefits of testing include preventing bugs, reducing development costs, and improving performance" [16].

However, the traditional software testing process is insufficient, as certain unknown flaws can still be present due to the failure of test cases to find all the underlying faults [17]. Having that in mind, one of the most effective and proven approaches to achieving a fault-free system is to induce, test, and remove the faults from the system, commonly known as Mutation Testing [18].

This research applies an existing technique, mutation testing, to Android-based applications. This chapter presents the background on mutation testing and Android apps.

2.1 Mutation Testing

Mutation testing (or mutation analysis or program mutation) is a fault injection white-box testing technique that has been empirically found to be exceptionally effective at assessing the quality of the test cases, a set of input values, execution preconditions, expected outputs, and execution post-conditions created for a specific purpose or test condition, such as exercising a certain program path or verifying compliance with a given requirement, and also for enhancing inadequate tests [19].

Through mutation testing, minor modifications are inserted purposely into the program under test to verify whether the existing test cases can detect the errors. To better understand mutation testing, we will give some historical context.

2.1.1 Historical Background

Mutation testing is based on two fundamental principles. The first principle is commonly known as the competent programmer hypothesis (CPH), and the second is known as the coupling effect [20].

The **Competent Programmer Hypothesis (CPH)** was first introduced by DeMillo in 1978, which claims, "Programmers have one great advantage that is seldom exploited: they create programs that are *close* to being correct!" [21].

As programs written by skilled programmers are nearly correct, the possibility of errors is low, and the ones that exist are minor faults and may be fixed by making small syntactical adjustments.

Hence, in Mutation Testing, if tests cannot differentiate between the original code and mutants that mimic faults made by competent programmers, they will also be unable to distinguish between correct and defective code either [5].

The **Coupling Effect** was also proposed by DeMillo *et al* [21], in 1978. According to Offutt, "complex faults are coupled to simple faults in such a way that a test data set that detects all simple faults in a program will detect a high percentage of the complex faults" [22]. Based on this hypothesis, *higher-order mutants* are mutants that have more than one modification. These mutations are likely to be found by test scenarios used to kill simple mutants [23]. As a result, only simple mutants are allowed to be used in traditional mutation testing [5].

2.1.2 Mutation Analysis

Mutation analysis is “the use of well-defined rules defined on syntactic descriptions to make systematic changes to the syntax or to objects developed from the syntax.” [24]

During the process of mutation testing, new versions, known as *mutants*, of a software artifact, such as a program, requirement specification, or configuration file (e.g. specifications, databases, tests, and XML), are generated [25].

Mutants are produced by mutation operators, which generate flaws in the original code. Those changes can be made by modifying expressions, adding, changing, or removing operators and/or statements. Mutation operators are designed to mimic simple programming mistakes that programmers make (e.g. using a wrong variable name). Thus, operators differ depending on the language the programmer is working on. Therefore we conclude they are language dependent.

Operators introduce flaws into the system, by following the rule that describes the mutation change, in a systematic and repeatable way, allowing the program to be tested efficiently.

A mutation testing procedure is shown in Figure 2.1.

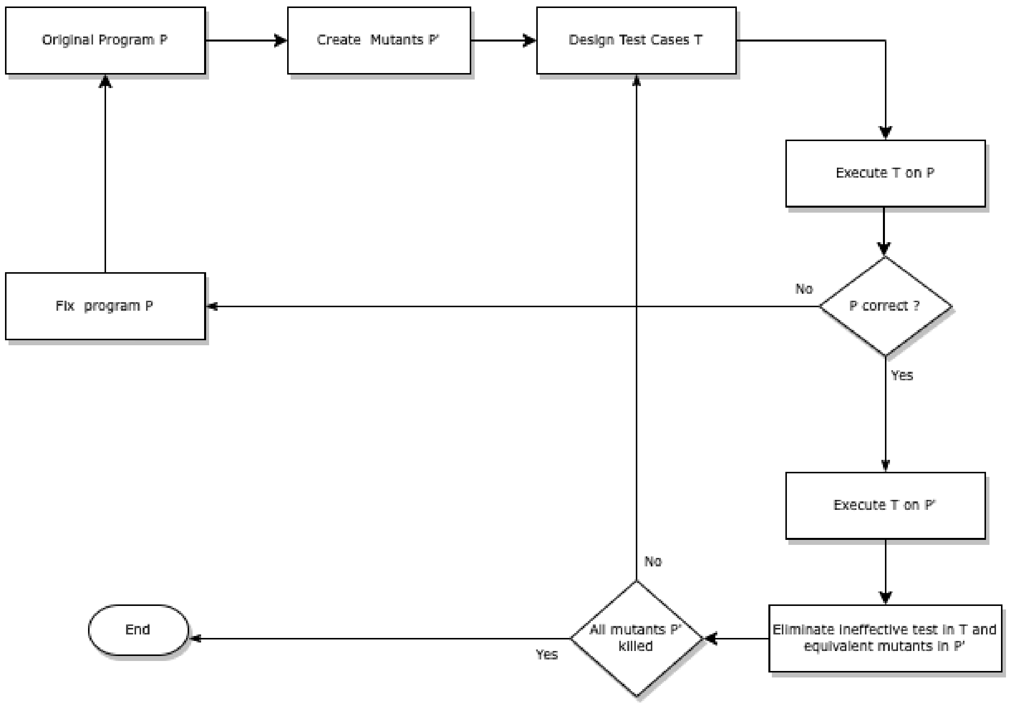


Figure 2.1: Mutation Testing Process

Firstly, mutants, designated as *P'*, are produced using a faulty version of the original program under test *P*. Once the mutants are applied in the source code, they are executed against the test suite, a collection of test cases that are grouped for test execution purposes [26] and designed by testers.

Then we must analyze the test result of each mutant and compare it with the result of program *P*. A test is considered to kill a program if it results in different outputs on the original *P* and a mutant program in *P'*. If the test case detects the mutants, they are called *Killed mutants*. Nevertheless, not all mutants generated will get killed. There are some mutants known as *Equivalent Mutants* which can never be killed, even if additional test cases are provided by testers in order to increase the number of mutants killed. They always yield the same outcome, despite being syntactically distinct. Due to the difficulty of autonomously detecting these types of mutations, human intervention is required for their detection and removal in order to assess test data quality. Another type of mutant is *Stillborn mutants*, which cannot be compiled due to the changes self-made, resulting in a program with an incorrect syntactic structure. Although they can be avoided if the mutation operators are thoughtfully created and properly applied, they cannot be eliminated entirely. A mutation system must be capable of identifying stillborn mutants and discarding them.

To evaluate the quality of the test cases generated, the mutation score is calculated. *Mutation score (MS)* is the percentage of mutants killed by the total number of non-equivalent mutants as exemplified in equation 2.1. The mutation score, which can go from 0% to 100%, is a quantitative indicator of how effective a test suite is. The more mutants a test suite can kill, the more effective the test set is [27]. A test suite is *mutation adequate* if the mutation score is 100%.

Usually, it is impractical to obtain a score of 100% in MS, so testers define a "threshold" value, which is a minimum acceptable mutation score. While the threshold is not attained, then the process is repeated, and new test cases are generated to target alive mutants [28].

$$MS(P,T) = \frac{M_K}{M_T - M_E} \times 100\% \tag{2.1}$$

- M_K - Number of killed mutants
- M_T – Number of all mutants generated
- M_E - Number of equivalent mutants

To better understand the process depicted, a snippet of an Arithmetic Operator Replacement mutation (AOR) for Java is shown in Figure 2.2. AOR replaces each occurrence of arithmetic operators (+, -, *, /, %) with another arithmetic operator [27].

In this particular case, on the original program, the sum operator was a plus that was replaced by a minus on the mutated version. The next step is to create a test case that can be able to kill this mutant. An example of a test to eliminate a mutation is shown in Table 2.1.

Original

```
public int sum(int a, int b) {
 int sum = 0;
 sum = a + b;
 return sum;
}
```

Mutant

```
public int sum(int a, int b) {
 int sum = 0;
 sum = a - b;  //AOR Mutant
 return sum;
}
```

Figure 2.2: Arithmetic Operator Replacement Example

Table 2.1: An Example Of Killing Mutant

	Ineffective Test	**Killing Test**
Input	a=5 & b = 0	a = 1 & b = 1
Original Output	5	2
Mutant Output	5	0

2.1.2.1 Mutation Operators

As stated in the previous section, mutation operators are language dependent. Having said that, this section dissects mutation operators for several languages(Fortran IV, COBOL, Fortran 77, C, C integration testing, Lisp, Ada, Java, and Java class relationships), providing a detailed overview of the same [28].

Traditional Mutation Operators The traditional mutation operators, depicted in Table 2.2, seek to provide equivalence operators across different languages. The traditional mutation operators are based on the operators described for Ada and Fortran and are utilized by the Mothra tool [9]. Table 2.3 further complements these operators by presenting some examples of the same.

Table 2.2: Traditional Mutation Operators from [9].

Mutation Operator	**Description**
AAR	array reference for array reference replacement
ABS	absolute value insertion
ACR	array reference for constant replacement
AOR	arithmetic operator replacement
ASR	array reference for scalar variable replacement
CAR	constant for array reference replacement
CNR	comparable array name replacement
CRP	constant replacement
CSR	constant for scalar variable replacement
DER	DO statement alterations
DSA	DATA statement alterations
GLR	GOTO label replacement
LCR	logical connector replacement
ROR	relational operator replacement
RSR	RETURN statement replacement
SAN	statement analysis
SAR	scalar variable for array reference replacement
SCR	scalar for constant replacement
SDL	statement deletion
SRC	source constant replacement
SVR	scalar variable replacement
UOI	unary operator insertion

Table 2.3: Examples of Some Traditional Mutation Operators from [9].

Operator	Description	Example
ABS	Absolute Value Insertion	$a = b + c$ –> $a = 0$
AOR	Arithmetic Operator Replacement	$a = b + c$ –> $a = b$-c
LCR	Logical Connector Replacement	$a = b\&c$ –> $a = b\|c$
ROR	Relational Operator Replacement	$while(a<b)$ –> $while(a>b)$
UOI	Unary Operator Insertion	$a = b$ –> $a =$-b

Class Mutation Operators There are four types of class mutation operators, such as Encapsulation, Inheritance, Polymorphism, and Java-Specific Features, according to Java language characteristics.

The useful mutation operator is one that can handle all of a programming language's conceivable syntactic changes. In general, mutation operators can be generated by the following methods: remove, insert, or modify a target syntactic element.

For the Java programming language, 29 class mutation operators were found for object-oriented and integration testing. Table 2.4 contains a list with some example class mutation operators [9].

The class mutation operator should be used at different levels given the nature of object-oriented languages. Because object-oriented languages are class-based, mutation operators must handle mutations linked to in-class and out-class language behaviour. These stages are illustrated below:

- **Unit level:** at this level, apply classical operators to a class function or method to ensure its validity.
- **Class level:** This level is concerned with the modification of object-oriented characteristics.
- **Integration level:** Checking function invocations at an intermediate level between the unit and system levels.
- **Multi-class level:** These operators are intended to test the entire programme, including interactions between functions, classes, and so on.

Table 2.4: Example of Some Class Mutation Operators from [9].

Category	Operator	Description	Example
Inheritance	AMC	Access Modifier Change	*public* Stack s; –> *private* Stack s;
Polymorphism	PNC	New method call with child class type	$a = new$A(); –> $a = new$ B(); where B is subclass of A;
Overloading	OAN	Argument number change	$s.Push(0.5, 2);$ –> $s.Push(2);$
Java-specific	JTD	"This" keyword deletion	$this.size = size;$ –> $size = size;$
Common Programming Mistakes	EOA	Reference assignment and content assignment replacement	$list2 = list1;$ –> $list2 = list1.clone();$

2.2 Android Applications

Android is an open-source and Linux-based Operating System for mobile devices such as smartphones and tablets. Android was developed by the Open Handset Alliance, led by Google and other companies [29].

Android applications can be written using Kotlin, Java, and C++ languages. Additionally, Android provides developers with a rich Software Development Kit (SDK), which includes a set of tools to compile source code, resource files (e.g., pictures, audios, and videos), and data into Android Application Pack (APK). This file comprises all the contents of an Android app. APK files are also required for installing the application on Android devices [30].

Every project in Android includes a Manifest XML file, which is AndroidManifest.xml, located in the root directory of its project hierarchy [31].

The manifest file defines the structure and metadata of the application, including configuration information and descriptions of the apps' components [31].

Application components are the building blocks of an Android app, as it is through each component that the user or system is able to enter the app. There are four different types of app components: Activities, Services, Broadcast receivers, and Content providers, and they are dependent on each other. Each type has a distinct purpose and a specific life cycle that defines how the component is created and destroyed [32].

The following section describes the five types of application components.

2.2.1 Android Activities

The activity component serves as the entry point for an app's interaction with the user. An activity presents a screen (graphical user interface (GUI)) to the user based on one or more layout designs,[1]. A GUI can contain widgets such as buttons, text views, and other advanced artifacts [4]. The majority of applications contain multiple screens, which means multiple activities coexist. Typically, the first screen to be displayed when the user starts the app is called the main activity. Each activity can start a new activity in order to perform different actions.

An activity can move through several stages, as it is demonstrated in Figure 2.3.

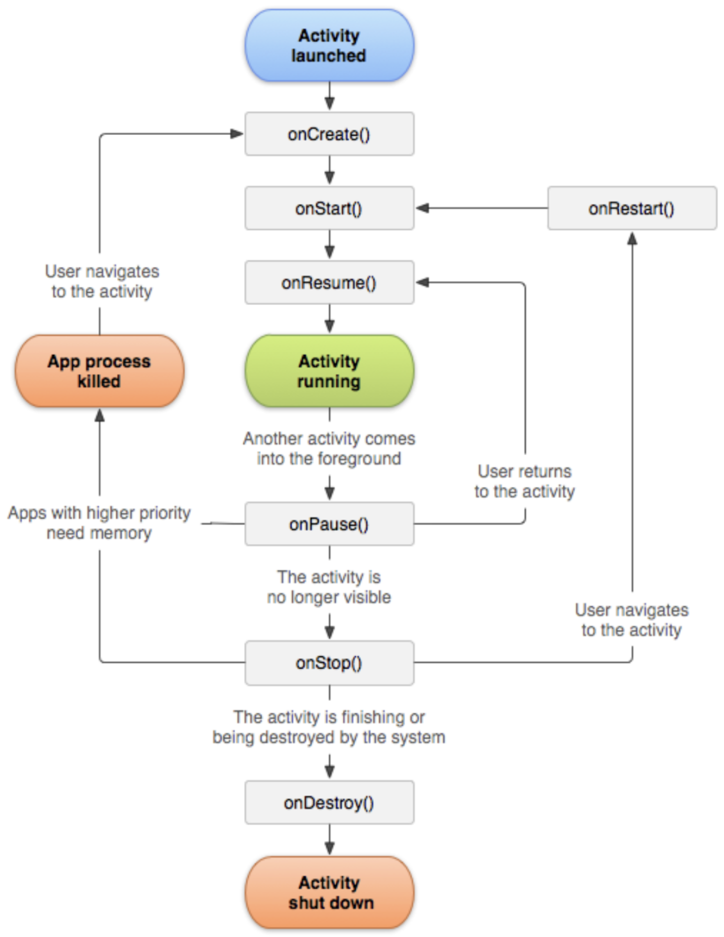

Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Figure 2.3: A simplified illustration of the activity lifecycle, image from [1].

Every activity must have the *onCreate()* callback, which is invoked when the system creates an activity and provides information about the activity's starting logic. When this procedure is finished, the *onStart()* function is invoked. This method prepares the activity before it is presented and ready for user interaction [1]. When the activity reaches the resume state, the callback *onResume()* is called. Furthermore, as long as there is no action that brings another activity to the foreground, this is the state that remains active and where the interactions with users take place. The *onPause()* method is eventually called by the activity, which stops any functions that are not necessary to be running while the activity is running in the background. In addition, this is the state in which it will be determined whether the user will use the activity by returning to the resumed state and calling its corresponding callback method or whether it should be stopped due to memory constraints or the activity's disappearance.

When the activity is no longer visible, it prompts the *onStop()* method to be called, which stops all heavy functionalities and some resources associated with that activity. The activity can then be either restarted using the callback *onRestart()* or destroyed using *onDestroy()* [1].

2.2.2 Services

A service is an element of an application that has the ability to do ongoing tasks in the background without any user interface being provided. Even if the user goes to another application after starting a service, the service may remain running for a while. Furthermore, a component can communicate with a service by binding to it and even carry out interprocess communication (IPC). For instance, a service can operate in the background while handling network transactions, playing music, performing file I/O, or interacting with content providers [33].

Three different types of services exist:

- **Foreground**: A foreground service performs actions that are visible to the user. A notification must be displayed so that consumers are aware that the service is operational. For example, an audio app might use a foreground service to play an audio track [33].
- **Background**: A background service conducts an operation that the user is not aware of [33].
- **Bound**: A bound service provides a client-server interface. It enables components to communicate with the service, send requests, and get results across processes via interprocess communication (IPC) [33].

2.2.3 Broadcast receivers

A broadcast receiver is a component that allows the system to send events to the app that are not part of a regular user flow, allowing the app to respond to system-wide broadcast announcements. Because the broadcast receivers are an entry point into the application, broadcasts can be delivered to applications that are not currently running.

Many broadcasts originate from the system. Nonetheless, it can also be initiated by Apps. Although broadcast receivers do not display a user interface, they may create a status bar notification to alert the user when a broadcast event occurs. More commonly, though, a broadcast receiver is just a gateway to other components and is intended to do minimal work. For instance, it might schedule a JobService to perform work based on the event with JobScheduler. A broadcast receiver is implemented as a subclass of BroadcastReceiver, and each broadcast is delivered as an Intent object [34].

2.2.4 Content Providers

Content Providers manage a shared set of application data that can be stored in file systems, SQLite databases, the web, or any persistent storage location accessible to our application. Through the content provider, other apps can query or even modify the data if the content provider allows it. It is tempting to think of content providers as an abstraction in a database because plenty of APIs and support are built into them for this common case. However, they have a different primary purpose from a system development perspective. For the system, the content provider is an entry point into an application for publishing named data items identified by a URI schema. Thus, an application can decide how it wants to map the data it contains to a namespace URI. This transfers those URIs to other entities, which can then use them to access the data [35].

2.2.5 Intents

In Android, intents are considered messaging objects that may be used to request actions from other components or applications. Although intents facilitate communication between components in a variety of ways, there are three fundamental use cases: *(1) starting activities by passing the intent to the method, (2) starting services using a JobScheduler, (3) delivering broadcasts using the sendBroadcast() or sendOrderedBroadcast() methods.*

An Intent object contains information used by the Android system to decide which component to initiate (the specific name of the component or category of the component that should receive the intent), as well as information (the action to take and the data to use) used by the receiving component to complete the action correctly.

Moreover, these intents may either be explicit when demanding to launch a specific application component by specifying the component name or implicit when there is no specification of the app to be launched. In this case, the system analyses and determines which of the installed apps support the said intent by comparing the information of the intent with the intent filter and, afterwards, displays them to the user.Figure 2.4 shows an example of this selection process.

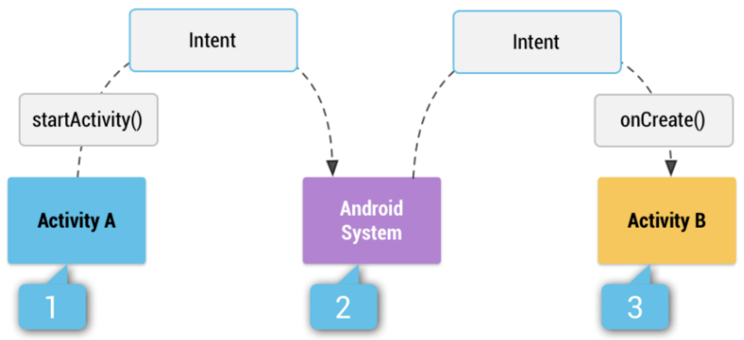


Figure 2.4: Implicit intent being delivered to the correct activity, image from [2].

After the overview of how mutation testing and Android applications work, we will explain, in the next section, how mutation testing can be applied to Android development.

Chapter 3

State of the Art

Despite existing for a long time and being thoroughly studied, mutation testing is still not widely used. However, the software industry is showing a substantial surge in interest in applying this technique to mobile applications due to the impact mobile apps have on our everyday activities. Nonetheless, due to Android applications' unique features, numerous challenges arise while trying to test them.

Although the first article on android-specific mutation testing was published in 2015 [36], the subject has sparked the most attention in the scientific community in the last five years. As a result, roundly 98% of the studies available were published within this period [10].

As the study of mutation testing's applicability in Android apps is still in its early stages, most research works focus on analyzing the adaptability of traditional mutation operators for mobile software and proposing new Android-specific operators capable of reproducing common failures in the environment.

Nevertheless, although the cost of mutation tests is quite high, articles that analyze and attempt to apply cost-reduction techniques in this new testing environment are rare.

As a result, in an attempt to comprehend the current status of mutation testing for Android applications, we will summarise all findings made in the field up to this point.

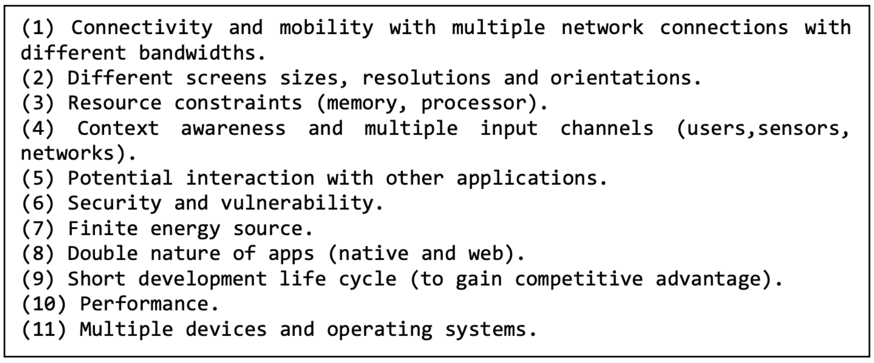
Furthermore, in order to understand the impact of applying a cost-reduction technique to the mutation process, a recent study (published in 2021) by Diego Naveiras *et al.* [3] will be explored. Based on the results, we can determine whether this new approach enhances the applicability of mutation testing in the industrial environment.

This section covers the existing literature on Android mutation testing.

3.1 Mutation testing on Android applications

According to Rene Just *et al.* [37], traditional Java mutation operators are not sufficient to assess all sorts of Android faults and must be accompaniment by alternative mutation operators.

Due to the fact that mobile software has unique characteristics, which are enumerated in Figure 3.1, we can not use the Java mutation analysis process nor the Java mutation operators. Mobile special characteristics have a strong influence on testing.



```
(1) Connectivity and mobility with multiple network connections with
different bandwidths.
(2) Different screens sizes, resolutions and orientations.
(3) Resource constraints (memory, processor).
(4)  Context  awareness  and  multiple  input  channels  (users,sensors,
networks).
(5) Potential interaction with other applications.
(6) Security and vulnerability.
(7) Finite energy source.
(8) Double nature of apps (native and web).
(9) Short development life cycle (to gain competitive advantage).
(10) Performance.
(11) Multiple devices and operating systems.
```

Figure 3.1: Special Characteristics of Mobile Software, image from [3]

The difference stems from the fact that Android operators must also alter XML layout and configuration files, unlike Java mutation analysis tools, which only alter Java files. Additionally, each Android mutant must be converted into an APK file in order for it to be installed and run on mobile devices and emulators, in contrast to traditional Java mutation analysis tools that usually convert mutated Java source files into bytecode Java class files that are then dynamically linked by the language system during execution [36].

To suppress that need, in 2015, Deng *et al.* [36] introduced an innovative approach to conducting mutation analysis for Android apps. A mutation analysis tool was proposed along with eight mutation operators in order to generate errors in the basic programming elements of Android applications, such as intent, event handler, activity cycle, and XML files (like GUI and permissions files), along with the traditional java operators from Mujava. Through this study, it was found that mutation testing can be an interesting approach to analyzing the unique features of Android.

Additionally, Deng *et al.*[38] assess the usefulness of mutation testing for defect identification in Android apps. They discovered difficulties in testing Android apps, investigated frequent flaws in Android app development, and created 17 unique Android mutation operators to assist testers in dealing with these issues. Their investigation employed two methods to gather faults in

Android apps: mining open-source repositories for naturally occurring defects and crowdsourcing problems through a freelancer website.

Later, in 2017 Usaola *et al.*[39] presented an abstract specification for creating and implementing operators for mobile applications that are context-aware. All operators are specializations of a collection of abstract classes. Their purpose is to make new operator development easier and to separate operator implementation from external libraries used for altering Java bytecode and introducing errors.

Afterwards, Linares-Vasquez *et al.* [40] introduced the first taxonomy of faults in Android apps. The taxonomy is divided into 14 categories, each with 262 kinds. Then, based on the taxonomy, a set of 38 Android-specific mutation operators was designed, deployed in an infrastructure called MDroid+ that automatically seeds Android app changes from the source code.

Jabbarvand and Malek [41] innovated by proposing mutation operators for testing the behaviour of apps with different energy-consumption rates. They looked for 'energy anti-patterns' and created 50 operators based on them, such as increasing the frequency of location update requests or not turning off Bluetooth. The name of their testing framework is *µDroid*. The *µDroid* creates mutants and compares their power usage to that of the original software in order to determine if they are terminated. Regarding testing execution time, the authors only give the meantime frames for assessing whether mutants are killed (i.e., comparing energy consumption traces). The average time in their nine experiments is 11.7 seconds.

In 2019, Paiva *et al.*[42] described three mutation operators for testing the unique behaviour of mobile applications linked to the non-preservation of the UI state when apps are sent to the background and then returned to the foreground. The iMPAcT tool is intended to handle mutants produced by these operators [43; 44; 45; 46; 47; 48; 49; 50; 51].

A recent article from Escobar-Velásquez *et al.* [52], in 2020, extended a previous study of 2017 [40]. According to their findings, 65% of the faults are typical of any Java program, while the remaining 35% are explicitly tied to Android-specific traits. A unique and intriguing contribution is their MutAPK tool which inserts the defects straight into the compiled and packaged APK file. Mutating at the APK level with MutAPK is quicker and minimizes the ratio of non-compilable mutants, but it requires more mutants than when working on the source code and it also requires additional effort to understand bytecode instructions, since is a low-level representation of code.

With that said, we can confirm there is widespread agreement on the importance of having specialized mutation operators that recreate typical flaws in Android applications.

Nevertheless, few studies were found that took into consideration the analysis, modification and recommendation of the use use of specific strategies to lower the cost of mutation testing in the Android environment. Deng *et al.* [4] was the first to propose parallel execution, the usage of fewer mutations and the implementation of a faster test framework in order to improve performance.Later Linares-Vasquez et al. [53] have also used parallel execution.Recently, Luna *et al.* [54] have proposed applying strong mutation testing.

We will now describe the process proposed by Deng et.al [36] for conducting mutation analysis on Android apps.

3.1.1 Mutation analysis on Android apps

On the topic of Mutation analysis on Android apps, Figure 3.2 illustrates the process proposed by [4].

The first step to start a mutation process is to enter the location of the folder that contains the Android source files. Then, the next step is select which mutation operators to use. If Java mutation operators are selected, the system alters the original Java source code and converts it to bytecode class files. In the case of XML mutation operators, these are directly applied to the XML file, generating a new copy of the file for each mutant. When the APK file is created, they are switched into position for dynamic binding. The mutation system creates a mutated APK file from each modified Java bytecode class file and XML file by incorporating the mutated source and other project files. Some mutations may result in compilation problems. These stillborn mutants are promptly eliminated and are not included in the final results.

Android testing automation frameworks such as Robotium [55], Espresso [56], Selendroid [57], and JUnit [58] are supported by the Android mutation analysis tool. To eliminate mutants, test cases can be developed by testers using Android testing automation frameworks, or a set of externally created test cases, such as tests from other automated test creation tools, can be employed.

After creating mutants and compiling them into APK files, the system launches an emulator or a mobile device with the original (non-mutated) version of the app under test. The system then runs all test cases on the original app and logs the results as anticipated. To decide which mutants are killed, the outcomes of the mutant executions are compared with the results of the original app.

Afterwards, mutants are put into an emulator or a mobile device. The mutation system runs all of the test cases against the mutations and records the results.

Once all of the findings have been collected, the mutation system compares the predicted outcomes to the actual results. If the actual result of a test varies from the predicted outcome of the same test, the mutant is considered to have been killed by that test.

Finally, the mutation score is computed as a proportion of mutants who died as a result of testing.

Because the technology currently lacks any algorithms to assist in identifying comparable mutations, these must all be examined manually.

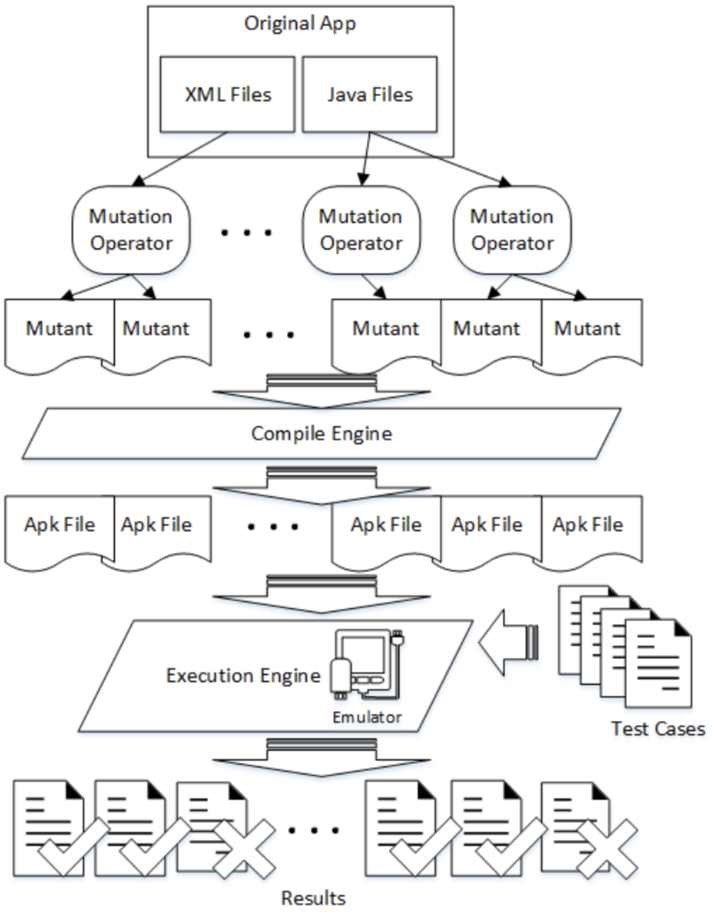


Figure 3.2: Performing Mutation Analysis on Android Apps, image from [4]

3.2 Cost reduction techniques

The mutation testing approach is widely recognized as one of the most effective methods for assessing the effectiveness of a test suite [38]. However, it has yet to acquire general acceptance in software engineering practice. Despite its effectiveness, many software testers and companies dismissed the method due to its limitations. According to researchers, the unpopularity of industrial mutation testing is attributable to the high costs associated with the method [59]. Creating, running, and executing a large number of mutants against a test suite is viewed as exceedingly costly, time-consuming, and tedious since it demands significant computing resources and vast storage space [6].

In order to turn Mutation Testing into a practical testing technique, several cost-reduction techniques have been proposed. Throughout the mutation process, various of these same techniques can be applied in order to reduce the cost of the mutation process. Figure 3.3 shows the numerous existing techniques in chronological order.

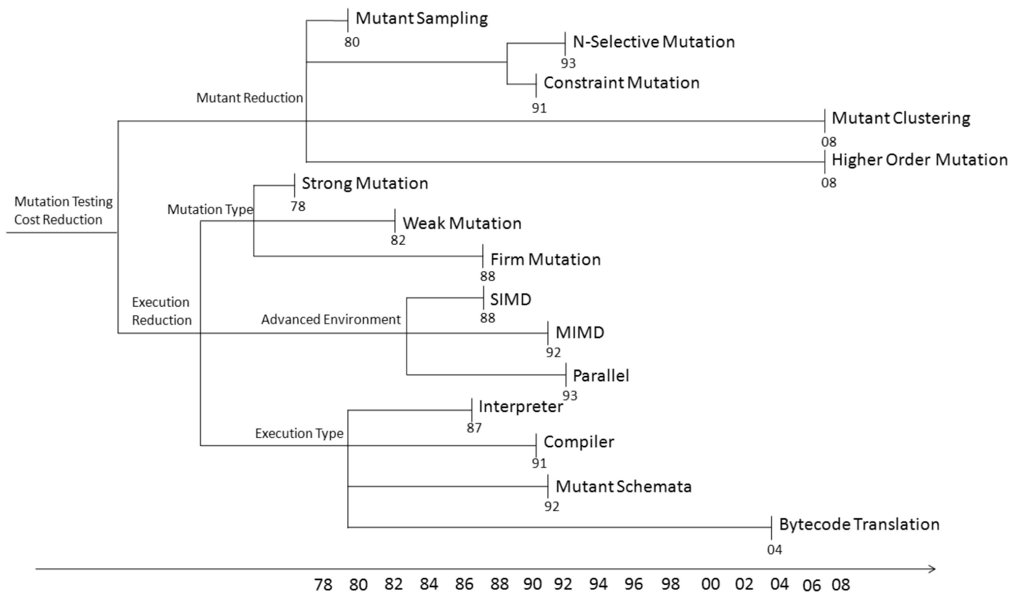


Figure 3.3: Overview of the chronological development of mutant reduction techniques, image from [5].

Regarding the excessively cost of mutation testing, the most expensive factor is, without doubt the number of mutants generated, and all the subsequent processes depend on it. So as to reduce the number of mutants produced, several mutant reduction approaches have been suggested, including Mutant Sampling, Mutant Clustering, Higher Order Mutation, and Selective Mutation.

Mutant sampling was first proposed by Acree and Budd. Its main objective is to select, randomly, a specific percentage of mutants to execute testing [60; 61], resulting in the decrease of mutants used and thus a lower time and cost spent.

Mathur and Wong investigated the influence of the sampling rate on mutation adequacy [62]. They conducted a series of trials by changing the rate from 0.1 to 0.4, and the experimental results suggested that the mutation score decreased by only 16% with a sampling rate of 0.1, when compared to the usage of a full set of mutants, establishing that Mutant Sampling is valid with an x value higher than 10%.

De Millo *et al.*[63] and King and Offutt [64] also corroborated these results. Zhang *et al.*[65], in their report, acknowledged that by killing randomly selected sets of mutants, composed of more than 50% of the initial set, the killing rate of all the mutants was more than 99%.

Derezińska and Rudnik *et al.*[66], recently suggested different mutant sampling criteria based on equivalence partitioning with respect to object-oriented program features. The results of the analysis indicated that class random sampling and operator random sampling are recommended for object-oriented (OO) in standard mutation testing, as the mutant sampling technique is easily applicable in comparison to other cost reduction techniques [3].

Another technique recommended is **Selective mutation**. This method was first introduced by Mathur[67] by the removal of two mutation operators (i.e., Array reference for Scalar variable Replacement (ASR) and Scalar Variable Replacement (SVR)), which generate around 30% to 40% of the total mutants. Later, this hypothesis was extended by Offutt, Rothermel, and Zapf [68], which stated that the number of mutants could be reduced by applying only a subset of the mutation operators. In a recent research work conducted by Namin *et al.* [69], was applied a linear statistical approach to identify a subset of 28 mutation operators from 108 C mutation operators. The outcomes implied that these 28 operators were sufficient to predict the effectiveness of a test suite, and it reduced 92% of all generated mutants, achieving the highest rate of reduction compared with other approaches. Hence, this method aims to find a small set of mutation operators that can generate a subset of all possible mutants without a major loss of test efficiency, therefore reducing the cost.

Regarding **Mutant Clustering** introduced by Shamaila Hussain, this approach emphasizes choosing a subset of mutants using clustering algorithms (e.g., K-means, DBSCAN...) that classify the first-order mutants into different clusters based on the killable test cases. Therefore mutants in the same cluster could be killed by the same test instead of selecting mutants randomly. In his research, Hussain's empirical results [70] suggest that Mutant Clustering is able to select fewer mutants, nonetheless upholding the same mutation score.

Lastly, **Higher Order Mutation** is a form of mutation testing introduced by Jia and Harman [71]. In their analyses, Jia and Harman applied meta-heuristic search algorithms to generate semantic higher-order mutants (two or more mutants into the same mutated program), which are more resistant, and significantly reduced the number of mutants used [72]. The empirical results of Polo *et al.* [73],[74] suggest that applying second-order mutants reduces the test effort by approximately 50%, without much loss of test effectiveness. Furthermore, Papadakis and Malevris uncovered that second-order strategies could reduce 80% to 90% of the equivalent mutants, with approximately 10% or less of test effectiveness loss.

More recently, Abuljadayel and Wedyan [75] presented an approach to generate higher-order mutants using a genetic algorithm, also more challenging to kill than first-order mutants.

Within the techniques described above, **mutant sampling** and **selective mutation** are integrated into the First Order Mutants (FOMs) perspective. Regarding High Order Mutants (HOMs), which represent more than one syntactic change in a program, the same not only embody complex faults in practical software but also reduce the number of mutants [71].

Apadakis and Malevris conducted an empirical study for the first and second-order mutation testing strategies and found that the first-order mutation testing strategies are generally more effective than the second-order ones. However, second-order mutation strategies demonstrated a drastic decrease in equivalent mutants, thus establishing a valid cost-effective alternative to mutation testing [76]. One downside of higher-order mutants is that cost of generating these mutants is significantly higher compared with first-order mutants, which further illustrates the necessity of reducing mutants.

The cost of mutation testing can additionally be decreased by altering the execution process. To accelerate the mutant generation and test case execution, more strategies were suggested.

Establishing the approach of the analysis of the execution process, Mutation Testing techniques can be classified into three types, Strong Mutation, Weak Mutation, and Firm Mutation. **Strong Mutation** is considered the traditional Mutation Testing, introduced by DeMillo *et al.* [5]. For a given program *p*, a mutant *m* is said to be killed only if mutant m gives a different output from the original program p. To optimize the execution of the Strong Mutation, Howden [77] proposed **Weak Mutation**, which, instead of checking mutants after the execution of the entire program, can check immediately after the execution point of the mutant or mutated component. However, as different components of the original program may give different outputs from the original execution, weak mutation test sets can be less effective than strong mutation test sets. Nonetheless, a theoretical proof of Weak Mutation by Horgan and Mathur [78] disclosed that under certain conditions, test sets generated by weak mutation could also be expected to be as effective as strong mutation. Lastly, **Firm Mutation** was first proposed by Woodward and Halewood [79], and its main goal was to overcome the shortcomings of both weak and strong mutations by providing a continuum of intermediate possibilities, laying between the intermediate states after execution (Weak Mutation) and the final output (Strong Mutation).

Lastly, run-time optimization techniques include byte-code translation and mutant schemata.

Mutation at bytecode level is a method that injects the changes directly in the compiled code, avoids the cost of mutant compilation, and can be used in programs that do not have available source code. This technique has been used by tools such as MuJava [80], Javalanche [81], and Bacterio [82]. However, not all programming languages provide an easy way to manipulate intermediate source code. There are also some limitations of Bytecode Translation application in Java, such as the impossibility of some of the mutation operators to be represented at the Bytecode level.

Mutant Schemata is another strategy available designed to reduce the total cost of mutation testing. The central idea of this technique is to compose different programs into a metaprogram (all program versions are included in a single file). To determine which of the program versions included in the schema is to be executed, a control mechanism must be implemented. This technique's cost comprises a one-time compilation cost and the overall runtime cost. This metaprogram is a compiled program, so its runtime performance is faster than the interpreter-based technique. The pioneering work regarding Mutant Schema is from Untch, Offutt, and Harrold [6], who created a mutant schema generator for Fortran. They used *metamutants* and *metaprocedures*. A *metamutant* contains all the mutants in a single file as a set of *metaprocedures*, which are functions that gather the different changes introduced by mutation operators. Papadakis and Malevris [83] apply the original Untch *et al.* 's approach but adjust it to symbolic execution. Offutt and Kwon *et al.* [80] adapt the idea to Java programs, more specifically, in the MuJava tool, automating the *metamutant* generation. These authors created *metaprocedures* for the object-oriented characteristics, such as inheritance, polymorphism, and instantiation overhead. This approach was reused by other authors, such as Kim, Ma, and Kwon, in [84]. Reales and Polo *et al.* [85] and [86] include *metamutants* instrumenting the original Java bytecode with the insertion of if-else statements.

3.3 Mutation testing using Mutant Schemata in Android

Regarding mobile applications, the biggest concern related to mutation testing is the time required to compile, connect, install, and run the System Under Test (SUT) and its modified versions on the mobile device.

Despite existing papers [80; 86; 87; 88; 89], addressing the application of mutant schemata strategy, no author provides enough technical information to allow an exact replication of the methodology, only mention to Untch *et al.*[6] as a reference.

To understand what the process entails, a brief description of the method follows.

3.3.1 The Mutant Schema Generation Method

The Mutant Schema Generation Method (MSG) method consists of creating a specially parameterized program called *metamutant*. The *metamutant*, originating from the program *P* under test, is compiled using the same compiler used to compile the program *P* and executes at compiled speeds. During execution, the *metamutant* can be instantiated to show the execution behaviour of any of the alternative programs found in N, the program neighbourhood of P.

A list of mutant descriptors, D, is generated when the *metamutant* of P is created. These mutant descriptors are utilized to dynamically instantiate the *metamutant* in order for it to act as a mutant of P. Each mutant in the neighbourhood of P has one mutant description. Each mutant descriptor is made up of a collection of *metamutant* parameter values that describe a specific mutation.

A mutant description must have at least two elements: a mutation point and an alternative action to be executed at that mutation point. A *driver* or *harness* calls the *metamutant* and specifies which mutant to instantiate by picking the matching mutant descriptor from D.

During the execution of the *metamutant* through each internal change point, a check is conducted to see if the change point matches the mutant description. If this is the case, the alternative (mutating) action is carried out. Otherwise, the default (non-mutating) action is carried out. The driver handles administrative concerns such as managing test case input and output, exception handling, comparing the mutant output to the original program output, and logging the results, in addition to picking mutant descriptors from D and calling the *metamutant*. The driver also computes and presents statistics information on the mutant's current condition, most notably the mutation score. All *metamutant*s share a driver [87]. Figure 3.4 provides a conceptual model for using the MSG approach, where G means the set of mutants.

$$\begin{tikzcd} \left.\begin{matrix}P\\G\end{matrix}\right\} \arrow[r, Rightarrow, "\text{mutation}"] & N \arrow[r, Rightarrow, "\text{substantiation}"] & \left\{\begin{matrix}M\\D\end{matrix}\right\} \arrow[r, Rightarrow, "\text{instantiation}"] & P_i \arrow[rr, Rightarrow, "\text{execution}"] & & MS_G(P,T) \\ & & & & T \arrow[u] & \end{tikzcd}$$

Figure 3.4: Model Of MSG Method, image from [6].

3.3.2 Untch Mutant Schema

In 2021, Diego Naveiras [3] proposed its own approach to Untch's implementation [6] which is adapted for Android mobile apps.

3.3.2.1 Untch Mutant Schema process

This process was implemented in the *BacterioWeb v.2* testing tool, which is a web tool for the mutation testing of Android mobile applications in a distributed environment, also developed by Diego Naveiras as an evolution of *BacterioWeb v.1* [39].

Untch Mutant Schema (UMS) needs to analyze the program code to perform the following operations: detect the statements to be changed, substitute the original statements by calls to the *metaprocedures*, create the *metaprocedures*, and implement the test driver. All without making any copy of the system under test (SUT) [3].

To begin the testing process, the tester must select the files to be mutated. Then, the next step is to choose which mutation operators to use. Subsequently, mutants are generated.

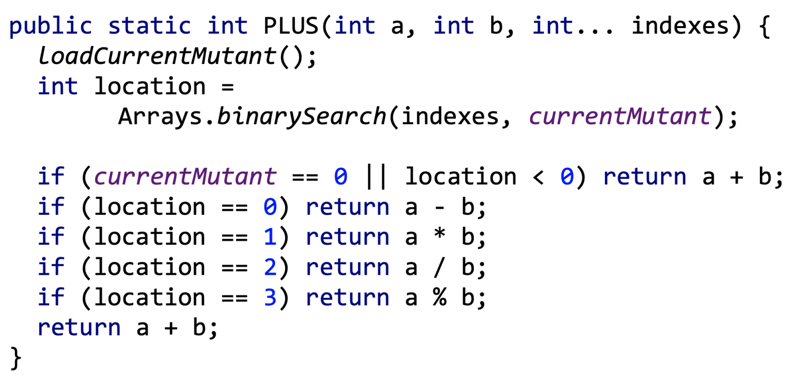
Following that, each Java source file is processed with the Javaparser library [90], which constructs the abstract syntax tree (AST) of each processed file. The source code and serialized AST are saved in the relational database utilizing the Javaparser serialization functionalities.

The mutant schema generator then iterates, attempting to apply each specified mutation operator to the file under consideration.

Consider the classic AOR operator, which reads all binary expressions in the file and updates the original statement by calling MutantDriver.X, where X is the name of the original operator if the matching operator is one of these $+,-,\times,/,or\%$.

Observe the statement $a+b$. If we substitute the operator by a call to a PLUS method in a MutantDriver, the statement can be rewritten as MutantDriver.PLUS(a, b).

The MutantDriver implements the PLUS(int x, int y, int... indexes) metaprocedure as seen in Figure 3.5, where the first two parameters are the integers to be added and the others are the mutant indexes.



```
public static int PLUS(int a, int b, int... indexes) {
  loadCurrentMutant();
  int location =
        Arrays.binarySearch(indexes, currentMutant);

  if (currentMutant == 0 || location < 0) return a + b;
  if (location == 0) return a - b;
  if (location == 1) return a * b;
  if (location == 2) return a / b;
  if (location == 3) return a % b;
  return a + b;
}
```

Figure 3.5: Implementation of PLUS in the MutantDriver, image from [3].

Every operator in this case may be substituted by the other four operators. This means that:

- $+$ can be replaced by $-,\times,/,\%$
- $-$ can be replaced by $+,\times,/,\%$
- $\times$ can be replaced by $+,-,/,\%$

- $/$ can be replaced by $+, \times, -, \%$
- $\%$ can be replaced by $+, \times, /, -$

Thus, MutantDriver.PLUS(a, b) will be written as MutantDriver.PLUS(a, b, 1, 2, 3, 4) to reflect the four potential mutants. The mutant indexes are referenced by the (1-4) which are all the possible exchanges.

Below are three scenarios to help you understand how it works, based on what has been said.

The original program, which has a mutant index of 0 (zero), is being tested. The PLUS implementation reads the value of currentMutant from a file and since it has a mutant index of 0, it provides the same result as the original program, which is $a+b$, as shown by the first if, in Figure 3.5.

The third mutation after $a+b$ is now being executed, and its index value is 3. In the loadCurrentMutant method, this value, which was saved in the aforementioned file, is given to currentMutant. The value (3) is searched in the array passed in the variable parameters, which included the values [1, 2, 3, 4], and found with $location = 2$. As a result, the method returns a/b.

When running a test suite against a mutant with index number 10, which is not present in the array. The function returns the expression from the first if, $a+b$.

3.4 Mutation testing vs Mutation testing using Mutant Schemata

Mutation testing based on mutant schemata method has numerous advantages when compared to the traditional method. The most obvious advantage is that MSG mutation systems are faster than interpretative systems [6].

As previously said, one limitation in mobile mutation testing is the deployment of the program on the device. Using the traditional mutation technique the application must be deployed once per mutant. When employing the Mutant Schema approach, all mutants may be packed into a single version of the application, reducing the number of deployments to one.

Another advantage is that mutation testing using mutant schemata is more accurate than traditional mutation testing since the mutant schemata approach generates a compilable program in the same language as the program under test, allowing testing to be done using the same compiler and environment as the program under test. As a result, the software is tested in the same operational context in which it will be deployed, and it keeps all or almost all of its original operational behaviour [6].

The inherent portability of mutant schemata mutation systems is a substantial benefit, considering it can be readily migrated from machine to machine or compiler to compiler since they work at the source level [6].

The mutant schemata method, on the other hand, does not disclose many technical specifics regarding the mutant schemata structure or the controller in charge of assigning the current mutant [3].

On many occasions, when the mutant schemata technique is used, several mutants overlap, resulting in a number of mutations that do not always match the number generated by the traditional way. Furthermore, trivial mutants, which are mutants that always or frequently fail during runtime, are often produced. On the other hand, the implementation of the mutant schemata method is easier, and structure assembly is faster, but the readability of the mutant code is heavy [3].

3.5 Android Specific Mutation Tools

There are few mutation testing tools available for Android applications. In this study, we considered only the ones that are specific to Android testing. A total of six tools were analyzed. All these tools were discovered through scientific articles. For each tool, besides the presentation of the main characteristics of each one, an analysis of the operators supported by each one of them was made.

3.5.0.1 MuDroid

There are two tools called muDroid, in this study, we only going to present the tool developed by Lin Deng as first author. The tool muDroid is used for Android testing in integration level. It has a mechanism to simulate click events and get screenshots, that are used to determine the killed mutants and calculate the mutation score.

MuDroid is an Android mutation analysis tool that includes 17 Android-specific mutation operators and extends 19 Java traditional method-level mutation operators, 15 Java mutation operators from muJava [91], and 4 deletion operators [92].

MuDroid is able to install compiled mutants and execute tests on both Android emulators and real devices, and store mutation execution results. It was developed to execute from a command line, and every step of Android mutation analysis offers a list of APIs. MuDroid is able to compile the source code and other necessary files to an APK file, install this APK file to an Android emulator or a mobile device, and control an unlimited number of emulators and devices to execute in parallel. It is also compatible with tests developed with JUnit, and other major automated Android app testing frameworks, such as Robotium [55] and Espresso[56].

MuDroid uses two parsers (Java and XML) to recognize, understand, and mutate the source code. It allows the design of more effective tests and the possibility of choosing any subset of operators. Additionally, muDroid provides users with a graphical user interface to observe every mutant, with the mutated code highlighted in colors. The original version of the mutated file is in the left pane, while the mutated versions are in the right pane. The changes are highlighted in both panes. Furthermore, muDroid saves the result of execution into a text (TXT) result file that lists the mutation score of the tests and which tests killed which mutants. The file name of the result includes a timestamp that indicates the exact finish date and time.

3.5.0.2 μDroid

μDroid is a framework for energy-aware mutation testing of Android apps. It implements 50 energy-aware mutation operators and relies on a novel, automatic oracle to determine if a mutant can be killed by a test. μDroid challenges the developers to design tests that are more likely to reveal energy defects. None of the existing automated Android testing tools are able to generate tests to kill many of the mutants produced by μDroid [41].

3.5.0.3 DroidMutator

DroidMutator is a mutation analysis tool designed exclusively for Android applications. It utilizes type-checking to reduce the generation of stillborn mutants, and the scope of each mutation operator can be configured so that it only generates mutants in specific code blocks. Beyond that, DroidMutator implements a total of 32 mutation operators divided into two types, Android-specific mutation operators and Java-specific mutation operators respectively, additionally, external mutation operators can be added by the user. The Java-specific mutation operators handle the primitive features of programming languages. For example, they modify expressions by replacing, adding, and deleting primitive operators. The Android-specific mutation operators handle Android-specific features such as intents, views, and locations [93].

3.5.0.4 Edroid

Edroid is a graphical user-friendly Android mutation tool whose goal is to mutate Android's main components such as activities, services, content providers, and broadcast receivers using the source code of XML files. Edroid applies strong mutation testing, also called traditional mutation testing, with a full generation of source code for mutants. Edroid as a tool allows the user to mutate their Android app using a list of 14 mutation operators, 11 of them target the graphical user interface, while 3 of them target the configuration files.

Edroid consists of an interface, similar to Mujava's interface and design, that allows the user to generate Android mutants. Edroid first requires the user to enter the location of the folder that contains the Android source files. Then, the next step is for the user to select the list of mutation operators to be applied to the Android app. Mutants are generated. Edroid detects whether the type of file is an XML Layout or Android Manifest file and generates the mutants that are selected by the user according to its type. The console shows the type and number of generated mutants. For every mutant that is generated, the mutated file and the rest of the files are also generated with it. The mutated file shows which excerpt of the source code was modified by the mutant.

In order to generate each mutant, Edroid follows a four-step workflow similar to the methodology used by Oliviera *et al.* [94]: (1) Each XML file is read as a text file; (2) a specific keyword or set of keywords are found on the source code; (3) the keywords are replaced one at a time and the part of code being replaced is commented with the name of the mutant; (4) the newly modified version of the file is saved in a folder named after the name of the mutant and its current generation number.

The purpose of Edroid is to reduce the cost of mutating Android applications, which tends to be time-consuming and highly expensive to achieve. Nevertheless, Edroid does not detect equivalent mutants during any part of the mutation process, nor does it allow them to be removed manually through the interface.

3.5.0.5 MDroid+

MDroid+ is a mutation testing framework for Android applications that supports 38 mutation operators, Android and Java specific, automates the process of detecting potential mutant locations and generating mutants, and facilitates the addition of new operators and the maintenance of existing operators through an extensible architecture. MDroid+ statically analyzes a target mobile app, looking for locations where operators can be implemented. For each location, MDroid+ generates a mutant. This process is performed using text or AST manipulation rules specific to each implemented operator. Thus, for each location related to an operator, the text/AST transformation is applied to the specified location in either the code or .xml file. MDroid+ creates a project-level clone of a target and applies a single mutation to a specified location in the cloned project, resulting in one mutant project for each seeded instance of a mutation operator. MDroid+ allows for easy modification/extension of the operator's list, in order to keep pace with rapid evolution.

3.5.0.6 BacterioWeb v.2

BacterioWeb v.2 was created as an improvement to BacterioWeb v.1 allowing users to test Android mobile applications for mutations in a distributed environment. BacterioWeb v.2 is the first mutation testing tool to be made available online. Additionally, BacterioWeb v.2 provides the option of importing mutants so that users can benefit from the strength of this and other mutant generating tools. BacterioWeb v.2 implements mutation techniques that enable testers to conduct mutation testing at a reasonable cost and in a reasonable amount of time, including: (1) Selective Mutation, which allows testers to choose the mutation operators to apply; (2) Parallel execution of mutants; (3) Mutant Schema to speed up mutant packaging; (4) Only Alive method; (5) Mutant Sampling; using a small random selection of mutants from the total set. Testers enter through the Central Web Server (CWS). It enables access to the mobile devices made available by the Device Web Servers as well as the creation and management of projects. The CWS can manage the SUT on the linked devices via the web interface provided by the Devices Web Server (DWS). The CWS stores the entire project on a relational database that the DWSs are also familiar with. In the CWS, mutant generation is carried out. Despite the fact that the tests are executed on the hardware attached to each DWS, the CWS initiates and manages their execution [3].

Table 3.1 summarizes the tools addressed in this section, for an easier visualization.

Table 3.1: Summary Table

Tool Name	**Available**	**Open-source**	**Number of Android specific Mutants**	**Type Of Mutants**	**Year**
muDroid Deng	✗	✗	17	Event-based,Network-related, XML-related, Component Lifecycle,Energy-Related,...	2015-2017
µDroid	✓	✗	50	Energy-aware	2017
Edroid	✗	✗	14	GUI and configuration files	2018

Continued on next page

Table 3.1: Summary Table (Continued)

MDroid+	✓	✓	38	Activities, Intents, GUI, Connectivity, Back-end Server, I/O, Database,...	2018
DroidMutator	✓	✓	32	Intents, Views, and Locations	2020
BacterioWeb v.2	✗	✗	13	SharedPreferences files	2022

3.5.1 Android Mutation Operators

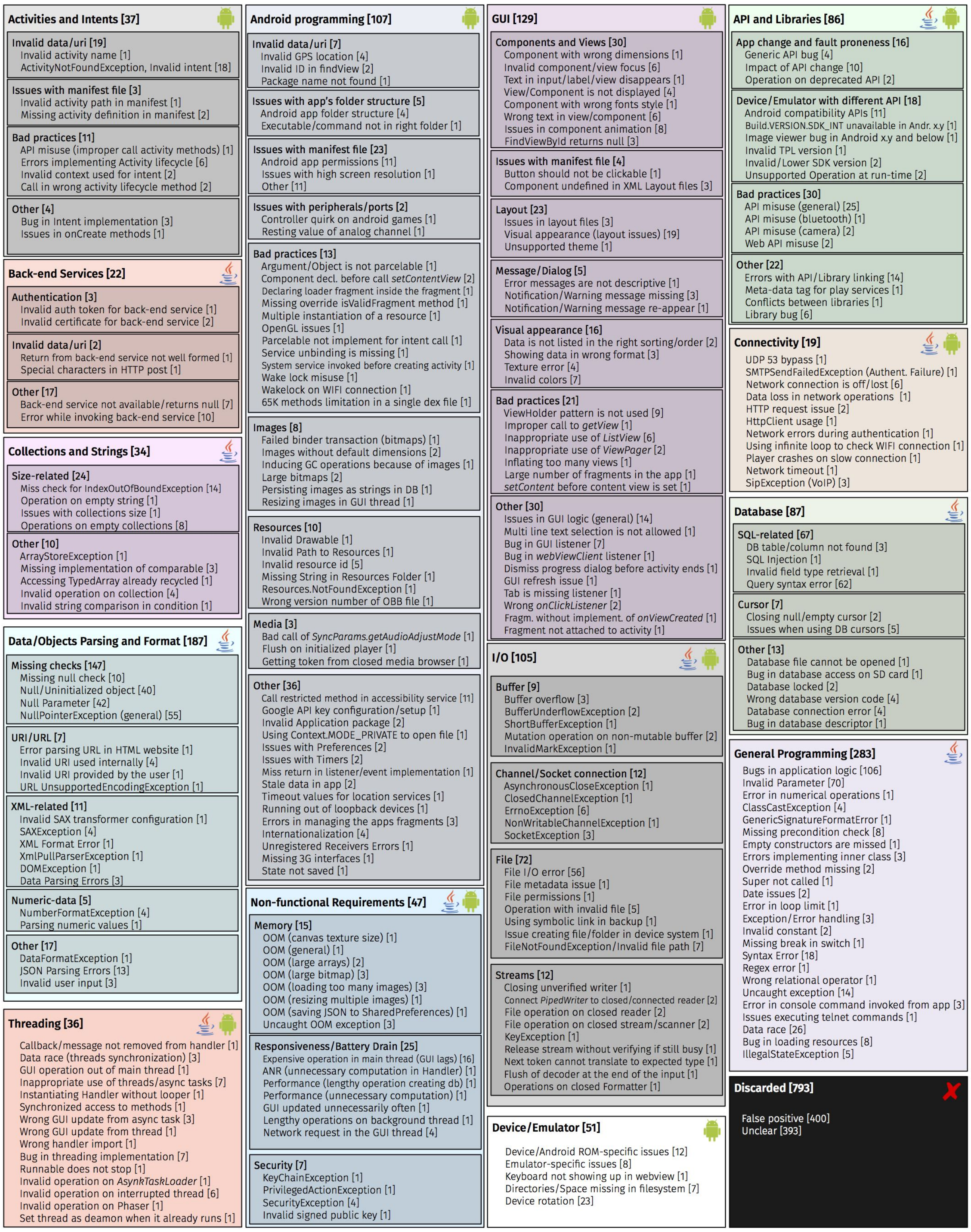
As introduced earlier, mutation operators are the transformation rules used to generate mutants from the original code [27], since mutation operators model simple programming mistakes, they are language dependent. The majority of Android apps are written in Java (despite being now migrating to Kotlin), nevertheless, a study presented by Rene Just Et. Al. (2014) has shown that the traditional Java mutation operators are not sufficient to test all the types of faults present in Android applications and specific mutation operators needed to be implemented since Android apps have different programming structures and are developed, installed, and tested in different ways when compared with traditional Java programs [37]. To guarantee a tight linkage between operators and faults that are likely to arise in an Android project, mutation operators must be specified in accordance with naturally occurring faults. For that, a taxonomy of 262 types of faults grouped in 14 categories, four of which relate to Android-specific faults, five to Java-related faults, and five mixed categories, present in Figure 3.6, was analyzed.

Over the last 5 years, in order to minimize the faults present in Android applications, many operators have been proposed,as enumerated in Section 3.1. In 2020, Henrique Neves da Silva *et al.* [10] realized a mapping study on mutation testing for mobile apps, where 16 primary studies were analyzed, in order to examine all the existing Android operates until date, a total of 138 operators were found. However due to the development of operators is still at an early stage of development, there are no well-defined categories for grouping operators resulting in a lack of consensus among studies. In different studies, the same operator besides having a different name, differs in the category in which is inserted, since this attribution and aggregation is done based on the author's knowledge. Henrique Neves da Silva *et al.* [10] tried to categorize them into 8 distinct categories, among them Traditional, Location, Intent, Graphical User Interface (GUI), and Connectivity, Table 3.2, aiming to provide a comprehensive classification of Android mutation operators. A list of all the existing operators can be seen in [95].

Table 3.2: Types Of Operators , from [10]

Types Of Mutants	**Description**	**Quantity**
Traditional Operators	Introduce modifications related to general bugs found in programming languages (Java)	9
Intent Operators	Introduce changes related to the Android-specific component Intent, used mostly for communication among apps	11
Configuration Operators	Modify attributes or parameters that configure the operation of mobile apps	32
Connectivity Operators	Interfere with the app's communication, like Bluetooth, WiFi, and HTTP requests.	21
GUI Operators	Mutate GUI elements and their event handlers, life cycle, and navigation	47
Location Operators	Inject geolocation related faults (GPS)	11
Persistence Operators	Mutate persistence mechanisms for local storage like files, and databases (SQLite)	7
Sensor Operators	Modify events and instructions related to sensors (e.g., gyroscope, step counter, proximity)	2

It is expected that the implementation in *metamutant* form will not be possible for some mutation operators. As a result, this research is devoted to the implementation of mutation operators capable of producing *metamutants*.



Activities and Intents [37]

- **Invalid data/uri [19]**
  - Invalid activity name [1]
  - ActivityNotFoundException, Invalid intent [18]
- **Issues with manifest file [3]**
  - Invalid activity path in manifest [1]
  - Missing activity definition in manifest [2]
- **Bad practices [11]**
  - API misuse (improper call activity methods) [1]
  - Errors implementing Activity lifecycle [6]
  - Invalid context used for intent [2]
  - Call in wrong activity lifecycle method [2]
- **Other [4]**
  - Bug in Intent implementation [3]
  - Issues in onCreate methods [1]

Back-end Services [22]

- **Authentication [3]**
  - Invalid auth token for back-end service [1]
  - Invalid certificate for back-end service [2]
- **Invalid data/uri [2]**
  - Return from back-end service not well formed [1]
  - Special characters in HTTP post [1]
- **Other [17]**
  - Back-end service not available/returns null [7]
  - Error while invoking back-end service [10]

Collections and Strings [34]

- **Size-related [24]**
  - Miss check for IndexOutOfBoundException [14]
  - Operation on empty string [1]
  - Issues with collections size [1]
  - Operations on empty collections [8]
- **Other [10]**
  - ArrayStoreException [1]
  - Missing implementation of comparable [3]
  - Accessing TypedArray already recycled [1]
  - Invalid operation on collection [4]
  - Invalid string comparison in condition [1]

Data/Objects Parsing and Format [187]

- **Missing checks [147]**
  - Missing null check [10]
  - Null/Uninitialized object [40]
  - Null Parameter [42]
  - NullPointerException (general) [55]
- **URI/URL [7]**
  - Error parsing URL in HTML website [1]
  - Invalid URI used internally [4]
  - Invalid URI provided by the user [1]
  - URL UnsupportedEncodingException [1]
- **XML-related [11]**
  - Invalid SAX transformer configuration [1]
  - SAXException [4]
  - XML Format Error [1]
  - XmlPullParserException [1]
  - DOMException [1]
  - Data Parsing Errors [3]
- **Numeric-data [5]**
  - NumberFormatException [4]
  - Parsing numeric values [1]
- **Other [17]**
  - DataFormatException [1]
  - JSON Parsing Errors [13]
  - Invalid user input [3]

Threading [36]

- Callback/message not removed from handler [1]
- Data race (threads synchronization) [3]
- GUI operation out of main thread [1]
- Inappropriate use of threads/async tasks [7]
- Instantiating Handler without looper [1]
- Synchronized access to methods [1]
- Wrong GUI update from async task [3]
- Wrong GUI update from thread [1]
- Wrong handler import [1]
- Bug in threading implementation [7]
- Runnable does not stop [1]
- Invalid operation on *AsyncTaskLoader* [1]
- Invalid operation on interrupted thread [6]
- Invalid operation on Phaser [1]
- Set thread as deamon when it already runs [1]

Android programming [107]

- **Invalid data/uri [7]**
  - Invalid GPS location [4]
  - Invalid ID in findView [2]
  - Package name not found [1]
- **Issues with app's folder structure [5]**
  - Android app folder structure [4]
  - Executable/command not in right folder [1]
- **Issues with manifest file [23]**
  - Android app permissions [11]
  - Issues with high screen resolution [1]
  - Other [11]
- **Issues with peripherals/ports [2]**
  - Controller quirk on android games [1]
  - Resting value of analog channel [1]
- **Bad practices [13]**
  - Argument/Object is not parcelable [1]
  - Component decl. before call *setContentView* [2]
  - Declaring loader fragment inside the fragment [1]
  - Missing override isValidFragment method [1]
  - Multiple instantiation of a resource [1]
  - OpenGL issues [1]
  - Parcelable not implement for intent call [1]
  - Service unbinding is missing [1]
  - System service invoked before creating activity [1]
  - Wake lock misuse [1]
  - Wakelock on WIFI connection [1]
  - 65K methods limitation in a single dex file [1]
- **Images [8]**
  - Failed binder transaction (bitmaps) [1]
  - Images without default dimensions [2]
  - Inducing GC operations because of images [1]
  - Large bitmaps [2]
  - Persisting images as strings in DB [1]
  - Resizing images in GUI thread [1]
- **Resources [10]**
  - Invalid Drawable [1]
  - Invalid Path to Resources [1]
  - Invalid resource id [5]
  - Missing String in Resources Folder [1]
  - Resources.NotFoundException [1]
  - Wrong version number of OBB file [1]
- **Media [3]**
  - Bad call of *SyncParams.getAudioAdjustMode* [1]
  - Flush on initialized player [1]
  - Getting token from closed media browser [1]
- **Other [36]**
  - Call restricted method in accessibility service [11]
  - Google API key configuration/setup [1]
  - Invalid Application package [2]
  - Using Context.MODE_PRIVATE to open file [1]
  - Issues with Preferences [2]
  - Issues with Timers [2]
  - Miss return in listener/event implementation [1]
  - Stale data in app [2]
  - Timeout values for location services [1]
  - Running out of loopback devices [1]
  - Errors in managing the apps fragments [3]
  - Internationalization [4]
  - Unregistered Receivers Errors [1]
  - Missing 3G interfaces [1]
  - State not saved [1]

Non-functional Requirements [47]

- **Memory [15]**
  - OOM (canvas texture size) [1]
  - OOM (general) [1]
  - OOM (large arrays) [2]
  - OOM (large bitmap) [3]
  - OOM (loading too many images) [3]
  - OOM (resizing multiple images) [1]
  - OOM (saving JSON to SharedPreferences) [1]
  - Uncaught OOM exception [3]
- **Responsiveness/Battery Drain [25]**
  - Expensive operation in main thread (GUI lags) [16]
  - ANR (unnecessary computation in Handler) [1]
  - Performance (lengthy operation creating db) [1]
  - Performance (unnecessary computation) [1]
  - GUI updated unnecessarily often [1]
  - Lengthy operations on background thread [1]
  - Network request in the GUI thread [4]
- **Security [7]**
  - KeyChainException [1]
  - PrivilegedActionException [1]
  - SecurityException [4]
  - Invalid signed public key [1]

GUI [129]

- **Components and Views [30]**
  - Component with wrong dimensions [1]
  - Invalid component/view focus [6]
  - Text in input/label/view disappears [1]
  - View/Component is not displayed [4]
  - Component with wrong fonts style [1]
  - Wrong text in view/component [6]
  - Issues in component animation [8]
  - FindViewById returns null [3]
- **Issues with manifest file [4]**
  - Button should not be clickable [1]
  - Component undefined in XML Layout files [3]
- **Layout [23]**
  - Issues in layout files [3]
  - Visual appearance (layout issues) [19]
  - Unsupported theme [1]
- **Message/Dialog [5]**
  - Error messages are not descriptive [1]
  - Notification/Warning message missing [3]
  - Notification/Warning message re-appear [1]
- **Visual appearance [16]**
  - Data is not listed in the right sorting/order [2]
  - Showing data in wrong format [3]
  - Texture error [4]
  - Invalid colors [7]
- **Bad practices [21]**
  - ViewHolder pattern is not used [9]
  - Improper call to *getView* [1]
  - Inappropriate use of *ListView* [6]
  - Inappropriate use of *ViewPager* [2]
  - Inflating too many views [1]
  - Large number of fragments in the app [1]
  - *setContent* before content view is set [1]
- **Other [30]**
  - Issues in GUI logic (general) [14]
  - Multi line text selection is not allowed [1]
  - Bug in GUI listener [7]
  - Bug in *webViewClient* listener [1]
  - Dismiss progress dialog before activity ends [1]
  - GUI refresh issue [1]
  - Tab is missing listener [1]
  - Wrong *onClickListener* [2]
  - Fragm. without implement. of *onViewCreated* [1]
  - Fragment not attached to activity [1]

I/O [105]

- **Buffer [9]**
  - Buffer overflow [3]
  - BufferUnderflowException [2]
  - ShortBufferException [1]
  - Mutation operation on non-mutable buffer [2]
  - InvalidMarkException [1]
- **Channel/Socket connection [12]**
  - AsynchronousCloseException [1]
  - ClosedChannelException [1]
  - ErrnoException [6]
  - NonWritableChannelException [1]
  - SocketException [3]
- **File [72]**
  - File I/O error [56]
  - File metadata issue [1]
  - File permissions [1]
  - Operation with invalid file [5]
  - Using symbolic link in backup [1]
  - Issue creating file/folder in device system [1]
  - FileNotFoundException/Invalid file path [7]
- **Streams [12]**
  - Closing unverified writer [1]
  - Connect *PipedWriter* to closed/connected reader [2]
  - File operation on closed reader [2]
  - File operation on closed stream/scanner [2]
  - KeyException [1]
  - Release stream without verifying if still busy [1]
  - Next token cannot translate to expected type [1]
  - Flush of decoder at the end of the input [1]
  - Operations on closed Formatter [1]

Device/Emulator [51]

- Device/Android ROM-specific issues [12]
- Emulator-specific issues [8]
- Keyboard not showing up in webview [1]
- Directories/Space missing in filesystem [7]
- Device rotation [23]

API and Libraries [86]

- **App change and fault proneness [16]**
  - Generic API bug [4]
  - Impact of API change [10]
  - Operation on deprecated API [2]
- **Device/Emulator with different API [18]**
  - Android compatibility APIs [11]
  - Build.VERSION.SDK_INT unavailable in Andr. x.y [1]
  - Image viewer bug in Android x.y and below [1]
  - Invalid TPL version [1]
  - Invalid/Lower SDK version [2]
  - Unsupported Operation at run-time [2]
- **Bad practices [30]**
  - API misuse (general) [25]
  - API misuse (bluetooth) [1]
  - API misuse (camera) [2]
  - Web API misuse [2]
- **Other [22]**
  - Errors with API/Library linking [14]
  - Meta-data tag for play services [1]
  - Conflicts between libraries [1]
  - Library bug [6]

Connectivity [19]

- UDP 53 bypass [1]
- SMTPSendFailedException (Authent. Failure) [1]
- Network connection is off/lost [6]
- Data loss in network operations [1]
- HTTP request issue [2]
- HttpClient usage [1]
- Network errors during authentication [1]
- Using infinite loop to check WIFI connection [1]
- Player crashes on slow connection [1]
- Network timeout [1]
- SipException (VoIP) [3]

Database [87]

- **SQL-related [67]**
  - DB table/column not found [3]
  - SQL Injection [1]
  - Invalid field type retrieval [1]
  - Query syntax error [62]
- **Cursor [7]**
  - Closing null/empty cursor [2]
  - Issues when using DB cursors [5]
- **Other [13]**
  - Database file cannot be opened [1]
  - Bug in database access on SD card [1]
  - Database locked [2]
  - Wrong database version code [4]
  - Database connection error [4]
  - Bug in database descriptor [1]

General Programming [283]

- Bugs in application logic [106]
- Invalid Parameter [70]
- Error in numerical operations [1]
- ClassCastException [4]
- GenericSignatureFormatError [1]
- Missing precondition check [8]
- Empty constructors are missed [1]
- Errors implementing inner class [3]
- Override method missing [2]
- Super not called [1]
- Date issues [2]
- Error in loop limit [1]
- Exception/Error handling [3]
- Invalid constant [2]
- Missing break in switch [1]
- Syntax Error [18]
- Regex error [1]
- Wrong relational operator [1]
- Uncaught exception [14]
- Error in console command invoked from app [3]
- Issues executing telnet commands [1]
- Data race [26]
- Bug in loading resources [8]
- IllegalStateException [5]

Discarded [793]

- False positive [400]
- Unclear [393]

Figure 3.6: Fault Taxonomy of Android, image from [7]

Chapter 4

Metamutant Approach

This section will present the approach selected for implementing the Mutant Schemata Generation (MSG) method for Android applications.

The generation of mutants requires defined mutation operators. Therefore, a comprehensive analysis was executed that focused on identifying as many operators as possible. The analysis encompassed the study of numerous aspects of the code, including common programming errors, coding conventions, and industry best practices. In this study, we found that certain patterns and scenarios are associated with an increase in faults. Consequently, implementing mutation operators needs cautious consideration of Abstract Syntax Tree (AST).

AST is a data structure used to represent the structure of a program. It is crucial in mutation testing as it provides a representation of the code syntax and semantics, which allows accurate analysis of code and mutation generation. Regarding mutant generation, the Abstract Syntax Tree (AST) provides a hierarchical representation of the code, denoting the relationships and associations between distinct code elements. Through AST, mutation operators can be selectively applied at specified points, enabling the creation of mutants with targeted modifications. Another feature AST supports is the possibility of comparing data between mutants and the original code, facilitating the detection of modified code elements, for example, altered expressions, removed or added statements, or changes in control flow. It also helps determine the impact of mutations. In order to generate the AST, we used the existing tool *Kadabra* [96].

4.1 Kadabra Tool

Kadabra is a Java-to-Java compilation tool that uses LARA framework for code instrumentation and transformations. It utilizes an open-source Java-to-Java library developed by the Spirals research team, Spoon [97], to perform the operations.

Spoon's AST-based approach enables *Kadabra* to analyze the structure and semantics of Java code at a detailed level and to perform several transformations, such as code instrumentation for profiling, debugging, or injecting additional behaviour.

LARA increases the abilities of Spoon by accepting as input scripts written in Javascript, that describe the analyses and transformations to be applied to the Java code. This approach enhances code modularity, maintainability, and reusability.

Initially LARA framework used a domain-specific language (DSL),a programming language with a higher level of abstraction optimized for a specific class of problems, that only supported queries and direct code insertions. However, over time, the framework progressed to support arbitrary JavaScript code embedded within the DSL. Currently, the LARA framework accepts pure JavaScript and is even in the process of transitioning to TypeScript.

Moreover, in the LARA framework, every element of the AST is represented as a *joinpoint*. The term "**joinpoint**" is commonly used in aspect-oriented programming (AOP) to refer to specific points in the execution of a program where aspects can be applied [98]. Aspects are cross-cutting concerns that can be added to an application to provide additional functionality without modifying the core code. *Joinpoints* aim to incorporate all the different elements that can be found in the AST and serve as points of interest within the code where mutations can occur. These Mutations can take the form of replacement, removal, or insertion before and/or after the identified *joinpoint*. In [99], are represented the available *joinpoints* in *Kadabra*, and their class hierarchy relationships.

LARA also provides several tools and APIs that support the development of LARA compilers for additional programming languages. An example is Weaver Generator, which allows the creation of LARA compilers tailored to specific languages. In this work, several key APIs were applied, including the `lara.Io` API that provides utility methods for handling input/output operations on files, simplifying file-related tasks within the LARA compilers, and additionally the `weaver.Query` API that enables selection of specific *joinpoints* from the AST.

Following the usage of the *Kadabra* tool, it is possible to analyse the structure and components of the generated AST Tree. The AST results from the examination of the nodes and branches obtained from the source code of a program. Thus, with *Kadabra*, it is possible to identify specific elements or patterns in the code that can be mutated, such as variables, statements, expressions, control flow structures, function calls, and more.

With the identification of these elements, mutation operators that mimic those variations may be introduced. Implementing mutation operators leads to the production of mutants that exhibit different behaviours compared to the original program.

The process of constructing mutant operators generally involves the following steps:

- **Identifying mutation points:** Based on the analysis of the code structure and components, mutation points are identified as specific elements or patterns that can be mutated. Variables, statements, expressions, control flow structures, and other code constructs can be selected.
- **Defining mutation strategies:** For each mutation point, different strategies or variations can be conceived, which determine the types of modifications implemented to the mutation points. For instance, for a variable mutation point, strategies such as changing the variable's value, data type, or scope could be applied.
- **Designing the operator logic:** Once the mutation points and strategies are defined, the operator logic is delineated. This encompass determining how the operator will identify and transform the code to introduce the desired variations. As stated before the logic should take into account the AST structure and properties to perform the appropriate modifications.
- **Implementing the operator:** The designed operator logic is then carried out by writing the necessary code. This requires traversing the AST, locating the mutation points, and applying the defined strategies to create the desired mutants.
- **Testing and evaluating the operators:** Sequentially, the implemented operators are tested by applying them to sample code snippets or test cases. The generated mutants are then assessed to verify that the desired variations have been introduced. The effectiveness of the operators in generating meaningful mutants is estimated, and adjustments or refinements are made if needed.

4.2 Operators Architecture

Considering that every mutation operator follows a similar set of steps, the process of constructing mutation operators can be generalized. Therefore, a reusable architecture was used to implement mutation operators. Each custom mutation operator extends the Mutator class from LARA framework. The Mutator class offers a standardized interface that contains methods and properties which facilitate the mutation process as it establishes the basic structure and behaviour of a mutation operator. Supplementary methods or overridden versions may be put into practice depending on the specific requirements and implementation of the operator.

Presented below are some of the most commonly used methods in the Mutator class:

- ***constructor:*** This method is primarily used to initialize the mutation operator, it accepts parameters and performs any initializations required.

- ***addJp(joinpoint):*** The focal point of this method is to identify and add the suitable *joinpoints* in the code where the mutation should be applied. It recognises elements and patterns meeting the mutation criteria, returning true if the *joinpoint* is added and false otherwise.
- ***getMutationPoint():*** The operation of this method is to return the next mutation point to be mutated and it is based on the current index or position in the mutation points list.
- ***_mutatePrivate():*** This private method applies the mutation to identified *joinpoint* of the code. It, also, defines how the mutation will modify the code, by replacing a statement, modifying an expression, or introducing a new code element. The specifics will depend on the nature of the mutation being implemented.
- ***hasMutations():*** This method assesses if there are any remaining mutations to be performed, returning true if more mutations need to be applied, and false otherwise.
- ***_restorePrivate():*** This private method is used to restore the code to its original state after applying a mutation, by undoing the introduced variation. The implementation of this method will depend on the mutation operation performed in _mutatePrivate().

Additionally to the methods mentioned above, it was implemented the *toString()* which returns a string that offers important information about the operator, such as the name of the current operator and the current mutation point as well as the applied mutation.

Another method used was the *toJson()*, whose purpose is to return a JSON object that encapsulates the mutation operator. One example is the operator name and the input values, and also if it is an Android-specific operator.

Lastly, the method *isAndroidSpecific()* was implemented, which returns true if the operator belongs to the android specific operators, and false if it does not.

This section demonstrates a specific example of a mutation operator. The operator chosen was the "NullIntentOperator", which targets instances of "Intent" in the code. Firstly, the mutator class initiates by importing the necessary classes and packages from the external libraries common to every operator. Then, the class is declared and extended the Mutator Class inherits its properties and methods. These include *addJp(), hasMutations(), getMutationPoint(), _mutatePrivate()*, and *_restorePrivate()*, which are overridden to provide customized behaviour specific to the NullIntentOperator.

Inside the class, it is defined a *constructor* that initiates several instance variables: *mutationPoints*, an empty array used to store mutation points; *currentIndex*, used as an array index for *mutationPoints* array, it helps to determine if there are any remaining mutation points in the array.*mutationPoint* and *previousValue* represent the value of the added mutation point, and the previous value represents the value before the mutation has been applied.

Afterwards, the *isAndroidSpecific()* method indicates whether the operator is specific to Android.

```
laraImport("lara.mutation.Mutator");
laraImport("kadabra.KadabraNodes");
laraImport("weaver.WeaverJps");
laraImport("weaver.Weaver");

class NullIntentOperator extends Mutator {
  constructor() {
    super("NullIntentOperator");

    this.mutationPoints = [];
    this.currentIndex = 0;
    this.mutationPoint = undefined;
    this.previousValue = undefined;
  }
  isAndroidSpecific() {
    return true;
  }
```

Another method is the *addJp()*. It receives a *joinpoint* as a parameter corresponding to a node in the *AST tree*. This method declares the filters required to select the specific element. Firstly, it is evaluated if the *joinpoint* received and the *joinpoint* type are undefined, as this specific mutation is intended to capture nodes with Intent value as type.

As the main purpose is to catch intent instantiations, the AST was examined, and a pattern was observed. The pattern is that, in order to be an instantiation, the first child of the *joinpoint* has to have the name "<init>" and the type "Executable". Thus, the first child must not be undefined. The other filters were specified in order to avoid specific cases encountered.

```
  addJp(joinpoint) {
   if (joinpoint === undefined) {
   return false;
}

if (joinpoint.type === undefined) {
   return false;
}

if (joinpoint.type == "Intent" &&
      joinpoint.children[0]!=undefined &&
      joinpoint.children[0].name != undefined &&
      joinpoint.parent!=undefined &&
      joinpoint.children[0].name === "<init>" &&
      joinpoint.children[0].type === "Executable" &&
      joinpoint.type != "Package"
    ) {
      this.mutationPoints.push(joinpoint);
    return true;
    }
    return false;
}
```

The *hasMutations()* method as the name suggests checks if there are any remaining mutations to be performed, by analyzing the current index in the mutationPoints array and comparing it to its length.

```
  hasMutations() {
    return this.currentIndex < this.mutationPoints.length;
  }
```

The *getMutationPoint()* method verifies if the current code is already mutated, if not returns the next mutation point to be mutated.

```
  getMutationPoint() {
    if (this.isMutated) {
      return this.mutationPoint;
    } else {
      if (this.currentIndex < this.mutationPoints.length) {
        return this.mutationPoints[this.currentIndex];
      } else {
        return undefined;
      }
    }
  }
```

The *mutatePrivate()* method is where the mutation is applied in this case we wanted to replace the instantiation with a null value using the *insertReplace()* method provided by *Kadabra*.

```
_mutatePrivate() {
  this.mutationPoint = this.mutationPoints[this.currentIndex];

  this.previousValue = this.mutationPoint;
  this.mutationPoint = this.mutationPoint.insertReplace("null");

  println("/*--------------------------------------*/");
  println(
    "Mutating operator n." +
      this.currentIndex +
      ": " +
      this.previousValue +
      " to " +
      this.mutationPoint
  );
  println("/*--------------------------------------*/");

  this.currentIndex++;
}
```

The *restorePrivate()* method is used to convert the mutated code to the original. It undoes the mutation introduced in the *mutatePrivate()* method by replacing the mutation point value for the one before applying the mutation.

```
_restorePrivate() {
  this.mutationPoint = this.mutationPoint.insertReplace(this.previousValue);
  this.previousValue = undefined;
  this.mutationPoint = undefined;
}
```

Both *toString()* and *toJson()* methods are used to get more details about the mutation operator.

```
toString() {
  return `Null Intent Operator from ${this.previousValue} to ${this.
      mutationPoint}, current mutation points ${this.mutationPoints},
      current mutation point ${this.mutationPoint} and previous value ${this.
          previousValue}`;
}
toJson() {
    return {
      mutationOperatorArgumentsList: [],
      operator: this.name,
      isAndroidSpecific: this.isAndroidSpecific(),
    };
  }
}
```

After implementing a mutation operator, testing is a critical step to ensure its correctness and effectiveness. To verify this, a set of test cases is made to cover the numerous scenarios and edge cases. These test cases are usually source code samples where the mutation operator is expected to be applied. Each mutant should exhibit the specific mutation introduced by the operator, in the case of the NullIntentOperator, the "Intent" expressions should be replaced with the value "null" in the mutants.

Below we demonstrate an example of the *metamutant* generated when the operator NullIntentOperator was applied to the following snippet code.

```
    @NonNull
    public static Intent getIntentToOpenFeed(@NonNull Context context, long
        feedId) {
        Intent intent = new Intent(context.getApplicationContext(), MainActivity.
            class);
        intent.putExtra(MainActivity.EXTRA_FEED_ID, feedId);
        intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
        return intent;
    }
```

For each instance of the Intent component, the resulting *mutant* replaces it with null.

```
    @NonNull
    public static Intent getIntentToOpenFeed(@NonNull Context context, long
        feedId) {
        Intent intent;
        if (getMUID().equals("
            NullIntentOperator_MainActivity_id_9e026db6_f10c_4a8b_9091_2fd746397b7f
            ")){
            intent = null;
        }else{
        intent = new Intent(context.getApplicationContext(), MainActivity.class);
        }
        intent.putExtra(MainActivity.EXTRA_FEED_ID, feedId);
        intent.addFlags(FLAG_ACTIVITY_CLEAR_TOP);
        return intent;
    }
```

By examining the generated mutants, it is possible to determine whether the mutation operator was implemented correctly. If the mutants do not reflect the expected mutations, it indicates that the operator may not have been properly implemented and further adjustments are required. It is crucial to emphasize that each mutation operator will have its own unique implementation of the *addJp(), _mutatePrivate()*, and *_restorePrivate()* methods. The implementation will depend on the mutation, the type of *joinpoint* being targeted, and the desired modification in the code.

4.3 Mutant generation process

This section discusses the mutant generation process. As stated before the mutant schemata strategy implements all mutants at once instead of compiling each mutant separately. Consequently, all mutants are incorporated into the codebase, with each mutant enclosed within a conditional statement. These conditional statements allow the activation of individual mutants during *runtime*, which eliminates the need for maintaining separate code bases for each mutant. Additionally, only a single compilation is necessary to produce the final executable. To streamline this process, the following method was implemented.

It is important to mention that the responsibility of the implementation of the *runTreeAndApplyMetaMutant()* was not our responsibility. However, as it has a crucial significance in this work, it will be described briefly. For a more detailed version, check [100].

The *runTreeAndApplyMetaMutant()* method starts by declaring a variable called mutanList, that will store all the mutants generated. By using the *Query.root().descendants* method it iterates over each node of the AST in order to perform a set of operations. Firstly, for each mutation operator previously selected to mutatorList, another variable, it tries to add the actual joipoint using the *addJp()* method. If that *joinpoint* has the required characteristics, the number of mutationPoints increases. mutationPoints variable is used to understand the necessity of if else statement based on the number of mutations. Since the mutation points were already added, it is necessary to mutate them. Each operator was iterated again in order to check if mutations were still present through the method *hasMutations()*. Through the method *mutate()*, the mutation can be applied. Then, depending on the number of mutations added, it inserts the mutated code within an "if" or "else if" statement. After each mutation, the original code is restored using the *restore()* method. This ensures that subsequent mutations are applied correctly without the interference from previous mutations. In the end, the mutantList array contains all the generated mutants exhibiting the specific mutation introduced by the corresponding mutation operator. These mutants are them used for further testing and evaluation to validate the correctness and effectiveness of the implemented mutation operators. As observed in this method, all the methods defined within the operator are utilized.

During the process of generating the *metamutant*, we observed that certain code statements, particularly declarations and instantiations, were resulting in invalid code, as shown below.

```
@NonNull
public static Intent getIntentToOpenFeed(@NonNull Context context, long
    feedId) {

    if (getMUID().equals("
        NullIntentOperator_MainActivity_id_9e026db6_f10c_4a8b_9091_2fd746397b7f
        ")){
     Intent  intent = null;
    }else{
     Intent intent = new Intent(context.getApplicationContext(), MainActivity
         .class);
    }
    intent.putExtra(MainActivity.EXTRA_FEED_ID, feedId);
    intent.addFlags(FLAG_ACTIVITY_CLEAR_TOP);
    return intent;
}
```

To address this issue, an additional method, *changeVarDeclarations()* was implemented to standardize the code prior to applying the mutation operators.

The purpose of the code standardization method was to ensure that the code structure and syntax adhered to the expected format, thus minimizing the occurrence of syntax errors or inconsistencies. This approach improved the effectiveness of the mutation operators by increasing the number of valid mutants generated.

```
function changeVarDeclarations() {
  for (var jp of Query.root().descendants) {
    if (
      jp.instanceOf("localVariable") && !jp.toString().includes("final") && jp.
          numChildren >= 2 && !(jp.parent.type == "for") &&
      !jp.parent.instanceOf("try")) {
```

```
   // Create write reference to local variable
   let localVar = KadabraNodes.var(jp, true);
   // Get initialization, and remove it from declaration
   const varLhs = jp.init;
   jp.init = undefined;
   // Create assignment
   if (varLhs != undefined && localVar != undefined) {
    const varAssign = KadabraNodes.assignment(localVar, varLhs); // Add assignment after initialization jp.insertAfter(varAssign);
   }
  }
 }
}
```

By applying the *changeVarDeclarations()* function, variable declarations in the code are transformed into separate variable assignments, an example is presented below. This allowed for more fine-grained control and manipulation of variables during the mutation process.

Before

```
Intent intent= null;
```

After

```
Intent intent;
intent = null;
```

4.4 Mutant execution process

Once mutants have been generated, mutants may be compiled and tests executed. As demonstrated earlier in the previous section, before each mutation point, the following line of code is added.

```
if(getMUID().equals("' +  mutantId +'")){\n
```

The *getMUID()* is a function that is added to every class that has an Android specific mutant. It manages the control of the mutant execution through the use of the "MUID" system property. It retrieves its value when accessed by the applications at runtime. This value is then compared to the unique identifier mutantId If the result of the conditional is true this means that that specific mutant is the one that should be executed and so the code inside the curly braces follows. David Mata was also in charge of this function, so read [100] for more details.

The various values of the "MUID" system property enable the selective activation of specified mutations during *runtime* without the need for extra compilations or different code bases. Using the "MUID" system property provides a straightforward way to test and evaluate certain mutations.

Chapter 5

Mutation Operators

Although some mutation operators have already been implemented as *metamutants* in existing literature [3], the number of available (meta)operators is limited. With this work, we increase the number of mutation operators that may be implemented as *metamutants* using the approach outlined in the preceding section.

Firstly, the existing mutation operator resultants of the work developed in [101] were analyzed and checked if they were working properly. In chapter 4, it was mentioned that the framework utilized a DSL that only supported queries and direct code insertions. The operators were initially written in LARA language , as part of the current work, was the task of modernizing the existing code by migrating it from the LARA DSL to JavaScript. Consequently, the mutation operators had to be rewritten in JavaScript to align with the new language chosen for the framework. Since most of the operators had significant flaws, the initial step was to address and correct them.

As the majority of the implemented operators implemented by [101] were not specific to Android, and the goal of the study was on Android-specific operators, it was decided that the *dataset* utilized would be derived from [10], available at [95], which encompassed 110 Android-specific operators obtained from an analysis of 16 studies, that were also carefully examined. The *dataset* is organized by studies, and for each study, the referenced operators and their respective categories are listed.

It was brought to light that some studies implemented the same operators, resulting in repeated operators in the dataset. The duplicated operators were eliminated from the dataset to make the selection process more efficient. **This step led to a final collection of 98 distinct operators.**

To gain insights into the most frequently encountered Android components and elements, it was conducted an in-depth analysis of the source code of multiple Android applications available on GitHub [102].

Out of the eight categories that encompassed the 98 mutants, three specific categories were chosen: "Intent," "GUI," and "Traditional." The main reason for selecting these categories was the elements they modify:

- The **Intent** category encompasses changes related to the specific Android component Intent, which is widely used in Android applications to enable communication and interaction between different Android components such as Activities, Services, and Broadcast receivers.

- The **GUI** category includes changes that modify GUI elements, as the name suggests, such as XML files, Activities, and event handlers.

- The **Traditional** category introduces changes based on common errors in the Java programming language.

Initially, we began by implementing Java-specific operators, followed by Intent operators, and finally, GUI operators. During the process, it was evident that some selected operators could not be applied to *metamutants*, including the operators that apply changes to XML files. Due to the nature of XML files, they do not support the inclusion of executable code such as conditional statements (e.g., if statements) or other programming constructs. Therefore, the generation of *metamutants* based on if structures and conditional logic is not practical within the realm of XML files. A new approach to tackle this issue was devised. However, it should be noted that this process can only be applied to XML layout files. As a consequence, it was not possible to implement any operators related to the manifest.xml file.

Another limitation found was related to the operators that add or remove the "implements" statement in Java files, like Not Serializable Operator. The reason is that Java files cannot have the same class declaration twice within a single file, and it is not possible to apply this operator without causing an error.

Based on these limitations, a total of 53 mutation operators were implemented of which 14 are general, listed in Table 5.1, 14 are java specific listed in Table 5.2 and 25 Android specific listed in Table 5.3.

Table 5.1: List of Operators - General

Category	**Operators**
General Specific	Arithmetic Operator
	Bitwise Operator
	Comparison Operator
	Logical Operator
	Assignment Operator
	Arithmetic Deletion Operator
	Bitwise Deletion Operator
	Comparison Deletion Operator
	Logical Deletion Operator
	Assignment Deletion Operator
	Constant Operator
	Unary Operators
	Unary Logical Negation Operator
	Unary Deletion Operators

Table 5.2: List of Operators - Java Specific

Category	**Operators**
Java Specific	Constructor Call
	Remove Conditional
	Non Void Call
	Nullify Input Variable
	Nullify Return Value
	Return Value
	Invalid Date
	Invalid Method Call Argument

Continued on next page

Table 5.2: List of Operators - Java Specific (Continued)

	Null Method Call Argument Not Serializable Fail On Null String Argument Replacement String Call Replacement Conditional Expression Replacement

Table 5.3: List of Operators - Android Specific

Category	**Operators**
Android Specific	Buggy GUI Listener Lengthy GUI Listener Lengthy GUI Creation Find View By Id Returns Null View Component Not Visible Invalid View Focus Invalid ID FindView Null Intent Random Action Intent Definition Intent Target Replacement Invalid Key Intent Null Value Intent PutExtra Intent Payload Replacement XML Edit TextWidget Invisible XML ViewGroup Widget Invisible XML Button Widget Invisible XML EditText Widget Deletion

Continued on next page

Table 5.3: List of Operators - Android Specific (Continued)

	XML Button Widget Deletion
	XML TextView Widget Deletion
	XML Invalid Color
	XML Button Widget Change Appearance
	XML EditTex tWidget Change Appearance
	XML ViewGroup Widget Change Type
	Null Bluetooth Adapter
	Null GPS Location

Only 7 of the 54 were previously implemented using *metamutant* approach, specifically Arithmetic Operator, Logical Operator,Bitwise Operator, Unary Operator, Invalid Method Call Argument,Intent Target Replacement and Intent Payload Replacement in [3].

Below, we present a more detailed description of the implemented mutation operators.

5.1 General Operators

General operators define a group of mutation operators that are not exclusive to any specific language characteristics, thus, can be used in various programming languages. The main focus of this type of operator is on fundamental elements of programming.

5.1.1 Binary Operators

"Binary operators represent an operation upon two operands of the same type, producing a result of the same type as the operands" [103]. Hence, two input values are required when applying these operators. The first represents the operator to be mutated, and the second input value refers to the operator that will replace the first input.

5.1.1.1 Arithmetic Operator

Arithmetic operators, an example being $+, -, /, *$, and %, are binary operators frequently used to perform mathematical operations on numerical values. If applied as mutation operators, they can modify the arithmetic operations in the code and alter the behaviour of the program by replacing one arithmetic operator with another.

5.1.1.2 Bitwise Operator

Bitwise Operators, such as AND (&), OR (|), left shift (<<), and right shift (>>), are also binary operators commonly used to manipulate the bit value (0 or 1) of a variable. As mutation operators, they replace one bitwise operator with another, thus altering the behaviour of the program.They differ from arithmetic in that they receive and return Boolean values.

5.1.1.3 Comparison Operator

Comparison Operators including $==, !=, >, <, >=$, and the $<=$, are also binary operators frequently used to compare values and determine the relationship between them. If applied as mutation operators, they modify the comparison operations in the code and alter the program's behaviour by replacing one comparison operator with another.

5.1.1.4 Logical Operator

Logical Operators, such as && and ||, are also binary operators commonly used to perform logical operations on boolean values or expressions. When applied as mutation operators, they modify the logical operations in the code by replacing one logical operator with another, thus altering the behaviour of the program.

5.1.1.5 Assignment Operator

Assignment operators, as $=, +=, -=, /=, *=$, and $\% =$, are binary operators commonly used for compound assignments. When used as mutation operators, they alter the program's behaviour by replacing one assignment operator with another.

5.1.2 Binary Operator Deletion Mutator

This specific type of operator focuses on the removal of the selected operator. Consequently, it generates two sub-mutators that individually mutate the operation's first and second operands [101; 104]. Depending on the type of operator selected they can be classified as:

- **Arithmetic Deletion Operator**, when the target is one of the arithmetic operators
- **Bitwise Deletion Operator**, when the target is one of the bitwise operators
- **Comparison Deletion Operator**, when the target is one of the comparison operators
- **Logical Deletion Operator**, when the target is one of the logical operators

- **Assignment Deletion Operator**, when the target is one of the assignment operators

To better understand this operator an example will be given. When applying the Arithmetic Deletion Operator in the expression $a = b + c$, the arithmetic operator $+$ is removed and will originate two mutations, $a = b$, and $b = c$ [101].

5.1.3 Constant Operator

Constant operators work differently in comparison to the one described in [101]. This operator searches for inline, constant, or assignment variables and modifies their value by replacing it with the received input value. Typically, constants are replaced with the following values, 1, $0, -1$.

5.1.4 Unary Operators

A Unary operator "represents an operation on a single operand that produces a result of the same type as its operand" [105].

Examples of unary operators include right and left-hand increments or decrements, as well as, unary minus operators and unary plus operators $(++\_, --\_, \_++, \_--, -\_, +\_)$. This implementation permits modification of unary operations within the code because it specifies as input values both unary operators, the one being mutated and the unary operator to be used.

5.1.5 Unary Logical Negation operator

The functionality of this unary operator involves the addition of the logical negation operator $(!)$ in the conditions and boolean expressions. When applied, it negates the entire expression, effectively flipping its truth value.

5.1.6 Unary Deletion Operators

This operator performs the opposite action compared to the one mentioned above. This operator searches for the logical negation operator $(!)$ within the expression and removes it, flipping its truth value too.

5.2 Java Specific

The following mutation operators are specific to the Java language. They are tailored to manipulate Java-specific features and constructs. To enhance understanding, an example will be provided for each of the operators mentioned. These examples are primarily sourced from established mutation operator tools such as Pit [104], Mdroid+ [7], and DroidMutator [106].

5.2.1 Constructor Call

The purpose of a constructor is to set initial values to the instance variables of a class or perform necessary setup operations for proper object initialization [107]. In this context, the Constructor Call operator specifically targets constructor calls and replaces them with a null value. This means that the object creation and initialization are bypassed. Thus, it may result in runtime errors such as NullPointerExceptions or unexpected behaviour when the code does not cover this scenario.

Before

```
Object o = new Object();
```

After

```
Object o = null;
```

5.2.2 Remove Conditional

This operator modifies the behaviour of conditional statements by effectively removing the condition with the boolean value true. Therefore, the mutated conditional statement will always be executed, regardless of its actual value.

Before

```
if (a == b) {
  // do something
}
```

After

```
if (true) {
  // do something
}
```

5.2.3 Non Void Call

The Non Void Call operator is designed to eliminate method calls to non void methods. The return value of the method is substituted with the Java Default Value for the corresponding type. Table 5.4 demonstrates the corresponding default value for each type.

Before

```java
public int getValue() {
  return 5;
}

public void foo() {
  int i =  getValue();
  // something using i
}
```

After

```java
public int getValue() {
  return 5;
}

public void foo() {
  int i = 0;
  // something using i
}
```

Table 5.4: Java Default Values

Data Type	Default Value
boolean	false
int	0
byte	0
short	0
long	0
float	0.0
double	0.0
char	'\u0000'
String	'\u0000'
Object	null

5.2.4 Nullify Input Variable

Nullify Input Variable is specifically designed to test user-defined methods for proper handling of null inputs. It achieves this by replacing the parameters of a function with null values. This allows the evaluation of whether the method implementation adequately handles null scenarios or not. Here's an example snippet to illustrate its usage:

- **Before**

```java
public boolean exampleFunction(String test1, String test2) {
   return test1.equals(test2);
}
public void testFunction() {
    boolean i = exampleFunction("Test1", "Test2");
}
```

- **After**

```
public boolean exampleFunction(String test1, String test2) {
   return test1.equals(test2);
}
public void testFunction() {
    boolean i = exampleFunction(null, "Test2");
}
```

5.2.5 Nullify Return Value

The Nullify Return Value operator functions similarly to the previous one, by replacing a variable or constant with the value null. However, in this case, the focus is on modifying method return values to be null. Its primary objective is to evaluate how the code handles null return values.

- **Before**

```
public String exampleFunction(String test) {
   return test;
}
public void testFunction() {
    String i = exampleFunction("Test");
}
```

- **After**

```
public String exampleFunction(String test) {
   return null;
}
public void testFunction() {
    String i = exampleFunction("Test");
}
```

5.2.6 Return Value Operator

This operator also focuses on replacing the return value of methods, however, this one operates in specific return types. For methods returning int, short, long, char, float, or double, the operator replaces the return value with 0. For methods returning a boolean, the operator replaces the return value with true.

- **Before**

```
public boolean exampleFunction(String test) {
   return test == null;
}
public void testFunction() {
   boolean i = exampleFunction("Test");
}
```

- **After**

```
public boolean exampleFunction(String test) {
   return true;
}
public void testFunction() {
    boolean i = exampleFunction("Test");
}
```

5.2.7 Invalid Date

This operator mutates a Date object by setting a random timestamp value to it. The main goal is to understand the behaviour of this portion of the code. In the example, the value 12345678910L represents Sat May 23 21:21:18 GMT 1970, in milliseconds.

- **Before**

```
Date stdDate = new Date(year, month, date);
```

- **After**

```
Date stdDate = new Date(12345678910L);
```

5.2.8 Invalid Method Call Argument

This mutation operator introduces variation in the arguments passed to method calls with unexpected or invalid inputs. Replacing the original argument type with a different input, as shown in the example, allows the assessment of whether the code can appropriately handle the data or not.

- **Before**

```java
int transaction = getTransactionValue();
setValue(transaction);
```

- **After**

```java
int transaction = getTransactionValue();
setValue(-1);
```

5.2.9 Null Method Call Argument

The purpose of this mutation operator is similar to the previous one. The operator verifies the capability of the code to appropriately handle null references and to guarantee that the suitable null-checking mechanisms are in place, by replacing the original argument type with a null value.

- **Before**

```java
 String argument = getArgument();
 methodCall(argument);
```

- **After**

```java
 String argument = getArgument();
 methodCall(null);
```

5.2.10 Not Serializable

As previously mentioned, the Not Serializable operator cannot be implemented in *metamutant*, and for a better understanding, an example is given. In this case, the operator changes the behaviour of the class in terms of its serializability by removing the "*implements Serializable*" declaration from the class.

- **Before**

```java
public class Book implements Serializable {
    private int mData;
  ...
}
```

- **After**

```java
public class Book {
    private int mData;
  ...
}
```

5.2.11 Fail On Null

This operator's purpose is to recognize potential *NullPointerExceptions*, by means of the insertion of a "fail on null" statement before each object reference in the code. This occurs in Java when attempting to access or manipulate an object that is null, meaning it does not refer to any valid object instance. Figure 5.1 represents our implementation of failOnNull method.

- **Before**

```java
List<ResourceType> res = new LinkedList<>();
List<Member> members = collection.getMembers();
for (WebDavResource member : members){
res.add(newResource(member.getName(), member.getETag()));
}
return res.toArray(new Resource[0]);
```

- **After**

```
List<ResourceType> res = new LinkedList<>();
List<Member> members = collection.getMembers();
failOnNull(members);
for (WebDavResource member : members){
res.add(newResource(member.getName(), member.getETag()));
}
return res.toArray(new Resource[0]);
```

```
void failOnNull(Object object) {
    if (object == null) {
        throw new NullPointerException("Fail_on_null");
    }
}
```

Listing 5.1: Fail on Null Implementation

5.2.12 String Argument Replacement

The String Argument Replacement operator, used to test how the code handles empty values, replaces the value of a string argument with an empty value.

Before

```
getValueByName("Tom");
```

After

```
getValueByName("");
```

5.2.13 String Call Replacement

String objects in Java provide several methods for String manipulation, such as: *length(), charAt(), substring(), substring(), startsWith(), endsWith(), toUpperCase(), toLowerCase().*

The string Call Replacement operator replaces the string object call method by invoking a different String call method, as demonstrated below.

Before

```java
urlStr.startWith("http");
```

After

```java
urlStr.endwith("http");
```

5.2.14 Conditional Expression Replacement

The Conditional Expression Replacement operator, also known as the Ternary Operator Replacement, modifies the ternary operator expressions in the code. The ternary operator is a shorthand way of writing an if-else statement and is represented as *condition* ? *value1* : *value2*. It evaluates the condition and returns *value1* if the condition is *true*, or *value2* if the condition is *false*.

This operator changes the return value, by modifying the values in the expression, usually, both *values* have the same value when applied this operator.

Before

```java
int data=a>b?c:d;
```

After

```java
int data=a>b?d:d;
```

5.2.15 For Loop Initial Condition Replacement

For Loop Initial Condition Replacement operator replaces the initial condition of a for loop with a randomly generated integer value. This means that the loop counter will start from a random value instead of the original initial value. Sometimes, the loop statement may never get executed.

Before

```java
for(int i=0;i<size;i++)
```

After

```java
for(int i=1;i<size;i++)
```

5.3 Android Specific

The Android Specific category includes mutation operators specifically tailored for the Android Framework. These operators target key components such as Intent and activities (Activity), that are exclusive of the Android Operating System. In addition, this section will also cover the approach decided on to designing operators capable of generating *metamutants* suitable for the manipulation of XML layout files.

5.3.1 Intent

In this section, we delve into the development of specific operators that manipulate the Android Intent component. Intents play a vital role in the communication between different components within an Android application. They facilitate the launching of activities, passing data between components, and invoking various system services [2].

5.3.1.1 Random Action Intent Definition

Random Action Intent Definition operator replaces the parameter of an intent instantiation with a random value.

- **Before**

```java
Intent intent = new Intent(Intent.ACTION_VIEW);
```

- **After**

```java
Intent intent = new Intent(Intent.ACTION_EDIT);
```

5.3.1.2 Null Intent Operator

Null Intent Operator modifies the instantiation of an Intent object by setting it to null, resulting in a null intent object.

- **Before**

```java
Intent intent = new Intent(main.this, ImportActivity.class);
```

- **After**

```java
Intent intent = null;
```

5.3.1.3 Intent Target Replacement

The Intent Target Replacement (ITR) operator operates by searching for all classes within the same package as the current class. It then replaces the target of each Intent with all compatible classes found in the package [4].

- **Before**

```java
Intent intent = new Intent(ActivityA.this, ActivityB.class);
```

- **After**

```java
Intent intent = new Intent(ActivityA.this, ActivityC.class);
```

5.3.1.4 Invalid Key Intent Operator Mutator

putExtra() is an Intent method used to include extra information along with the Intent. It uses a key-value pair, where the key is a unique identifier and the value is the data being passed. The key is used to retrieve the data later, and the value can be of various types, including primitives, arrays, strings, or objects implementing Parcelable. Parcelable is an interface for classes whose instances can be written to and restored from a Parcel [108].

The Invalid Key Intent Operator is responsible for randomly generating a different key in an *intent.putExtra(key, value)* call.

- **Before**

```java
intent.putExtra(key, value);
```

- **After**

```java
intent.putExtra("ecab6839856b426fbdae3e6e8c46c38c", value);
```

5.3.1.5 Null Value Intent Put Extra

Null Value Intent Put Extra operator modifies the behaviour of Intent.putExtra(key, value) by replacing the value argument with a new instance of Parcelable[0]. Using an empty array of Parcelable objects as the value indicates that there are no actual objects to be passed with the Intent.

- **Before**

```
intent.putExtra(key, value);
```

- **After**

```
intent.putExtra(key, new Parcelable[0]);
```

5.3.1.6 Intent Payload Replacement

The Intent Payload Replacement Operator is designed to replace the payloads of an Intent object with predefined default values. This operator assigns specific default values based on the data type of each payload. For instance, primitive types like integers or floats are replaced with the value zero, boolean payloads are replaced with both true and false values and string payloads are replaced with empty strings or null values. In the case of an array or other types of payloads, they are replaced with null values cast into the corresponding type [4].

- **Before**

```
intent.putExtra(EXTRA_MESSAGE, message);
```

- **After**

```
intent.putExtra(EXTRA_MESSAGE, "");
```

5.3.2 GUI

This section describes the GUI operators implemented for Android. They focus on activity components and XML-related operations and are designed to modify and manipulate the graphical user interface (GUI) elements of an Android application, the activity operators target the behaviour and functionality of activities, while the XML-related operators target the layout and appearance of the user interface defined in XML files.

5.3.2.1 FindViewById Returns Null

In Android development, the *findViewById()* method is often used to retrieve a reference to a specific view component within an activity's layout using its unique identifier. Nonetheless, when the FindViewById Returns Null operator is applied, the result of the *findViewById()* method is intentionally set to null. This means that the variable holding the reference to the view will no longer point to a valid object in the UI, effectively disconnecting it from the corresponding view component.

- **Before**

```
ImageButton loadButton = (ImageButton) findViewById(R.id.load_data_button);
```

- **After**

```
ImageButton loadButton = null;
```

5.3.2.2 Invalid ID FindView

When the Invalid ID FindView operator is applied, the original ID argument in the findViewById call is substituted with an invalid ID that does not correspond to any existing view component.

- **Before**

```
TextView emailTextView = (TextView) findViewById(R.id.EmailTextView);
```

- **After**

```
TextView emailTextView = (TextView) findViewById(839);
```

5.3.2.3 Invalid View Focus

In Android applications, GUI components such as buttons, text fields, and checkboxes can receive focus, meaning selection for user interaction. The focus determines which component will receive user input, keyboard events or touch events. The Invalid View Focus operator works by randomly selecting a GUI component in the application and changing its focus. This means that the originally focused component will lose focus, and a different component will become focused instead.

- **Before**

```
TextView emailTextView = (TextView) findViewById(R.id.EmailTextView);
```

- **After**

```
TextView emailTextView = (TextView) findViewById(R.id.EmailTextView);
emailTextView.requestFocus();
```

5.3.2.4 View Component Not Visible

View Component Not Visible Operator makes the view element invisible by setting its visibility attribute to false [7].

- **Before**

```
TextView emailTextView = (TextView) findViewById(R.id.EmailTextView);
```

- **After**

```
TextView emailTextView = (TextView) findViewById(R.id.EmailTextView);
emailTextView.setVisibility(android.view.View.INVISIBLE);
```

5.3.2.5 Buggy GUI Listener

The GUI listener is responsible for handling user interactions and events, button clicks or touch events. Buggy GUI Listener modifies the instantiation of the GUI listener with a null value. This means that the GUI component will have no listener attached to it, resulting in no response to user interactions or events.

- **Before**

```java
private View.OnClickListener listener = new View.OnClickListener() {
   @Override
   public void onClick(View view) {
     clicksCount += 1;
   }
}
```

- **After**

```java
private View.OnClickListener listener = null;
```

5.3.2.6 Lengthy GUI Creation

Lengthy GUI Creation operator introduces a deliberate delay in the creation of the GUI thread of an Android application by using the Thread.sleep() method, which pauses the execution of the current thread for a specified amount of time. The GUI thread is responsible for creating and initializing the graphical user interface components of the application. Inserting a long delay in the GUI thread may cause the application's user interface to appear unresponsive or frozen during that period.

- **Before**

```java
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
}
```

- **After**

```java
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    try {
        Thread.sleep(10000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    setContentView(R.layout.main);
}
```

5.3.2.7 Lengthy GUI Listener

This operator introduces a long delay in the listener GUI thread. It is responsible for handling user interactions and events, button clicks or touch events. By adding a delay in this thread, the responsiveness of the application's user interface may be impacted.

- **Before**

```java
private View.OnClickListener listener = new View.OnClickListener() {
    @Override
    public void onClick(View view) {
      clicksCount += 1;
    }
```

- **After**

```java
private View.OnClickListener listener = new View.OnClickListener() {
    @Override
    public void onClick(View view) {
      clicksCount += 1;

    try {
        Thread.sleep(10000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

}
```

5.3.2.8 XML Layout Operators

Up to this point, there are no implementations of mutation operators that generate *metamutant* and can directly affect XML files. This is due to the absence of logic in XML files, which prevents the inclusion of the necessary if statements required for creating *metamutants*.

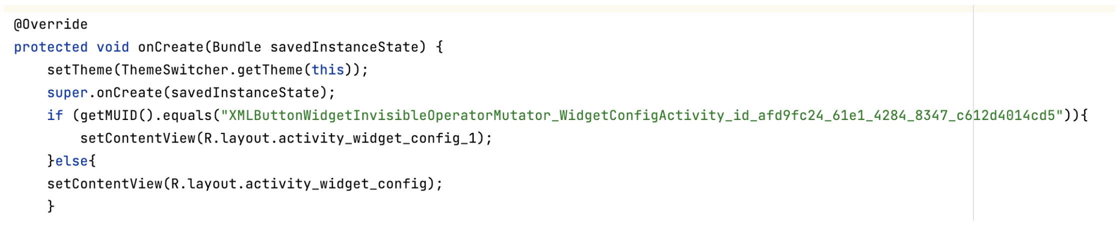
Nevertheless, a new approach has been developed to address this limitation and apply targeted changes to XML files within the context of *metamutants*. This approach focuses specifically on XML layout files, which define the structure and elements of the user interface in an Android app.

XML layout files are written in XML (eXtensible Markup Language) and constitute a representation of the hierarchy of interface elements in an Android app. These elements include buttons, images, text fields, and many other visual components easily customized and edited. Typically, each activity has its own corresponding layout file. This layout file dictates how the UI elements are organized and displayed on the screen specifically for that activity. To associate an activity with its respective layout, the *setContentView()* method is used within the activity class. This method is typically called in the *Activity.onCreate()* method, where the layout resource is passed as a reference using the *R.layout.layout_file_name* notation. When the layout XML file is assigned as the content view for the activity, the UI elements specified within the layout become visible on the screen when the activity is active.

Hence, the opted approach to modifying layout files using *metamutants* was the following: Begin by searching for a specific node, whose reference is "setContentView - Executable", in order to find instances of *setContentView()* method and consequently the name of the layout file that is being referenced. Once the layout file name was identified, a search was carried out for the corresponding file within the res/layout folder of the project. This folder contains all the layout files used in the application's user interface.

After finding the layout file, a copy is made and a new name is assigned to the duplicate file. This ensures that the original layout file remains unchanged, while modifications can be freely applied to the copied file. Finally, the reference to the layout file in the activity file was updated, by replacing the original layout file name with the name of the mutated file. Figure 5.1 represents the structure of the resulting *metamutant* when any XML operator is applied, in this case, was used the XML Button Widget Invisible Operator. Figure 5.2 represents the resulting res/layout folder after inserting a mutation.

After the explanation of the process for applying XML mutations through *metamutants*, the implemented operators will be defined. The implemented operators can be classified into three categories based on the action they perform, such as Apply Change, Set Invisible, and Deletion.



```
@Override
protected void onCreate(Bundle savedInstanceState) {
    setTheme(ThemeSwitcher.getTheme(this));
    super.onCreate(savedInstanceState);
    if (getMUID().equals("XMLButtonWidgetInvisibleOperatorMutator_WidgetConfigActivity_id_afd9fc24_61e1_4284_8347_c612d4014cd5")){
        setContentView(R.layout.activity_widget_config_1);
    }else{
    setContentView(R.layout.activity_widget_config);
    }
```

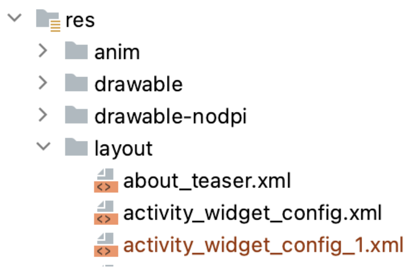
Figure 5.1: Resultant *metamutant*

Apply Change:

These operators make changes to the appearance of specific XML elements. They enable alterations to the visual attributes and properties of these elements, such as the background colour, text colour, text size, etc. Four operators were defined:

- **XML Invalid Color**, this operator modifies the "android:textColor" attribute of any XML element that has this attribute.
- **XML Button Widget Change Appearance**, this operator modifies the "android:textSize" attribute of Button elements in XML layouts.
- **XML EditText Widget Change Appearance**, this operator modifies the "android:textSize" attribute of EditText elements in XML layouts.
- **XML ViewGroup Widget Change Type**, this operator modifies the ViewGroup Type by replacing it with another type of View group, such as Relative Layout, Linear Layout, Table Layout, etc.

The impact of the operators is visually demonstrated in Figure 5.3. The initial image showcases the original layout, while the subsequent images showcase the altered layouts resulting from the application of the corresponding operators mentioned earlier. Each image corresponds to a specific operator, as enumerated previously.



```
res
    anim
    drawable
    drawable-nodpi
    layout
        about_teaser.xml
        activity_widget_config.xml
        activity_widget_config_1.xml
```

Figure 5.2: Addition of Activity_widget_config_1 file to *Layout* folder

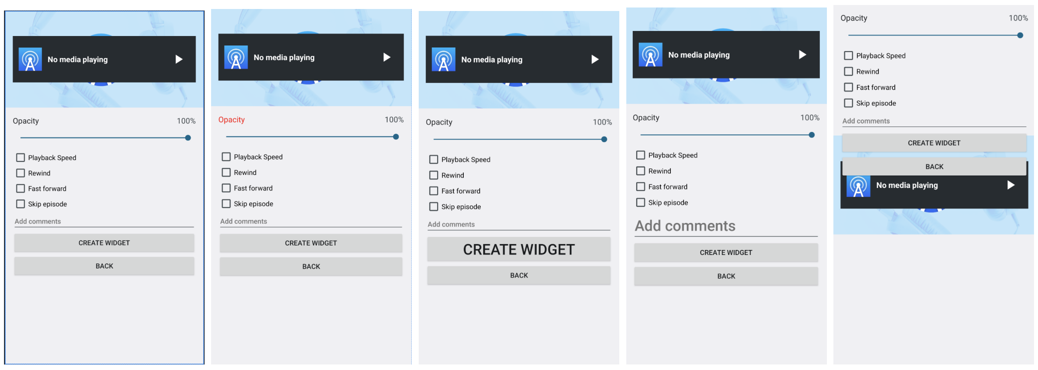


Figure 5.3: Layouts after applying the change operators

Set Invisible:

These operators are designed to modify the visibility of specific XML elements by adding or changing the visibility attribute ("android:visibility") to "invisible". This action makes the element invisible and inaccessible from the graphical user interface (GUI). However, the functionality specified in the Java code for that element still remains intact, and the element continues to occupy the same space within the layout. Three operators were defined:

- **XML EditText Widget Invisible** - when the target element is an EditText.
- **XML Button Widget Invisible** - when the target element is a Button
- **XML ViewGroup Widget Invisible** - when the target element is a ViewGroup

The effects of the operators are illustrated in Figure 5.4. The first image represents the original layout, while the subsequent images depict the modified layouts resulting from the application of the respective operators mentioned earlier. Each image corresponds to a specific operator as previously enumerated.

Deletion :

These operators are responsible for deleting specific XML elements from the layout by modifying the visibility attribute ("android:visibility") to "gone". Unlike the previous operators, when these elements are deleted, their functionality specified in the Java code is also removed, and the space they occupy within the layout is eliminated. Three operators have been defined for this purpose:

- **XML EditText Widget Deletion** -when the target element is an EditText
- **XML Button Widget Deletion** - when the target element is a Button
- **XML TextView Widget Deletion** - when the target element is a TextView

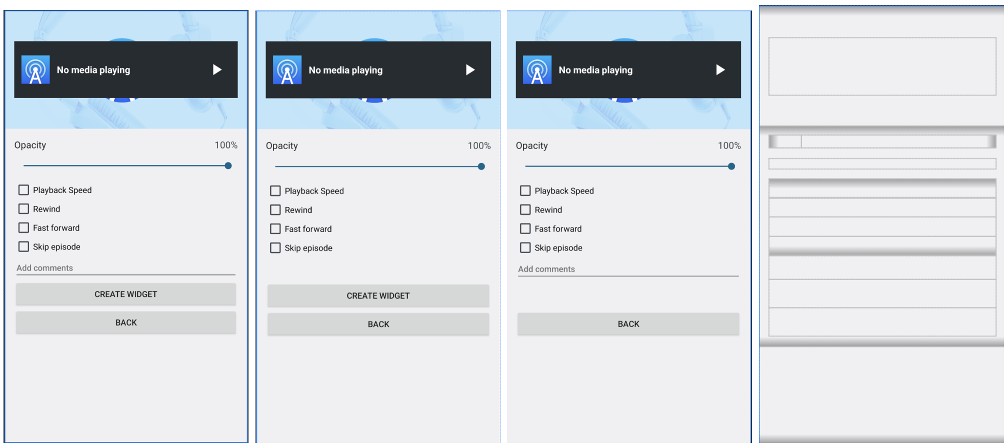


Figure 5.4: Layouts after applying the set invisible operators

Figure 5.5 demonstrates the effects of the operators. The first image shows the original layout, while the subsequent images display the respective layouts after applying the operators described previously. Each image corresponds to a specific operator, as enumerated before.

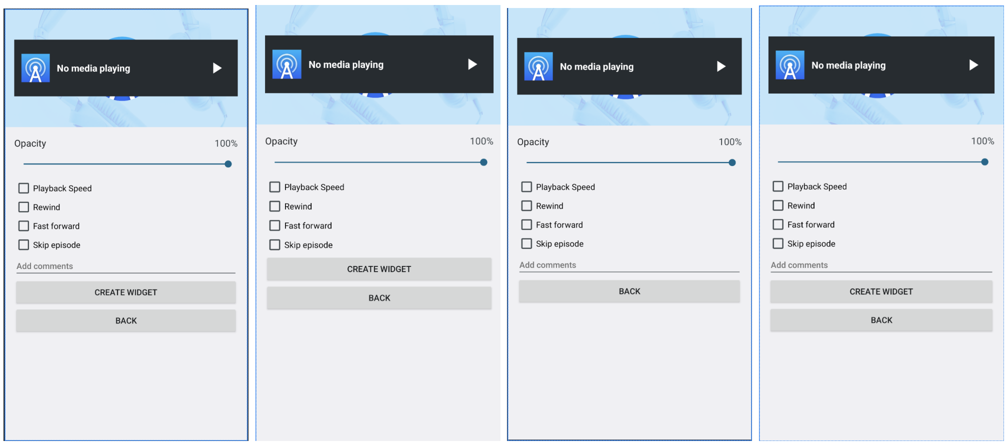


Figure 5.5: Layouts after applying the deletion operators

Additionally, two operators were defined that do not fall under any of the previously mentioned categories. These operators are:

5.3.2.9 Null Bluetooth Adapter

The Null Bluetooth Adapter operated replaces an instance of BluetoothAdapter with a null value in an Android application. BluetoothAdapter is a class in the Android framework that represents the device's Bluetooth functionality. By replacing it with null, the operator effectively removes the Bluetooth functionality from the application.

- **Before**

```
HttpResponse response = client.execute(httpGet);
```

- **After**

```
HttpResponse response = null;
```

5.3.2.10 Null GPS Location

The Null GPS Location operator injects a null GPS location into the location services of an Android application. This means that instead of receiving accurate GPS coordinates, the location services will return a null value for the GPS location.

- **Before**

```
Location GPSLocation = new Location(provider);
```

- **After**

```
Location GPSLocation = null;
```

In summary, we were able to implement Java and Intent operators and a considerable amount of GUI operators. Additionally, we established an approach that allows the generation of *meta-mutants* capable of affecting XML files, an aspect that was not possible with the traditional approach of Mutant Schemata. By applying declaration decomposition, it was also possible to increase the number of generated mutants in *metamutants*. Finally, we believe it is relevant to proceed with the investigation of Sensors, Configuration, Persistence, Location, and Connectivity in future research.

Chapter 6

Evaluating *metamutant* Implementation

In this chapter, we want to validate the effectiveness of the operators implemented in the previous section. To achieve this, first, we created a simple Android application [109] to study on a smaller scale the impact of the applied mutations, in order to understand if the mutations were being inserted correctly. However, to verify how operators behave in different scenarios, we decided to select a set of real-world applications, based on well-defined criteria.

6.1 Criteria to select Android Applications

Since part of the designed operators targets Android Specific components, we applied the following criteria:

- **Must be Android Native** because this work aims to define mutation operators that are specific to Android.
- **Source code must be available** in order to analyze the code and insert the mutations.
- **Mainly written in Java**, which is the target language of this work.
- **Have test cases implemented by the developers**, thus containing Android Instrumentation Tests, in order to test Android Specific components and Unit tests to test the Java-specific operators.
- **Have more than one Activity File defined**, otherwise Intent mutations cannot be applied.
- **May use *gradle*** to facilitate the elaboration and testing of the application.

- The **number of instrumentation tests** must be at least **higher than 30.**
- No older than 3 years.
- Have a number of forks >=1k and stars >= 2.5K

The highlighted ones are mandatory and the others are recommended.

6.2 Selection of Android Applications

The selection of the applications was made by browsing the Android projects available on GitHub [102]. Firstly we searched for Android applications written mainly in Java that already contained instrumentation tests using the query `topic:android language: Java "instrumentation tests"`. Despite using the query, the results did not match our criteria. After attempting multiple queries and not getting the results expected, we decided to utilise *ChatGPT* to search for git repositories with the desired characteristics. We inserted the following statement: "Android apps written entirely in Java that contain instrumental tests and are available on GitHub." After several tries, we managed to get a set of 60 different applications. For each application, we first analyze the language percentage range, if the language was at least 55% Java if so we then looked for "androidTest" folder which is where instrumentation tests are defined in the Android application, and whether or not it was empty. Our purpose was to discover an "androidTest" folder with as many files as possible, thus a higher number of tests. Unfortunately, more than half did not fulfil these 2 criteria. The majority was developed in Kotlin and the ones that fulfilled the language criteria did not contain any tests implemented. Some apps despite being written mainly in Java contained instrumentation testing developed in Kotlin. Only a subset of 17 applications fulfilled all the highlight criteria.

From the 17 applications that resulted from this selection phase, only 6 applications fulfilled all the criteria and were successfully built during the compilation phase. However, some changes had to be made in order to compile it, such as updating the version of some dependencies that were used and removing some building options that were needed to connect the application with external sources. The general *build failed* error message, *gradle sync failed*, *gradle version not supported*, and *missing settings* were the most prevalent issues seen on the other 11 projects.

From those 6, we selected 3, based on the number of instrumentation tests and also based on the kind of application. In our view, it did not make sense to have two applications that have the same functionality. Therefore, the selected ones were: AntenasPod, Omni-notes, and AmazeFile.

AntennaPod is "a podcast organiser and player that provides instant entry to millions of free and paid podcasts from small podcasters to huge publishing organisations like the BBC, NPR, and CNN" [8].

Omni-Notes is " an open-source note-taking program with a simple UI. Enhances the typical note-taking functionality of other basic programs by allowing users to attach picture and video files, utilise a range of widgets, categorise and organise notes, search through notes, and customise the application's user interface "[110].

Amaze File Manager is " an open-source file management program that allows going through all of the directories on your Android smartphone, moving files and folders, renaming documents, copying and pasting files, and performing a variety of other things. Amaze File Organiser is a straightforward yet effective file organiser " [111].

Table 6.1 displays some quantitative information about the apps.

Table 6.1: Characteristics of the apps

App	Disk space	Number of Java files	Number of Android Test	Number of Unit Test
AntennaPod	159,3 MB	760	102	226
Omni-Notes	79,8 MB	470	102	23
Amaze File Manager	539,7 MB	976	36	996

6.3 Applying Mutation Operators

Two experiments were undertaken to validate the exactness and effectiveness of cost resource consumption of the *metamutant* implementation.

6.3.1 Comparing the Correctness of *Metamutant* Implementation:

In order to demonstrate the correctness of the *metamutant* implementation in relation to the traditional implementation, the first experience involved applying a set of mutation operators to the previously selected applications. The objective was to generate mutants using both the *metamutant* implementation and the traditional implementation to analyze the resulting number of generated mutants and, consequently, the results obtained from the tests conducted on the mutants generated by both methods. Due to time constraints and some *Kadabra* limitations, it was not possible to analyse all the selected applications. Therefore, we chose the one with more instrumentation tests and a higher number of Unit tests within the three applications selected before, since a higher number of tests means a higher probability of detecting mutants.

AntenasPod was the one selected, with a total of 102 instrumentation tests. However, six were failing before applying the mutations, and we decided to eliminate them.

For each selected mutation operator, the following generation approaches were applied:

1. Generate Mutants using the traditional implementation (T).

2. Generate Mutants using the *Metamutant* implementation (MT).

The results obtained from the previous step are listed in Table 6.2.

Table 6.2: Results of Selected Operators

Operator	**N$_T$**	**N$_{MT}$**	**Fail$_T$**	**Fail$_{MT}$**	**Passed**
Arithmetic Operator	38	38	27	27	199
InvalidDate	0	0	0	0	226
NullIntent	47	47	30	30	66
RandomActionIntentDefinition	47	47	36	36	62
IntentTargetReplacement	33	33	34	34	62
InvalidKeyIntent	41	41	30	30	66
NullValueIntentPutExtra	41	41	30	30	66
IntentPayloadReplacement	41	41	28	28	68
BuggyGUIListener	2	2	5	5	91
LengthyGUIListener	2	2	5	5	91
Lengthy GUI Creation	23	23	12	12	84
FindViewByIdReturnsNull	33	33	-	-	-
ViewComponentNotVisible	33	33	0	0	96
InvalidViewFocus	33	33	0	0	96
InvalidIDFindView	33	33	-	-	-

N_T - number of Traditional Mutants; N_{MT} - number of *metamutants*; $Fail_T$ - number of failed tests using traditional Mutants; $Fail_{MT}$ - number of failed tests using *metamutants*. As we can observe in Table 6.2, both implementations created the same number of mutants generated. As well as the test results, the number of failed tests was the same on the two implementations.

During the testing of this experiment, we observed that two operators, namely "InvalidIDFindView" and "FindViewByIdReturnsNull," can potentially cause the application to crash if the modified variables are accessed before being verified, although when the application did not crash 35 tests failed on the 2 Implementations. However, this issue can be mitigated by performing checks on the variables before accessing them. This behaviour is observed when both traditional and *metamutant* operators are applied.

Testing this application led us to conclude that the *metamutant* implementation is equivalent to the traditional implementation when considering these test cases.

Using the approach developed in the previous section, we decided to test the number of XML mutants generated in this application. Due to time constraints, this approach only generates mutants for XML files loaded through the *setContentView()* method, so the number of XML mutations applied depends on the number of XML files, which is relatively low in this application, and thus the number of generated *metamutants* is also low. Expanding this approach to the remaining existing methods would suffice to attain a higher number of mutants.

Table 6.3 presents the expected and the results obtained of the XML Operators. In order to calculate the number of expected operators, we analyzed all the XML files.

Table 6.3: Expected and Result of XML Operators

Operator	**Number of Mutants Expected**	**Number of Mutants Obtained**
XMLEditTextWidgetInvisible	0	0
XMLViewGroupWidgetInvisible	79	10
XMLButtonWidgetInvisible	23	3
XMLEditTextWidgetDeletion	6	0
XMLButtonWidgetDeletion	22	3
XMLTextViewWidgetDeletion	54	3
XMLInvalidColor	37	2
XMLButtonWidgetChangeAppearance	22	3
XMLEditTextWidgetChangeAppearance	6	0
XMLViewGroupWidgetChangeType	79	10

Since there were no tests developed to test XML mutants, manual tests were done. Only three XML files are being loaded through the *setContentView()* method, namely "activity_widget_config.xml", "bug_report.xml","main.xml" Therefore, the developed operators will only act on these three files.

To perform the manual tests, we applied the following steps:

- Initially, we generate XML mutants.
- Then, we analyzed the mutated files.
- Lastly, in order to understand how these mutants alter the application behaviour, we observed the impact of the operators on the application.

An example of the performed tests will be demonstrated to the operator "XMLButtonWidgetDeletion" and "XMLViewGroupWidgetInvisible" in bug_report XML file.

Figure 6.1 displays the layout of Bug_report file before applying any operator.

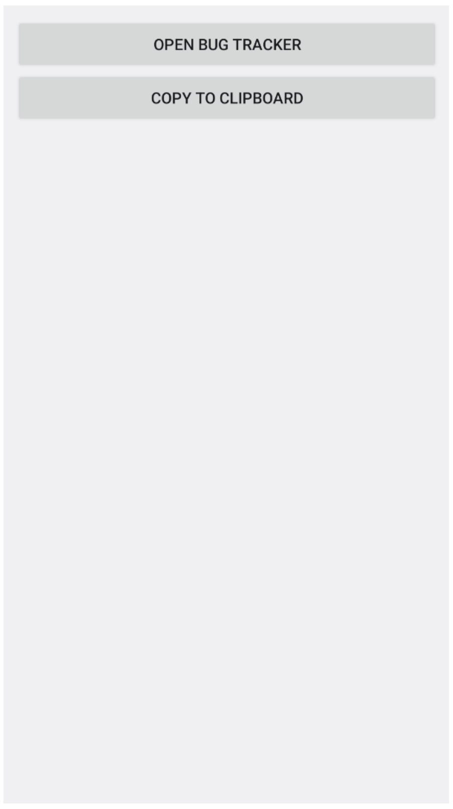


Figure 6.1: Layout of *bug_report* XML file from [8].

After applying the "XMLButtonWidgetDeletion" to the selected file, 2 new files were created, one for each button. Figure 6.2 shows the created files.

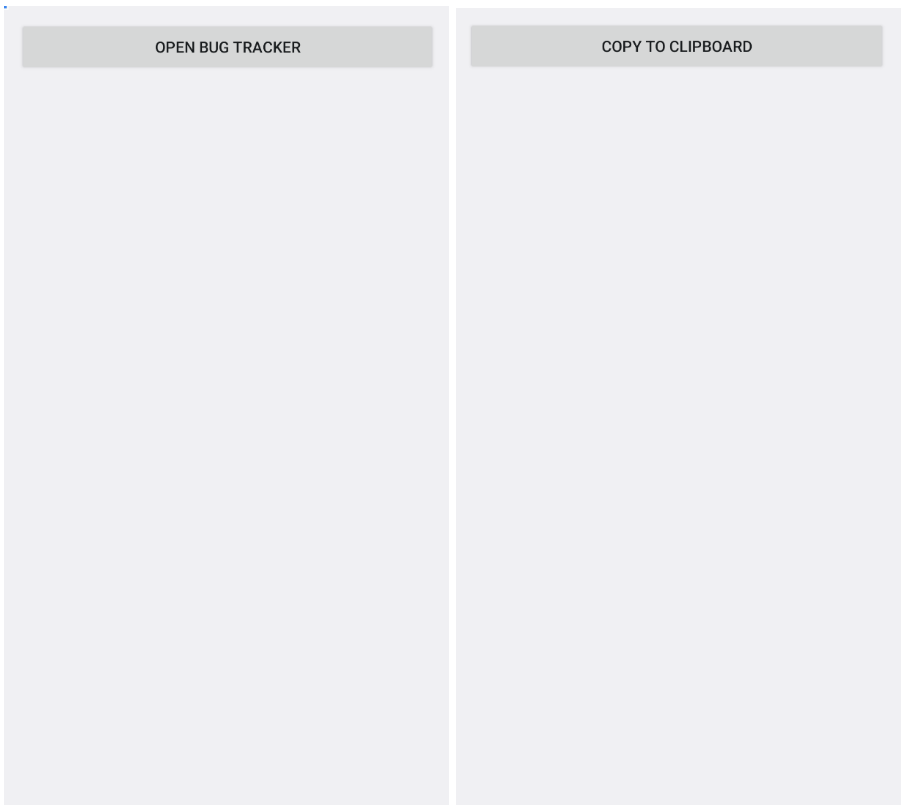


Figure 6.2: Layouts of *bug_report* file after applying "XMLButtonWidgetDeletion" operator.

One approach to detect this type of mutation is by conducting a test that checks for the presence (searching for each button id) of all buttons within a file, ensuring that all the buttons are properly displayed to the user.

By applying "XMLViewGroupWidgetInvisible" in bug_report XML file, one file was created. Figure 6.3 shows the file generated.

In order to detect the presence of this mutant, the test may attempt to perform any possible action on the page, such as trying to click on one of the buttons on the page. However, any action taken will not generate the expected result because no component is available. When an action is performed and the resulting outcome does not align with the expected behaviour, it serves as an indication that a mutant is present.

Despite not being given much prominence, XML operators play a significant role in mutation testing as they directly impact the usability of an application.

There are still some improvements that need to be made in order to support a greater amount of XML files and subsequently XML mutants in the presented approach, and we consider that supporting XML mutants in a *metamutant* implementation is a significant achievement.

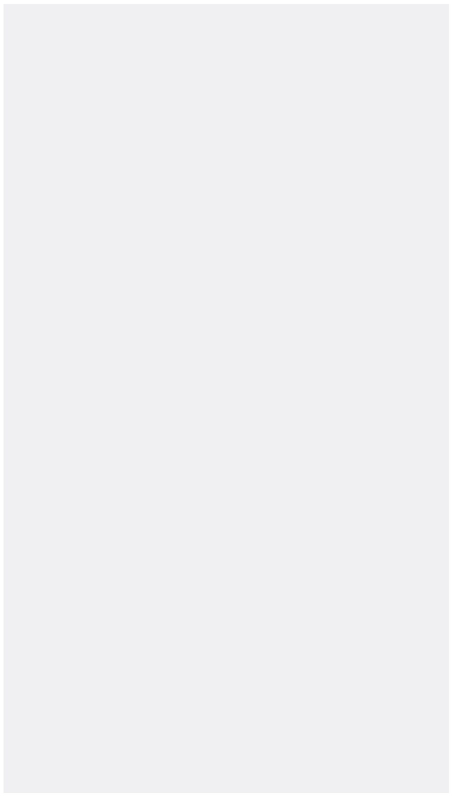


Figure 6.3: Layouts of *bug_report* file after applying "XMLViewGroupWidgetInvisible" operator.

6.3.2 Analyzing Resource Consumption:

To assess the resource consumption (time and memory) of the two implementations, a second experiment was performed. This experiment involved analyzing the generation time and disk space utilization as the number of mutants increased. To perform this experience we used a MacBook Pro with macOS Ventura (version 13.4.1), with a 2,3 GHz Intel Core i9 processor.

As demonstrated in Figure 6.4, with the increase of the number of mutants, the generation time for both approaches also rises, indicating that the generation process becomes more time-consuming with the increase of mutants.

The *metamutant* approach has an overall lower generation time compared to the traditional method for the same amount of mutants, suggesting that the *metamutant* approach is more efficient. The difference in generation times becomes more noticeable as mutants increase. For instance, at 48 mutants, the *metamutant* approach needed 37.4 seconds, while the traditional approach took 89.4 seconds. However, when the number was higher, at 502 mutants, the *metamutant* approach required 192.5 seconds (3 minutes), whereas the traditional approach 2025.3

seconds (34 minutes). This shows that the traditional approach suffers a steeper increase in generation time compared to the *metamutant* approach. This implies that the traditional approach may have problems when dealing with a larger number of mutants.

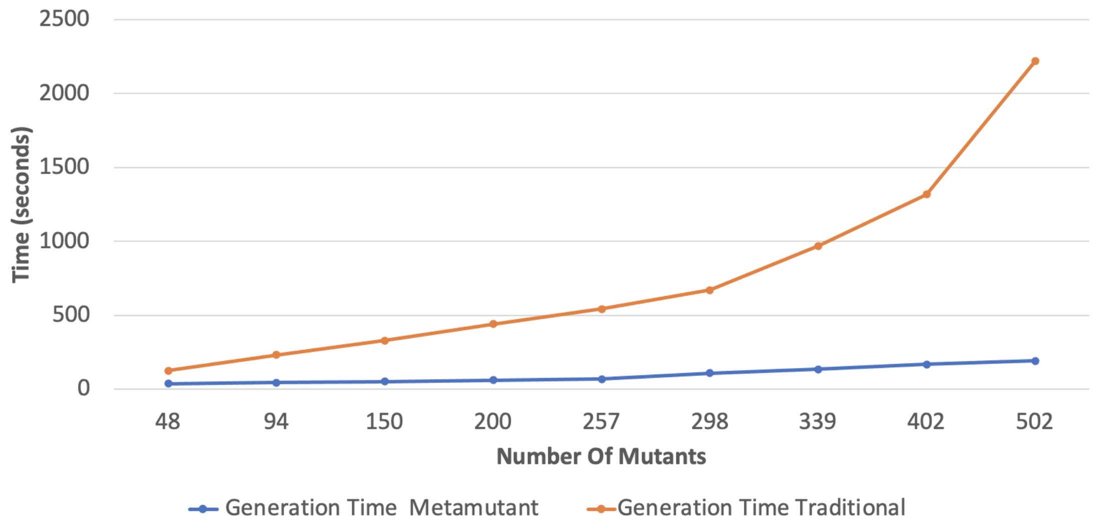


Figure 6.4: Relation between the number of mutants and utilized generation Time

Based on Figure 6.5, we can infer that disk space usage for both approaches also increases as the number of mutants rises, which is expected since more mutants require more storage space to store the generated data.

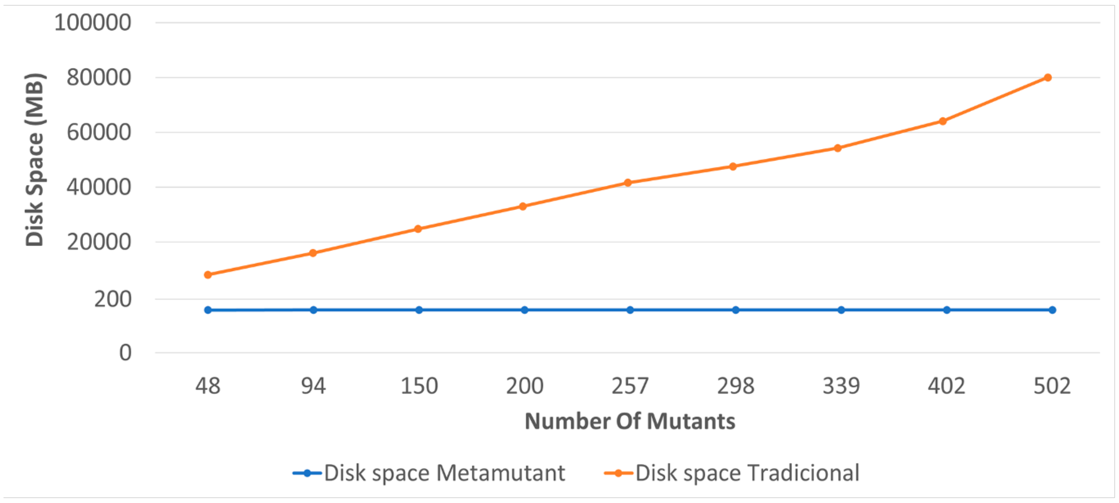


Figure 6.5: Relation between the number of mutants and utilized Disk Space

The disk space usage for the *metamutant* approach remains relatively constant across different numbers of mutants, at around 159-159.9 MB, demonstrating consistent and efficient use of disk space, regardless of the complexity of the mutation set.

In comparison, the disk space usage for the traditional method displays a significant increase as the number of mutants grows. For example, at 48 mutants, the traditional approach required 8040 MB (8GB) of disk space, and with the increase in number, at 502 mutants, it needed 79980 MB (80GB).

The difference between the two approaches becomes more marked as the number of mutants increases. The *metamutant* approach consistently uses significantly less disk space when compared to the traditional approach for the same number of mutants.

Based on the available data, it can be concluded that the *metamutant* approach outperforms the traditional approach in terms of generation time and demonstrates more efficient utilization of disk space.

Chapter 7

Conclusion and Future Work

In this work, several mutation operators were implemented in order to produce mutants with the structure of Mutant Schemata, also called *metamutants*. The implemented mutation operators are distributed into three categories: General (14), Java Specific (14) and Android Specific (25). It was also possible to apply mutations in XML files through *metamutant*, which, previously to this study, was not achieved.

In order to assess the correctness of the implemented operators, these mutation operators were inserted in two Android applications, one created and one available on GitHub. Due to time constraints, it was not possible to test the other two selected applications. When testing the generation of XML mutants in the AntennaPod we concluded that there are different ways to load the XML files in activity files that were not considered and thus this approach must be improved in order to apply mutations to a higher number of XML files.

The results obtained from this work were that the number of mutants generated by *metamutant* approach was equal to the traditional, as well as the number of failed tests.

Nevertheless, the use of *metamutant* presents a notable advantage in terms of time and memory compared to the traditional implementation. When the number of mutants exceeds 500, there is a significant time reduction of approximately 30 minutes. Moreover, the difference in disk space is even more substantial, with a size reduction of approximately 80GB. However, the *metamutant* approach has one major disadvantage, if a mutation operator leads to a compilation error (which is not expected to occur), it can result in the generation of invalid mutants. In such cases, all the generated mutants may be discarded, contrary to traditional.

To conclude, the Mutant Schema approach demonstrates significant advantages in terms of time efficiency and disk space utilization compared to the traditional approach when generating mutants.

The main limitations found while executing this work were the lack of investigation regarding this topic, the scarce documentation of *Kadabra* tool and some *Kadabra* limitations, which hindered and delayed all the processes.

For future work, we suggest expanding the current approach to cover the remaining categories of mutant operators and increasing the number of tested applications. This would provide a more comprehensive evaluation of the approach's effectiveness and applicability.

Additionally, it would be valuable to investigate approaches that initiate the testing process directly from the APK since it is difficult to find Java-based Android applications for testing purposes. An approach that starts from the APK file allows for testing real-world applications that may not have their source code readily available or accessible, and there are recent works that start from the APK and decompile the binary to Java source-code [112], allowing the application of Java-based approaches to APK files. However, it is important to acknowledge that mutation testing from the APK introduces additional complexities compared to traditional mutation testing on source code.

References

[1] "The activity lifecycle|android developers." https://developer.android.com/guide/components/activities/activity-lifecycle. (Accessed on 12/01/2023).

[2] "Intents e filtros de intents|desenvolvedores android|android developers." https://developer.android.com/guide/components/intents-filters. (Accessed on 12/01/2023).

[3] R. Trujillo, I. de la Caridad, *et al.*, "Mutation testing techniques for mobile applications.," 2021.

[4] L. Deng, *Mutation Testing for Android Applications*. PhD thesis, 2017.

[5] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2010.

[6] R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation analysis using mutant schemata," in *Proceedings of the 1993 ACM SIGSOFT international symposium on Software testing and analysis*, pp. 139–148, 1993.

[7] "Mdroid+ android developer tools." https://www.android-dev-tools.com/mdroid#demo2. (Accessed on 19/06/2023).

[8] "Antennapod: A podcast manager for android." https://github.com/AntennaPod/AntennaPod. (Accessed on 10/06/2023).

[9] D. Halabi and A. Shaout, "Mutation testing tools for java programs–a survey," *IJCSE Int J Comput Sci Eng*, vol. 5, pp. 11–22, 2016.

[10] H. N. Silva, J. Prado Lima, S. R. Vergilio, and A. T. Endo, "A mapping study on mutation testing for mobile applications," *Software Testing, Verification and Reliability*, vol. 32, no. 8, p. e1801, 2022.

[11] "Number of mobile devices worldwide 2020-2025." https://www.statista.com/statistics/245501/multiple-mobile-device-ownership-worldwide/. (Accessed on 29/11/2022).

[12] L. Deng and J. Offutt, "Reducing the cost of android mutation testing.," in *SEKE*, pp. 542–541, 2018.

[13] "Apps android no google play." https://play.google.com/store/apps?hl=pt_PT&gl=US. (Accessed on 29/11/2022).

[14] S. Barraood, "Test case quality factors," *Turkish Journal of Computer and Mathematics Education (TURCOMAT)*, vol. 12, pp. 1683–1694, 04 2021.

[15] "What is software development?." https://www.ibm.com/topics/software-development. (Accessed on 08/11/2022).

[16] "What is software testing and how does it work?." https://www.ibm.com/topics/software-testing. (Accessed on 22/11/2022).

[17] S. T. E. Abbas, R. Hassan, S. Abd Halim, S. Kasim, and R. Ramlan, "Investigation on java mutation testing tools," *JOIV: International Journal on Informatics Visualization*, vol. 6, no. 2-2, pp. 455–462, 2022.

[18] T. Titcheu Chekam, M. Papadakis, T. F. Bissyandé, Y. Le Traon, and K. Sen, "Selecting fault revealing mutants," *Empirical Software Engineering*, vol. 25, pp. 434–487, 2020.

[19] "Istqb glossary." https://glossary.istqb.org/en/term/test-case-3. (Accessed on 29/12/2022).

[20] A. P. Mathur, "Chapter 8: Test adequacy assessment using program mutation." https://www.oreilly.com/library/view/foundations-of-software/9788131794760/xhtml/chapter008-1.xhtml#h5-003, May 2013. (Accessed on 30/12/2022).

[21] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[22] A. J. Offutt, "Investigations of the software testing coupling effect," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 1, no. 1, pp. 5–20, 1992.

[23] S. M. A. Ferreira, "Mutation-based web test case generation," 2019.

[24] D. Amalfitano, A. C. Paiva, A. Inquel, L. Pinto, A. R. Fasolino, and R. Just, "How do java mutation tools differ?," *Communications of the ACM*, vol. 65, no. 12, pp. 74–89, 2022.

[25] "Software testing|mutation testing." https://www.geeksforgeeks.org/software-testing-mutation-testing/. (Accessed on 29/12/2022).

[26] "Test cases and test suites." https://www.ibm.com/docs/en/elm/7.0.3?topic=scripts-test-cases-test-suites. (Accessed on 29/12/2022).

[27] Y. Wei, "Mudroid: Mutation testing for android apps," *Univ. College London, London, UK, Tech. Rep*, 2015.

[28] "Introduction to:software testing." http://www.cse.hcmut.edu.vn/~hiep/KiemthuPhanmem/Tailieuthamkhao/Introduction%20to%20Software%20Testing.pdf. (Accessed on 10/01/2023).

[29] "Android-overview." https://www.tutorialspoint.com/android/android_overview.htm. (Accessed on 09/01/2023).

[30] "What is an apk file and what does it do? explained." https://www.makeuseof.com/tag/what-is-apk-file/. (Accessed on 07/12/2022).

[31] "Application manifest file in android." https://www.geeksforgeeks.org/application-manifest-file-in-android/. (Accessed on 12/01/2023).

[32] "Fundamentos de aplicativos|desenvolvedores android|android developers." https://developer.android.com/guide/components/fundamentals#DeclaringComponents. (Accessed on 12/01/2023).

[33] "Services overview|android developers." https://developer.android.com/guide/components/services. (Accessed on 12/01/2023).

[34] "Broadcasts overview | android developers." https://developer.android.com/guide/components/broadcasts. (Accessed on 18/01/2023).

[35] "Content providers|android developers." https://developer.android.com/guide/topics/providers/content-providers. (Accessed on 01/12/2023).

[36] L. Deng, N. Mirzaei, P. Ammann, and J. Offutt, "Towards mutation analysis of android apps," in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 1–10, IEEE, 2015.

[37] A. A. Saifan and A. A. Alzyoud, "Mutation testing to evaluate android applications," *International Journal of Open Source Software and Processes (IJOSSP)*, vol. 11, no. 1, pp. 23–40, 2020.

[38] L. Deng, J. Offutt, and D. Samudio, "Is mutation analysis effective at testing android apps?," in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp. 86–93, IEEE, 2017.

[39] M. P. Usaola, G. Rojas, I. Rodriguez, and S. Hernandez, "An architecture for the development of mutation operators," in *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 143–148, IEEE, 2017.

[40] M. Linares-Vásquez, G. Bavota, M. Tufano, K. Moran, M. Di Penta, C. Vendome, C. Bernal-Cárdenas, and D. Poshyvanyk, "Enabling mutation testing for android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 233–244, 2017.

[41] R. Jabbarvand and S. Malek, "μdroid: an energy-aware mutation testing framework for android," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 208–219, 2017.

[42] A. C. Paiva, J. M. Gouveia, J.-D. Elizabeth, and M. E. Delamaro, "Testing when mobile apps go to background and come back to foreground," in *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 102–111, IEEE, 2019.

[43] I. C. Morgado, A. C. R. Paiva, and J. P. Faria, "Dynamic reverse engineering of graphical user interfaces," 2012.

[44] I. C. Morgado, A. C. Paiva, and J. P. Faria, "Automated pattern-based testing of mobile applications," in *2014 9th International Conference on the Quality of Information and Communications Technology*, pp. 294–299, 2014.

[45] I. C. Morgado and A. C. R. Paiva, "The impact tool: Testing ui patterns on mobile applications," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 876–881, 2015.

[46] I. C. Morgado and A. C. R. Paiva, "Testing approach for mobile applications through reverse engineering of ui patterns," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pp. 42–49, 2015.

[47] I. C. Morgado and A. C. Paiva, "Impact of execution modes on finding android failures," *Procedia Computer Science*, vol. 83, pp. 284–291, 2016.

[48] I. C. Morgado and A. C. R. Paiva, "The impact tool for android testing," vol. 3, no. EICS, 2019.

[49] I. C. Morgado and A. C. R. Paiva, "Test patterns for android mobile applications," (New York, NY, USA), Association for Computing Machinery, 2015.

[50] A. M. P. Grilo, A. C. R. Paiva, and J. P. Faria, "Reverse engineering of gui models for testing," in *5th Iberian Conference on Information Systems and Technologies*, pp. 1–6, 2010.

[51] I. Coimbra Morgado and A. Paiva, "Mobile gui testing," *Software Quality Journal*, vol. 26, 12 2018.

[52] C. Escobar-Velásquez, M. Linares-Vásquez, G. Bavota, M. Tufano, K. Moran, M. Di Penta, C. Vendome, C. Bernal-Cárdenas, and D. Poshyvanyk, "Enabling mutant generation for open-and closed-source android apps," *IEEE Transactions on Software Engineering*, vol. 48, no. 1, pp. 186–208, 2020.

[53] K. Moran, M. Tufano, C. Bernal-Cárdenas, M. Linares-Vásquez, G. Bavota, C. Vendome, M. Di Penta, and D. Poshyvanyk, "Mdroid+ a mutation testing framework for android," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pp. 33–36, 2018.

[54] E. Luna and O. El Ariss, “Edroid: A mutation tool for android apps,” in *2018 6th international conference in software engineering research and innovation (CONISOFT)*, pp. 99–108, IEEE, 2018.

[55] “Robotium android ui testing.” `https://github.com/RobotiumTech/robotium`. (Accessed on 08/01/2023).

[56] “Espresso - android developers.” `https://developer.android.com/training/testing/espresso`. (Accessed on 17/01/2023).

[57] “Selendroid: Selenium for android.” `http://selendroid.io/`. (Accessed on 17/01/2023).

[58] “Junit 5.” `https://junit.org/junit5/`. (Accessed on 17/01/2023).

[59] S. Hamimoune and B. Falah, “Mutation testing techniques: A comparative study,” in *2016 international conference on engineering & MIS (ICEMIS)*, pp. 1–9, IEEE, 2016.

[60] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, “Mutation analysis.,” tech. rep., Georgia Inst of Tech Atlanta School of Information And Computer Science, 1979.

[61] T. A. Budd, *Mutation analysis of program test data.* Yale University, 1980.

[62] A. P. Mathur and W. E. Wong, “An empirical comparison of data flow and mutation-based test adequacy criteria,” *Software Testing, Verification and Reliability*, vol. 4, no. 1, pp. 9–31, 1994.

[63] R. DeMillo, D. Guindi, W. McCracken, A. Offutt, and K. King, “An extended overview of the mothra software testing environment,” in *[1988] Proceedings. Second Workshop on Software Testing, Verification, and Analysis*, pp. 142–151, 1988.

[64] K. N. King and A. J. Offutt, “A fortran language system for mutation-based software testing,” *Software: Practice and Experience*, vol. 21, no. 7, pp. 685–718, 1991.

[65] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei, “Is operator-based mutant selection superior to random mutant selection?,” in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pp. 435–444, 2010.

[66] A. Derezińska and M. Rudnik, “Evaluation of mutant sampling criteria in object-oriented mutation testing,” in *2017 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pp. 1315–1324, IEEE, 2017.

[67] A. P. Mathur, “Performance, effectiveness, and reliability issues in software testing,” in *1991 The Fifteenth Annual International Computer Software & Applications Conference*, pp. 604–605, IEEE Computer Society, 1991.

[68] A. J. Offutt, G. Rothermel, and C. Zapf, “An experimental evaluation of selective mutation,” in *Proceedings of 1993 15th international conference on software engineering*, pp. 100–107, IEEE, 1993.

[69] A. S. Namin and J. H. Andrews, “Finding sufficient mutation operators via variable reduction,” in *Second Workshop on Mutation Analysis (Mutation 2006-ISSRE Workshops 2006)*, pp. 5–5, IEEE, 2006.

[70] S. Hussain, “Mutation clustering,” *Ms. Th., Kings College London, Strand, London*, p. 9, 2008.

[71] Y. Jia and M. Harman, “Constructing subtle faults using higher order mutation testing,” in *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pp. 249–258, IEEE, 2008.

[72] Y. Jia and M. Harman, “Higher order mutation testing,” *Information and Software Technology*, vol. 51, no. 10, pp. 1379–1393, 2009.

[73] M. Polo, M. Piattini, and I. García-Rodríguez, “Decreasing the cost of mutation testing with second-order mutants,” *Software Testing, Verification and Reliability*, vol. 19, no. 2, pp. 111–131, 2009.

[74] M. Harman, Y. Jia, P. Reales Mateo, and M. Polo, “Angels and monsters: An empirical investigation of potential test effectiveness and efficiency improvement from strongly subsuming higher order mutation,” in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pp. 397–408, 2014.

[75] A. Abuljadayel and F. Wedyan, “An approach for the generation of higher order mutants using genetic algorithms,” *International Journal of Intelligent Systems and Applications*, vol. 12, no. 1, p. 34, 2018.

[76] M. Papadakis and N. Malevris, “An empirical evaluation of the first and second order mutation testing strategies,” in *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, pp. 90–99, IEEE, 2010.

[77] W. E. Howden, “Weak mutation testing and completeness of test sets,” *IEEE Transactions on Software Engineering*, no. 4, pp. 371–379, 1982.

[78] J. R. Horgan and A. P. Mathur, “Weak mutation is probably strong mutation,” techreport SERC-TR-83-P, Purdue University, West Lafayette, Indiana, 1990.

[79] M. Woodward and K. Halewood, “From weak to strong, dead or alive? an analysis of some mutation testing issues,” in *Workshop on software testing, verification, and analysis*, pp. 152–153, IEEE Computer Society, 1988.

[80] Y.-S. Ma, J. Offutt, and Y. R. Kwon, “Mujava: an automated class mutation system,” *Software Testing, Verification and Reliability*, vol. 15, no. 2, pp. 97–133, 2005.

[81] D. Schuler and A. Zeller, “Javalanche: Efficient mutation testing for java,” in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 297–298, 2009.

[82] P. R. Mateo and M. P. Usaola, “Bacterio: Java mutation testing tool: A framework to evaluate quality of tests cases,” in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pp. 646–649, IEEE, 2012.

[83] M. Papadakis and N. Malevris, “Automatic mutation test case generation via dynamic symbolic execution,” in *2010 IEEE 21st International Symposium on Software Reliability Engineering*, pp. 121–130, IEEE, 2010.

[84] S.-W. Kim, Y.-S. Ma, and Y.-R. Kwon, “Combining weak and strong mutation for a non-interpretive java mutation system,” *Software Testing, Verification and Reliability*, vol. 23, no. 8, pp. 647–668, 2013.

[85] P. R. Mateo and M. P. Usaola, “Reducing mutation costs through uncovered mutants,” *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 464–489, 2015.

[86] P. R. Mateo and M. P. Usaola, “Mutant execution cost reduction: Through music (mutant schema improved with extra code),” in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pp. 664–672, IEEE, 2012.

[87] R. H. Untch, M. J. Harrold, and A. J. Offutt, “Tums: testing using mutant schemata,” in *Proceedings of the 35th Annual Southeast Regional Conference*, pp. 174–181, 1997.

[88] C. J. Wright, G. M. Kapfhammer, and P. McMinn, “Efficient mutation analysis of relational database structure using mutant schemata and parallelisation,” in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pp. 63–72, IEEE, 2013.

[89] M. Polo-Usaola and I. Rodríguez-Trujillo, “Analysing the combination of cost reduction techniques in android mutation testing,” *Software Testing, Verification and Reliability*, vol. 31, no. 7, p. e1769, 2021.

[90] “Javaparser.” `https://javaparser.org/`. (Accessed on 17/01/2023).

[91] Y.-S. Ma and J. Offutt, “Description of mujava’s method-level mutation operators,” *Update*, 2016.

[92] M. E. Delamaro, J. Offutt, and P. Ammann, “Designing deletion mutation operators,” in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pp. 11–20, IEEE, 2014.

[93] J. Liu, X. Xiao, L. Xu, L. Dou, and A. Podgurski, “Droidmutator: an effective mutation analysis tool for android applications,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, pp. 77–80, 2020.

[94] R. A. Oliveira, E. Alégroth, Z. Gao, and A. Memon, "Definition and evaluation of mutation operators for gui-level mutation analysis," in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 1–10, IEEE, 2015.

[95] "Osf | supplementary material - a mapping study on mutation testing in mobile applications." https://osf.io/sbgm7/.

[96] "Kadabra tool." http://specs.fe.up.pt/tools/kadabra/. (Accessed on 11/06/2023).

[97] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A library for implementing analyses and transformations of java source code," *Software: Practice and Experience*, vol. 46, no. 9, pp. 1155–1179, 2016.

[98] P. Pinto, T. Carvalho, J. a. Bispo, and J. a. M. P. Cardoso, "Lara as a language-independent aspect-oriented programming approach," in *Proceedings of the Symposium on Applied Computing*, SAC '17, (New York, NY, USA), p. 1623–1630, Association for Computing Machinery, 2017.

[99] "Kadabra wiki." https://github.com/specs-feup/kadabra/wiki. (Accessed on 11/06/2023).

[100] D. Mata, "Cost reduction techniques for java applications - to be published," 2023.

[101] F. B. Azevedo, "Cost reduction technique for mutation testing," 2020. https://hdl.handle.net/10216/129884.

[102] "Android github." https://github.com/android. (Accessed on 17/06/2023).

[103] "Binaryoperator (java platform se 8)." https://docs.oracle.com/javase/8/docs/api/java/util/function/BinaryOperator.html. (Accessed on 18/06/2023).

[104] "Mutation operators." https://pitest.org/quickstart/mutators/#REMOVE_CONDITIONALS. (Accessed on 19/06/2023).

[105] "Unary operator (java platform se 8)." https://docs.oracle.com/javase/8/docs/api/java/util/function/UnaryOperator.html. (Accessed on 19/06/2023).

[106] "Droidmutator operators." https://github.com/SQS-JLiu/DroidMutator/blob/master/OperatorsDescription.md. (Accessed on 19/06/2023).

[107] "Java constructors." https://www.tutorialspoint.com/java/java_constructors.htm. (Accessed on 19/06/2023).

[108] "Parcelable | android developers." https://developer.android.com/reference/android/os/Parcelable. (Accessed on 10/06/2023).

[109] "Android app." https://github.com/RitaVeiga/AndroidApp/tree/main.

[110] "Omni-notes: Open source note-taking application for android." https://github.com/federicoiosue/Omni-Notes. (Accessed on 10/06/2023).

[111] "Amazefilemanager: Material design file manager for android." https://github.com/TeamAmaze/AmazeFileManager. (Accessed on 10/06/2023).

[112] N. Gregório, J. Bispo, J. P. Fernandes, and S. Queiroz de Medeiros, "E-apk: Energy pattern detection in decompiled android applications," *Journal of Computer Languages*, vol. 76, p. 101220, 2023.