Università degli Studi di Milano - Bicocca
Scuola di Scienze
Dipartimento di Informatica, Sistemistica e Comunicazione
Corso di laurea in Informatica

Development and evaluation of an automatic test case generation technique for Data Loss problems in Android apps

Relatore: Prof.ssa Daniela Micucci

Co-relatore: Dott. Oliviero Riganelli

Relazione della prova finale di:
Claudio Rota
Matricola 816050

Anno Accademico 2018-2019

Acknowledgements

Desidero ringraziare la Prof.ssa Daniela Micucci per avermi dato la possibilità di svolgere questo progetto di stage. Ringrazio il Dott. Oliviero Riganelli per avermi seguito durante tutto il progetto e per avermi aiutato con i suoi preziosi consigli.
Un ringraziamento speciale va alla mia famiglia e alla mia fidanzata Cristina, che mi hanno sempre sostenuto, incoraggiato e sopportato anche nei periodi di stress e nervosismo.
Un sentito ringraziamento va anche ad Alessandro, Simone, Stefano, Andrea Mi., Burt, Matteo e Andrea Ma., miei compagni di corso ed amici fin dal primo anno, senza i quali questi tre anni non sarebbero stati gli stessi. In particolare, ringrazio Simone per aver svolto questo progetto insieme a me.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Android apps, like all software applications, should be tested adequately to ensure a certain level of quality. Unfortunately, testing is usually one of the software development phases in which less time and money are invested. As a matter of fact, developers are used to implement test cases that are mainly focused on expected application behaviours, thus failing to consider less common events such as a home-button press or a sudden change in the screen orientation. These untested behaviours are often the cause of unexpected outcomes. As a result, these test cases are not effective on detecting application failures that may be found in real situations by the end users. The large number of negative comments in the Google Play Store indicates that a significant amount of applications are buggy.
Since Android applications are event-driven systems, in which the execution flow of the program is determined by events, it is often hard to perform tests on them. For this reason, testing automation for Android apps has become an attraction for many researchers and several tools have been developed with the purpose of helping Android developers to test their apps. These tools allow to explore the behaviours of the app automatically using different strategies, heuristics and input generation techniques in order to detect application crashes. However, this is their limitation, since there are software failures that do not always manifest themselves in crashes of the entire app, such as *Data Loss* failures.
Data Loss failures occur when the variables of the application lose their values after a *stop-start* event, which is an event that first stops the execution of the app and then resumes it from the same point it was stopped. If an app suffers from these problems and such events take place, the users may lose the data they have just entered, ruining the usability of the app. They are usually introduced by developers because they do not implement correctly the logic necessary for preserving and restoring the application state when these events happen. Since *stop-start* events occur very often, Android apps affected by *Data Loss* faults may manifest these failures very frequently, negatively impacting the user experience, which is very important for the success of the application.

Nowadays, there are no specific tools for detecting these types of failures. To address this problem, this thesis work presents an extension of an existing state-of-the-art test case generation tool with a novel technique able to detect *Data Loss* problems in Android apps. This technique analyzes the application under test automatically, using a *model-based* exploration approach, and heuristically generates *stop-start* events in conjunction with specific oracles. More in detail, the exploration strategy is based on a GUI model, which is a non-deterministic finite automaton (NFA) with *abstract states* as states, representing the app pages, and events as transitions between them. During the exploration, the events are triggered from the *abstract states* with different probabilities in order to favour event not yet triggered. By default, a specific event may not be generated for the second time until all the others are triggered, but developers can adjust such probability. Whenever a new *abstract state* is found, a *special event sequence*, which tries to maximize the *Data Loss* failure revelation and includes a *stop-start* event, is injected and the presence of *Data Loss* problems is verified using oracles. Since a correctly handled *stop-start* event should not modify the activity state at all, the oracles check whether the activity GUI states before and after such *stop-start* events are the same.
An evaluation study conduced on 48 buggy Android applications of the benchmark provided by Riganelli et al. [1] demonstrates the effectiveness of the aforementioned technique in detecting *Data Loss* failures. In fact, this study shows that such technique is able to detect 82 out of 110 *Data Loss* problems included in the benchmark, 36 out of 58 *Data Loss* problems already reported to the app developers and many others not yet known.
This thesis is the result of joint work with Simone Paolo Mottadelli and it has the purpose of showing both the development and the evaluation work carried out for this technique. It is structured as follows: chapter 2 characterizes the *Data Loss* problem and its causes, showing different examples from real world applications; chapter 3 discusses the state of the art of Android testing, focusing itself on both the available state-of-the-art test case generation tools for Android apps and how *Data Loss* problems are dealt with nowadays; chapter 4 shows the steps followed for the integration of the novel technique into an existing tool, providing an overview of its functioning and describing its implementation; chapter 5 reports the results of the evaluation of this technique, showing its abilities in both exploring the applications under test and detecting *Data Loss* problems, and, finally, chapter 6 concludes this thesis briefly summarizing the contents of this work and mentioning future improvements.

CHAPTER 2

Android applications and Data Loss problems

In the last decade, mobile applications have gained importance and popularity for almost everyone all over the world. People use these applications to perform a very large amount of tasks like reading e-mails, surfing the net, listening to music, making payments and so on. They use apps both for fun and for work, entrusting them with confidential data such as work e-mails and banking passwords.
Among these mobile applications, in the last few years, Android apps have experienced a significant increase of popularity. At the beginning of 2016 existing Android apps in the Google Play Store were about 2 million and, at the end of 2017, that number increased to 3.5 million [2].
This chapter presents Android activities and provides an overview of their life cycle, emphasizing the importance of preserving the activity states in order to correctly handle *stop-start* events. Later, it also presents the concept of *Data Loss* problems, specifying what they are, how they manifest themselves and how pervasive they are in the world of Android applications.

2.1 Introduction to Android activities

Android apps are composed by *activities*, which implement their user interface (UI). Generally, an app can have more than an activity, each of which provides cohesive functionalities to the user and represents a full-screen window of the app containing widgets, such as *Buttons* or *EditTexts*, through which the users can interact with the application. These widgets can be tied to callback methods, which are invoked when a corresponding event is triggered, and are characterized by multiple properties like the position in coordinates on the screen, the class (e.g., *TextView*), whether they are clickable or not and so on. In fact, Android applications are event-driven systems and they are based on callback methods invoked by the *ART*, the Android Runtime Environment, in response to particular events. For example, when the user presses a button, the associated callback method is called and its code is executed.

All the activities run in the main thread of the app and each of them has its own life cycle controlled through six fundamental callback methods, as shown in Figure 2.1. The *onCreate*() callback method is called when the activity is

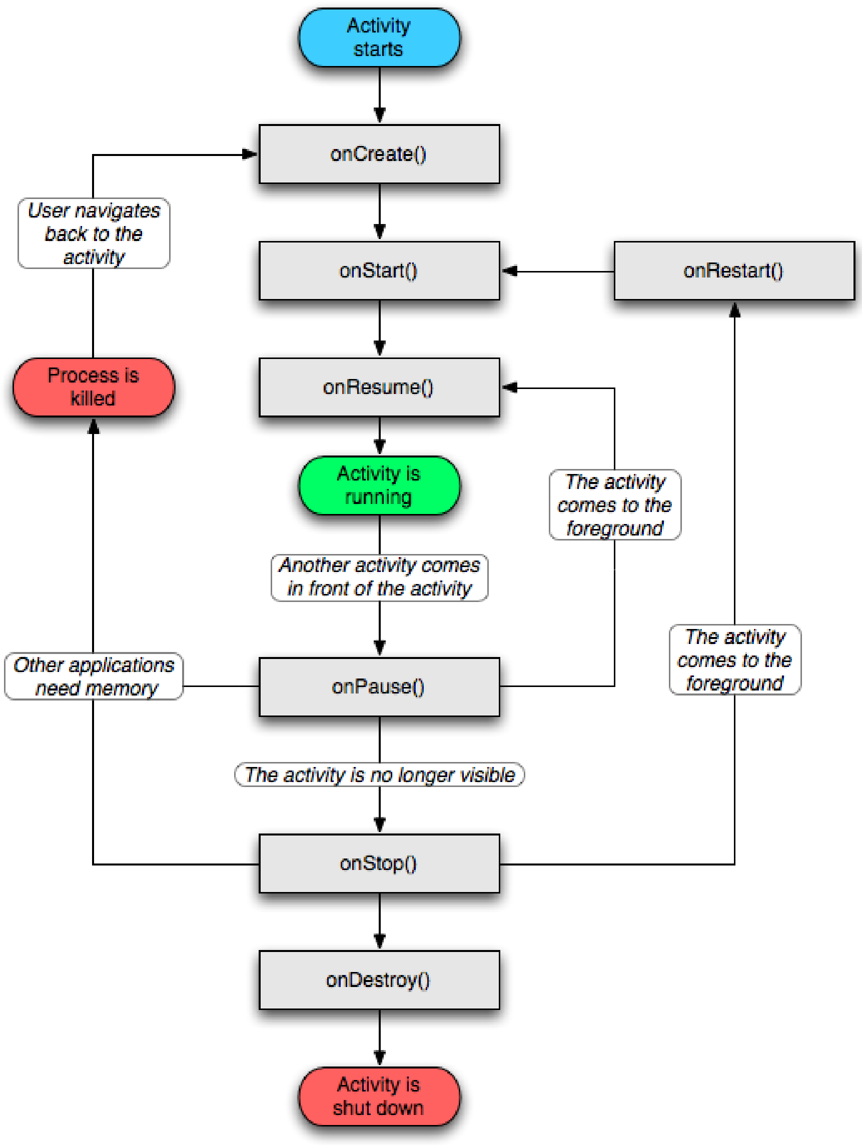

Activity starts
onCreate()
User navigates back to the activity
onStart()
onRestart()
Process is killed
onResume()
Activity is running
The activity comes to the foreground
Another activity comes in front of the activity
The activity comes to the foreground
Other applications need memory
onPause()
The activity is no longer visible
onStop()
onDestroy()
Activity is shut down

Figure 2.1: Activity lifecycle scheme

created for the fist time and thus enters the *Created* state. This callback method should contain statements to initialize the main resources used by the activity and takes a *Bundle* [3] object in input to restore a previously saved state, if the activity had already been created. Whenever the execution of this method ends, the *onStart*() callback method is invoked and the activity enters the *Started* state. The *onStart*() callback method should initialize the GUI content of the activity. When *onResume*() callback method is called, the activity enters the *Resumed* state, goes to the foreground and starts interacting with the user. The activity remains in this state until it loses the focus and this happens, for example, when

the user presses the “home” button or when an incoming call is received. In these cases, the *onPause*() callback method is invoked and the activity enters the *Paused* state. This method should release all those system resources that might be acquired by others when the activity is no longer in the foreground, such as camera sensors. The activity remains in this state either until it does not go back to the foreground or it becomes completely invisible to the user. In the latter case, the *onStop*() callback method is invoked and the activity enters the *Stopped* state. This method should complete heavy operations that were not performed in the previous state. The activity remains in this state until either it does not go back to the foreground or it terminates its execution. In the second case, the *onDestroy*() callback method is called and the activity enters the *Destroyed* state. Here, all the remaining resources not yet deallocated during the executions of the *onPause*() and the *onStop*() callback methods should be finally released.
The entire lifecycle of an activity begins when the *onCreate*() callback method is called by the *ART* and it ends after the execution of the *onDestory*().
These methods should be implemented by developers in order to guarantee that an activity works properly and to perform critical tasks, such as releasing resources. Since the available resources on mobile devices are limited, the Android operating system may kill the process of the application to free up memory. The likelihood that such event occurs depends on the state in which the activity resides and this probability is even higher if the activity is in the *Stopped* or *Destroyed* states.

2.2 Handling the activity state

During their entire lifecycle, a destroyed activity might be recreated in the future. This could happen both for actions performed by the users and for system needs. In fact, by default, the system automatically destroys and recreates the foreground activity in correspondence of a configuration change, because the activity needs to be adapted to the new configuration. For instance, when the device screen is rotated from *portrait* to *landscape* or when the language of the device is changed, all the widgets on that activity have to be adapted. In other contexts, the operating system is committed to preserve the activity state in the RAM, but sometimes this is impossible because more memory is needed to execute new tasks. For example, when the user navigates away from an activity, the Android operating system may kill that activity, destroying the application process in which it resides, and later recreate it when the user navigates back to it. Such events require mechanisms for saving and restoring the activity state to deal with the user expectations.
Preserving and restoring an activity state, which includes both its GUI state and its internal state, is a fundamental part for the user experience. The GUI state corresponds to the hierarchical structure of the widgets of an activity, that is, the way widgets are logically organized in the GUI, as well as to the properties of the widgets, such as the contents inside *EditTexts*. Instead, the internal state is constituted by the set of values assumed by its member variables.
There are contexts in which the activity state is expected to remain unchanged, and others in which it is expected to be cleared. For instance, if an activity is destroyed voluntarily, such as by pressing the “back” button or by killing the entire

app from the "Settings" app, the users think they have permanently navigated away from the activity and, hence, they do not expect to find the same GUI state if they return to that particular activity. Instead, in other contexts, for example when the "home" button is pressed or when the user switches among apps, the users expect to find the GUI state to remain unaltered.
Although the activity GUI state is preserved automatically by the *ART* under certain hypothesis (e.g., the widgets have the *android:id* property), this does not happen with its internal state. For this reason, Android developers have the responsibility to guarantee that the entire state is saved and restored throughout its destruction and recreation. This can be achieved either by overriding the callback methods *onSaveInstanceState*() and *onRestoreInstanceState*(), in order to serialize and deserialize small information on the disk, by using *ModelView* objects, which are used to save more complex objects in memory, or by using a local persistent storage, such a database or a file.
Since the usage of these mechanisms is more a good practice rather than an obligation, Android developers might not use them, making their applications unable to correctly handle events that require the destruction and the recreation of activities.

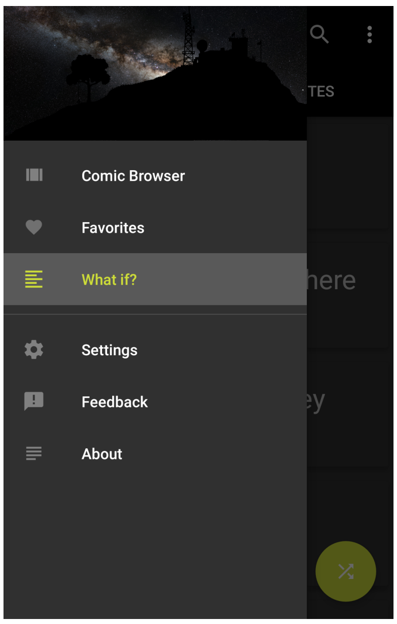
2.3 Data Loss problems

Android applications may be subjected to many *stop-start* events. A *stop-start* event requires stopping the execution of the app and later resuming it from the same point it was stopped. More precisely, it is defined as a sequence of events that makes the involved activity enter the *Stopped* state and then the *Resumed* state, regardless of the states crossed by the activity. For example, when an incoming call is received, the foreground activity goes to the *Stopped* state and it waits there until the call ends, thus returning to the *Resumed* state.
As introduced in section 2.2, there are contexts in which the involved activity is destroyed and recreated by default and others in which this might happen depending on system needs. In both cases, an incorrect management of these events can lead to unexpected behaviors due to *Data Loss* problems.
More in detail, *Data Loss* problems manifest themselves either in an inconsistency between activity GUI states or in application crashes, according to Riganelli et al. [4]. In fact, if the mechanisms for saving and restoring the activity state are not correctly used, when a *stop-start* event occurs the member variables of that activity may be reassigned to their default values, such as *null* or 0, with the consequence that unhandled runtime exceptions may be thrown, causing the app to crash suddenly (Figure 2.2). Since such problems may occur only when the activity is in particular states, it is very difficult to generate test event sequences able to reveal them.
According to Amalfitano et al. [5], activities affected by *Data Loss* problems can show one ore more of the following GUI failures:

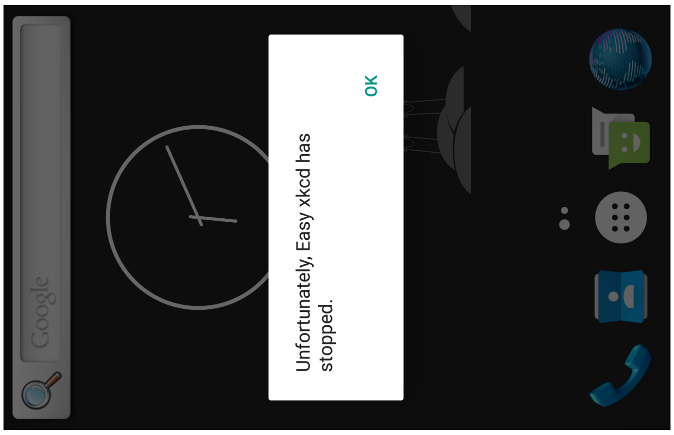
- GUI objects can disappear (Figure 2.3);
- GUI objects can appear (Figure 2.4);

- GUI objects can be displayed with wrong values (Figure 2.5).

These behaviours can be considered as *Data Loss* failures, except in those cases in which the developer has voluntarily programmed the app to behave in that way.
The figures 2.2, 2.3, 2.4 and 2.5 show four motivating examples of *Data Loss* failures observed after an orientation change. More in detail, Figure 2.2 represents the worst scenario in which the application crashes. Figure 2.3 shows a *Dialog* disappearing from the GUI, while Figure 2.4 shows a *Dialog* appearing. Finally, Figure 2.5 shows an *EditText* losing the data entered by the user.
Manifestations of *Data Loss* problems negatively affect the user experience and the usability of the app: the user is forced to reinsert the data lost after a *stop-start* event or to restart completely the application because it has crashed.

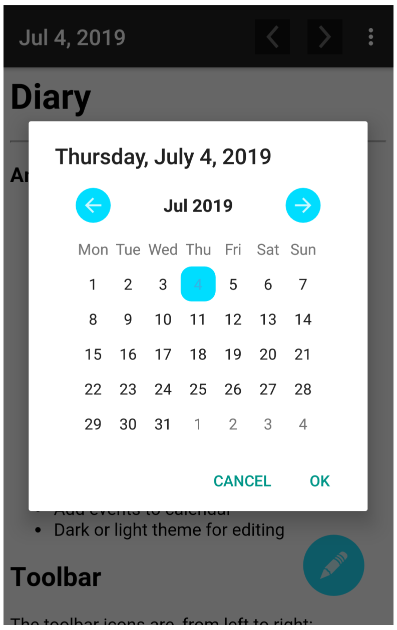


(a) Before a *stop-start* event


Unfortunately, Easy xkcd has stopped.

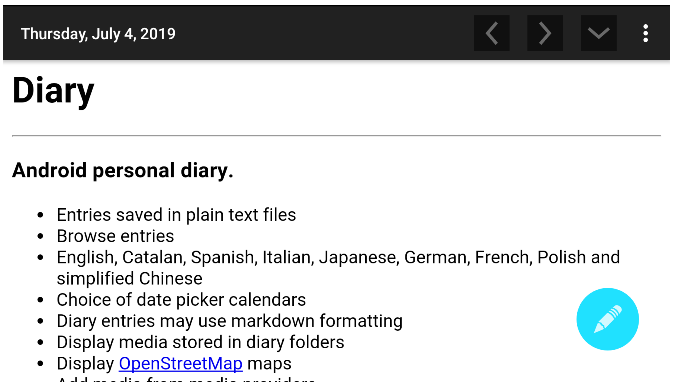
(b) After a *stop-start* event

Figure 2.2: A *Data Loss* failure that manifests itself in an application crash in Easy xkcd v6.0.4



Dark or light theme for editing

(a) Before a *stop-start* event

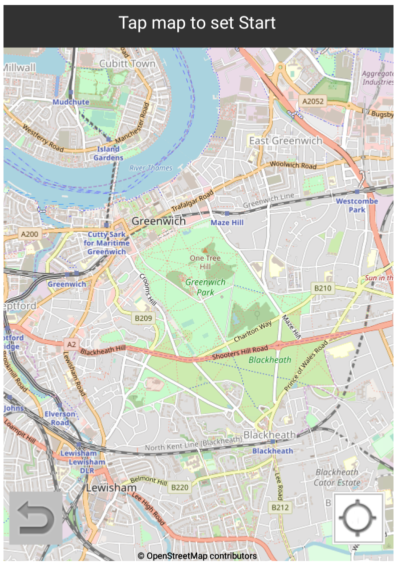


Android personal diary.

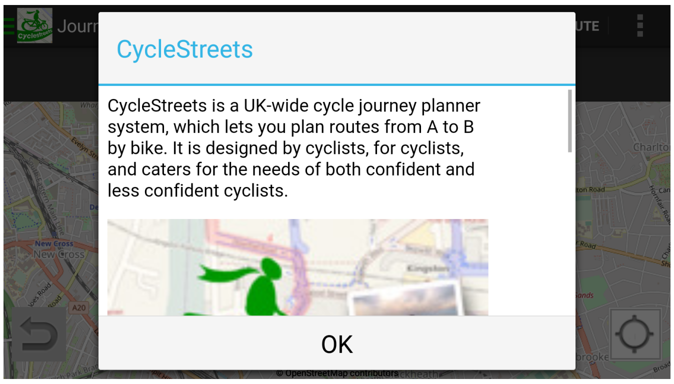
- Entries saved in plain text files
- Browse entries
- English, Catalan, Spanish, Italian, Japanese, German, French, Polish and simplified Chinese
- Choice of date picker calendars
- Diary entries may use markdown formatting
- Display media stored in diary folders
- Display OpenStreetMap maps

(b) After a *stop-start* event

Figure 2.3: A *Data Loss* failure that manifests itself in the disappearance of a *Dialog* in Diary v1.26



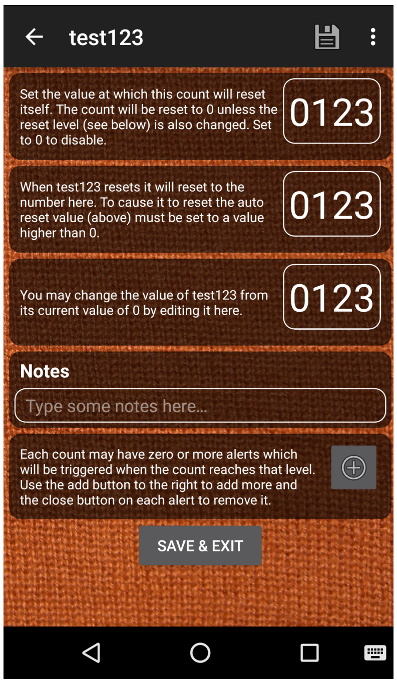
(a) Before a *stop-start* event



CycleStreets is a UK-wide cycle journey planner system, which lets you plan routes from A to B by bike. It is designed by cyclists, for cyclists, and caters for the needs of both confident and less confident cyclists.

(b) After a *stop-start* event

Figure 2.4: A *Data Loss* failure that manifests itself in the appearance of a *Dialog* in CycleStreets v3.5



Set the value at which this count will reset itself. The count will be reset to 0 unless the reset level (see below) is also changed. Set to 0 to disable.

0123

When test123 resets it will reset to the number here. To cause it to reset the auto reset value (above) must be set to a value higher than 0.

0123

You may change the value of test123 from its current value of 0 by editing it here.

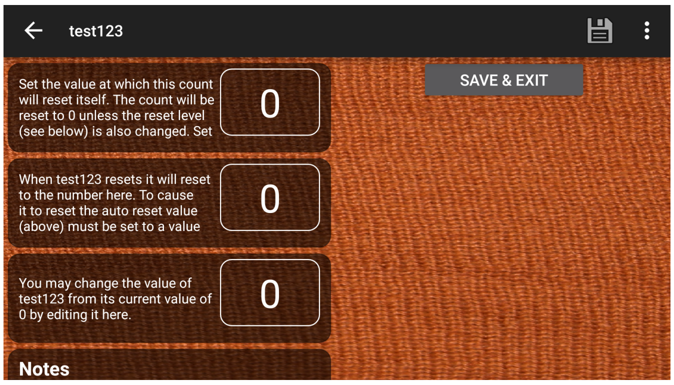
0123

Notes

Type some notes here...

Each count may have zero or more alerts which will be triggered when the count reaches that level. Use the add button to the right to add more and the close button on each alert to remove it.

(a) Before a *stop-start* event



Set the value at which this count will reset itself. The count will be reset to 0 unless the reset level (see below) is also changed. Set

0

When test123 resets it will reset to the number here. To cause it to reset the auto reset value (above) must be set to a value

0

You may change the value of test123 from its current value of 0 by editing it here.

0

Notes

(b) After a *stop-start* event

Figure 2.5: A *Data Loss* failure that manifests itself in the reappearance of an *EditText* with a wrong content in BeeCount v2.7.4

2.4 Pervasiveness of Data Loss problems

Data Loss faults can be introduced in an Android application very easily, hence, it is very common to find a large number of apps affected by them.
In accordance with Riganelli et al. [4], a *Data Loss* fault can be introduced in the code of an Android app in three ways:

- By a missing implementation of the callback methods provided by the Android framework. In this case, the developers forget to override *onSaveInstanceState*() and *onRestoreInstanceState*(), only relying on their default implementation and omitting to save and restore additional information of the activity state;
- By an incorrect overriding of the callback methods provided by the Android framework;
- By upgrades of the Android framework. Here, the provided callback methods are correctly overridden by the developers, but the continuous upgrades of the Android framework might make the application unable to handle *stop-start* events anyway.

Adamsen et al. [6] performed a similar study subjecting four Android applications to *stop-start* events in order to detect any failures. They concluded that all of the tested apps manifested a lot of failures, some of which could be considered *Data Loss* failures.

Finally, a recent study conducted by Riganelli et al. [1] underlines even more how pervasive *Data Loss* problems are. In particular, in the sample of applications used for their investigation, which was composed by 428 Android apps, 82 of them were affected by at least a *Data Loss* problem, which was nearly a fifth of their sample. Their study shows that these problems increase considerably the technical debt: the faults studied in their investigation would have required from one day to almost a year to be fixed.
These studies show both a great involvement by researchers in this field and a large diffusion of *Data Loss* problems in the world of Android applications.

CHAPTER 3

State of the art of Android testing

Like all software applications, Android apps should guarantee different qualities and meet their specifications. To achieve this, mobile apps need to be thoroughly tested, but, since they are event-driven systems, testing an Android app could be tricky and tedious. Android developers are used to writing test cases using testing automation frameworks or tools like *JUnit* [7], *Appium* [8], *Robotium* [9] or *Espresso* [10]. However, these still require a little effort from the developers, because, for example, they have to set up the testing environment, create test event sequences and design oracles.
This chapter provides an overview of both the main exploration approaches and the main tools available in the state of the art of Android testing. Finally, it presents how *Data Loss* problems are addressed nowadays.

3.1 State-of-the-art test case generation techniques

Android automation testing has become a research area of interest for many researchers, who developed different tools and techniques with the purpose of testing Android applications more efficiently, reducing considerably manual efforts in designing test cases and oracles. To reach this goal, these tools are able to generate input sequences, such as clicks or scrolls, in order to automatically explore the behaviours of the application under test. They aim to maximize both the code coverage and the number of application crashes during their exploration.
The currently available state-of-the-art test input generation tools mainly differ for their exploration strategy, according to Choudhary et al. [11]. In fact, these can be classified into:

- *Random* exploration strategy;
- *Model-based* exploration strategy;

- *Systematic* exploration strategy.

This section mentions the state-of-the-art test input generation tools analyzed for the decision of the tool to extend presented in section 4.1, without considering the others available in the research field.

3.1.1 Random exploration strategy

The *random* exploration strategy consists in the exploration of the app under test using user events and simple system events (e.g., SMS notifications) without any logic. However, this approach might be inefficient because the system can react only to a few of them and it does not consider the redundancy of the events already generated. In contrast, the tools that use this strategy are more effective in stressing the application rather then in maximizing its code coverage. They stop their explorations either when the application crashes or when they have reached the fixed number of events to be generated.
Monkey [12], the tool developed by Google, implements a random strategy and it is able to generate both user and system events. This tool allows the users to set the probability of each type of event to be triggered. The exploration strategy can also be restricted to a set of packages and it stops whenever the application crashes or the fixed number of events is reached. Since it is integrated in the Android SDK, it is one of the most used tool in industry, due also to its ease of use and its applicability to a vast category of apps.

3.1.2 Model-based exploration strategy

The *model-based* exploration strategy relies on GUI models, which are typically finite state machines having abstractions of activities as states and events to be triggered as transitions among states. These GUI models can either be built statically, analyzing both the source code and other files of the application, or dynamically, during the exploration of the app. The tools that use this strategy differ both in how they define the GUI model and in how they use it to generate events. Since this strategy can limit the number of redundant inputs, it should have better performance in terms of code coverage than the *random* one. Moreover, these tools stop their exploration either when all the events from all the discovered states conduce to states already visited, when a determined number of iterations is reached or when the allocated time for their execution ends. However, as Choudhary et al. [11] underline, the representation of the GUI states might not be sufficient because some events can change the internal application state without affecting the GUI, thus failing to consider those events relevant for the exploration.
DroidBot [13] is a tool that uses this exploration strategy. It is highly programmable and lightweight because it allows users to define their own exploration policies and it does not require any instrumentation of the app. As default, its GUI model is constructed on the fly during the exploration of the app and it explores its GUI model using *depth-first* search. More specifically, the GUI model is a direct graph in which the nodes are represented by the state of the device in terms of both services that are running in that moment and GUI objects displayed

on the screen.
Another tool is *Stoat* [14] which uses an evolutionary form of the *model-based* exploration strategy. This tool divides the exploration work in two phases. In the first one, it receives the *apk* file of the app in input and it constructs a probabilistic GUI model dynamically. During this first phase, *Stoat* [14] helps itself performing a static analysis to discover events that are not directly deducible from the dynamic one, looking for event listeners among the source files. The nodes of the GUI model are abstract states of the app, which represent the GUI hierarchy of an activity in that time instant, while the edges are labelled with a couple (event, probability), which represents the probability of the input event to be triggered in that specific abstract state. In the second phase, the model is iteratively mutated, perturbing the probabilities of the stochastic GUI model, in order to optimize its objective function that considers both the code coverage and the model coverage, as well as the diversity among test cases. Once the stochastic GUI model is mutated, the tool compares it with the previous model and decides whether to accept it based on its fitness score. If it is rejected, the previous model will be chosen for the next iteration. As soon as the tool reaches the provided number of iterations, the exploration ends.
A^3E Depth-first [15] is another *model-based* tool. Unlike the others, it uses a more abstract GUI model because the states are represented by activities. This technique mimics user actions to explore the app also considering the activity coverage, which is not taken in consideration by the previous tools. This tool creates a *dynamic activity transition graph* and it applies a *depth-first* search algorithm to thoroughly explore the behaviors of the app. All the generated events are recorded so that they are repeatable at the end of the exploration, allowing to reproduce application crashes encountered during the analysis.

3.1.3 Systematic exploration strategy

The *systematic* exploration strategy generates specific inputs in order to lead the exploration towards unexplored code. The tools based on this strategy can use static analysis to be aware about the structure of the application before the dynamic exploration. In this way, these tools can generate specific events to reach uncovered code, using advanced techniques like symbolic execution or evolutionary algorithms, and this should considerably increase the code coverage.
A^3E Targeted [15] is a systematic testing tool able to perform a fast exploration of the application activities. First, it analyzes the application's bytecode creating a *static activity transition graph* (SATG). Then, it uses this graph to guide the dynamic exploration towards a fast activity coverage. In fact, since there are activities that cannot be directly invoked using user events, because they are not designed to interact with the users, the approach exploits the SATG for starting and visiting these activities.
Sapienz [16] uses a genetic algorithm to optimize its fitness function, searching for the shortest input sequences able to maximize both the code coverage and the failure revelation. The population consists of test suites, which are composed by a set of test cases, which in turn are composed by test events. The evolutionary

algorithm first performs a selection operation, selecting the test suites with the best fitness score. Then, the test cases inside the test suites are shuffled and a single point crossover operation swaps the test cases between the test suites with a certain probability. After that, a mutation operation shuffles the order of the events inside the involved test cases with a certain probability. Unfortunately, this tool is no longer supported by its founders.

3.2 Data Loss problem detection nowadays

In the past years, different studies have been conduced to analyze and classify Android application failures manifested in correspondence of *stop-start* events. Android developers are used to writing test cases that subject their apps to expected sequences of events, which hardly occur in real situations. To address this problem, Adamsen et al. [6] developed *Thor* [6], a tool able to inject “neutral event sequences” into existing *Robotium* [9] or *Espresso* [10] test cases, whose goal is to expose the app under test to adverse conditions. These event sequences try to simulate real use cases in which, for example, the targeted application may be subjected to many *stop-start* events. The experimental evaluation of their tool on four real world Android apps shows how effective this approach is in revealing problems that are not directly highlighted by the original test cases. In fact, whereas these test cases initially had a positive outcome, the same ones modified by *Thor* [6] failed.
Amalfitano et al. [5] studied how double orientation changes, called *DOC* by them, may mess up the GUI of Android applications, classifying both the manifestations and the fault occurrences of these problems. In particular, they manually created the test suites, injected *DOC* events inside them and verified the presence of GUI failures comparing the GUI states before and after such *stop-start* events. Once observed the GUI failures, they investigated on the faults that caused them and classified both the detected behaviours and the class of widgets involved. Their investigation concerned open-source and industrial Android apps, showing that most of them were not able to correctly handle *DOC* events. In conclusion, they affirmed that *Dialogs*, *ListViews* and *ScrollViews* are the most common GUI objects involved in these types of failures, since they often disappear, change their properties or appear in wrong positions.
The tools described in the section 3.1 are designed to perform functional testing while generating test sequences to maximize code coverage and, as underlined by Zaeem et al. [17], they do not directly address the problem of oracles. Despite their ability to reveal unique crashes and obtain high code coverage in a short time, the absence of oracles does not allow them to be used for detecting failures that do not lead to application crashes.
Although *Thor* [6] is able to test Android apps in adverse conditions, it does not work without providing it test cases already implemented and, hence, its effectiveness depends on the quality of such test cases.
To address *Data Loss* problems that affect Android applications, Riganelli et al. [4] developed *DataLossHealer* [4], a tool capable of detecting and healing such

problems while the app is running. This tool detects changes between activity states before and after *stop-start* events and thus intervenes on the variables that lost their values, restoring them correctly. Since *DataLossHealer* [4] simply heals specific occurrences of *Data Loss* problems in a transparent manner to the final users, these have the feeling that the applications they are using are bug free. However, the limitation of this tool is that it does not act at fault level and, as a consequence, it does not directly fix these problems, leaving the affected applications buggy.
In conclusion, nowadays there are no tools able to detect *Data Loss* problems while exploring automatically the app behaviours, as well as able to help developers to localize and to fix such problems. For these reasons, the need of such tool arises.

CHAPTER 4

Development

The automatic test case generation technique described in this thesis work aims to detect *Data Loss* failures while dynamically exploring the Android application under test. Since the proposed technique needs to interact with the Android framework to inject both user and system events into the device, it has been integrated in *DroidBot* [13], a state-of-the-art test input generation tool for Android testing, which already implements these features.
This chapter shows the steps followed for the decision of *DroidBot* [13] as the tool to be extended among all the available tools. In addition, it explains the approach used both for the exploration of the app and for the *Data Loss* failure detection, presenting its implementation details at a high level.

4.1 The decision of the tool

The decision of the tool to extend with the technique described in section 4.2 has been taken after a high level analysis based on the study conducted by Wang et al. [18], who compared the main state-of-the-art test case generation tools available described in chapter 3.
The goal of this high level analysis was to find the most extensible tool and, to achieve this, each of them has been studied considering the following aspects:

- availability of the documentation;
- programming languages used for the implementation;
- code quality;
- exploration strategy;
- ease of use.

More in detail, such analysis consisted of two selection phases:

1. Analysis of both the available documentation associated with the tools and their source code with a high level of detail and execution of each tool to observe their ease of use and their functioning at runtime;

2. Detailed analysis of the source code in order to find the right place in the code to insert the extension for the detection of *Data Loss* problems.

If a tool did not pass the first phase, then it would not have been considered for the second one. The results of this high-level analysis are shown in Table 4.1.
Initially, each tool was downloaded from its respective GitHub page and the first one to be analyzed was *Monkey* [12]. It is completely written in Java and, at

Name	Monkey [12]	Sapienz [16]	Stoat [14]	DroidBot [13]	A^3E [15]
Documentation	✗	✓	✓	✓	✓
Programming language	Java	Python	Python, Java	Python	Ruby, Java
Code quality	Well commented, well organized, code understandable	Few comments, well organized and code understandable	No comments, bad organized, code hard to understand	No comments, well organized, code understandable	No comments, well organized, code hard to understand
Exploration strategy	Pseudo-random	Evolutionary	Model-based evolutionary	Model-based	Systematic
Ease of use	Easy, well explained			Very easy, well explained	

Note:
The **Programming language**s are ordered according with the GitHub code percentages;
The **Code quality** is based only on subjective opinions, considering the code readability, the names used for variables, methods and classes and the project organization into files and packages;
The **Ease of use** is also based on subjective opinions, considering the quality of the User Manual and the time spent to set up the tool.

Table 4.1: High-level analysis of the main state-of-the-art test input generation tools

first sight, its code seems of good quality. In fact, it is well organized and easy to understand, thanks to both the presence of many comments and the explanatory names assigned to variables, methods and classes. In addition, as underlined by Wang et al. [18], *Monkey* [12] is the best tool in terms of activity coverage. In contrast to the others, it was not developed in academic contexts and, for this reason, no documentation describing how it was designed and implemented was found.
Sapienz [16] is written in Python and, even though it is not well commented, its source code is understandable and well organized. Moreover, it reaches a high activity coverage, slightly less than *Monkey* [12].
Stoat [14] is written both in Python and in Java. Although its technique is well described in its documentation, its implementation is not very easy to understand, because its code is bad organized and it does not include any comments.
DroidBot [13], like *Sapienz* [16], is written in Python. The readability of its source code is the highest among all and the project structure is well organized.
A^3E [15] is mainly written in Ruby. Its source code is well organized, but, nevertheless, very hard to read due to the absence of comments and the presence of

variable and method names not very significant.
Then, each tool was evaluated in their ease of use and their functioning at runtime. Unfortunately, during their executions, *Sapienz* [16], *Stoat* [14] and *A^3E* [15] showed different problems that could not be easily solved even with the help of their founders. For this reason, such tools did not pass this first selection phase and the only ones left for the second one were *Monkey* [12] and *DroidBot* [13].
Monkey [12] is integrated in the Android SDK and it is very easy to use. However, it does not seem extensible because, after spending many hours analyzing its source code, no extension points could be found. Moreover, it was designed to stop as soon as the application crashed, but such behaviour is undesirable for the goals of the developed technique, whose aim is to maximize the number of *Data Loss* failure to be detected.
Instead, *DroidBot* [13] looks like a programmable and highly extensible tool, as also mentioned by its creators. In fact, the whole exploration strategy can be changed just by implementing a new version of the method that generates the events. It is very easy to use, as its functioning is well described on its GitHub page. Besides, the oracles for the *Data Loss* failure detection can be easily introduced in the exploration process.
Since only *DroidBot* [13] passed all the selection phases, the proposed technique has been integrated in it.

4.2 Approach overview

The developed technique has the purpose of maximizing both the activity coverage and the number of *Data Loss* failure revelations. The reason why the activity coverage is preferred to the other metrics (e.g., code coverage or method coverage) is because it is important to test how activities handle their destruction and recreation in correspondence of *stop-start* events.
To achieve these goals, the technique explores the apps using a *model-based* exploration strategy (see subsection 3.1.2), leveraging on a GUI model created at runtime, and it heuristically generates both *special event sequences* and *stop-start* events.
The GUI model is a non-deterministic finite automaton (NFA), which is formally defined as a tuple (Q, Σ, q_0, δ, F). Q is the finite set of *abstract states*, which are abstractions of activity GUI states. Σ is the finite set of events e that can be triggered from such *abstract states*, such as clicks on buttons, swipes or *stop-start* events. $q_0 \in Q$ is the first *abstract state* encountered as soon as the exploration begins. $\delta : Q \times \Sigma \rightarrow 2^Q$ is the transition function, which, given $q \in Q$ and $e \in \Sigma$, returns the set of *abstract states* reachable from q triggering e. $F = Q$ is the set of final *abstract states*. Since all *abstract states* are accepting states, no event sequences are rejected.
The GUI model is defined as a NFA because an event $e \in \Sigma$ triggered from an *abstract state* $q \in Q$ does not always lead to the same *abstract state*. For instance, this might happen if the tested application is non-deterministic.

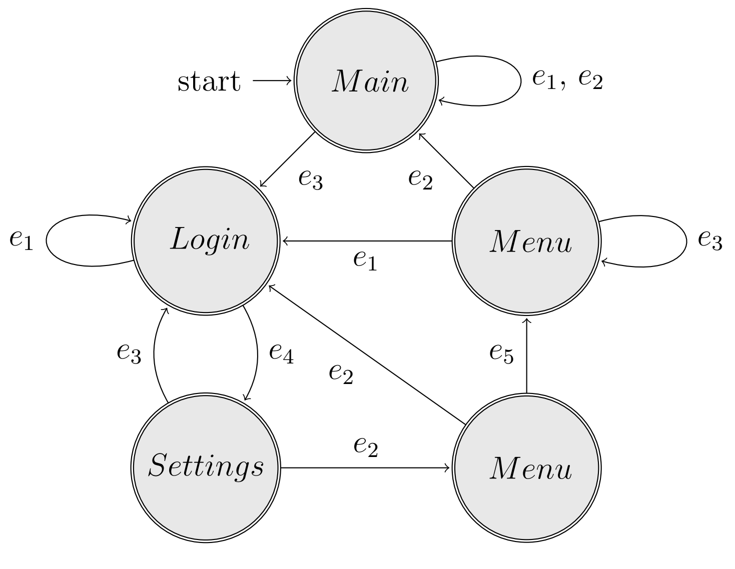


Figure 4.1: Example of the GUI model

Definition 4.2.1. An *abstract state* $q \in Q$ is a pair (a, E), where a is an activity name and $E \subseteq \Sigma$ is an ordered set of events that can be triggered from q.

Definition 4.2.2. Let $q = (a, E)$ and $q' = (a', E')$ be two *abstract states*, where $E = (e_1, e_2, ..., e_n)$ and $E' = (e'_1, e'_2, ..., e'_n)$, then:

$$q \equiv q' \Leftrightarrow a = a' \land e_i = e'_i \; \forall i \in \{1, ..., n\}$$

Property 4.2.1. Given the GUI model (Q, Σ, q_0, δ, F) then $\nexists \, (q, q') \in Q^2$ such that $q \equiv q'$.

Figure 4.1 shows an example of GUI model, where $Q = \{(Main, (e_1, e_2, e_3))$, $(Login, (e_1, e_4))$, $(Settings, (e_2, e_3))$, $(Menu, (e_2, e_5))$, $(Menu, (e_1, e_2, e_3))\}$ and Σ $= \{e_1,\ e_2,\ e_3,\ e_4,\ e_5\}$.
This GUI model is dynamically generated at runtime: initially, it is composed just by an *abstract state*, which corresponds to the first *abstract state* encountered as soon as the exploration begins. Then, if a triggered event leads to a new *abstract state*, the latter is added to the GUI model. When a new *abstract state* is discovered, a *special event sequence*, containing a *stop-start* event, is injected to maximize the *Data Loss* failure revelation in the current activity. More specifically, a *special event sequence* alters the properties of GUI objects, such as the contents inside *EditTexts*, and then injects a *stop-start* event, which is necessary to reveal *Data Loss* problems. In fact, after every *stop-start* event, regardless of whether it is triggered from a *special event sequence* or not, an oracle verifies that the properties of the GUI objects remained the same.
Generally, given an *abstract state*, it is preferable to generate different events every time in order to discover new *abstract states*, thus favoring the exploration of the app. However, there are cases in which this behaviour is undesired because a specific event might be a key element to discover new activities. For instance, when a press on a button opens a *NavigationDrawer* and a new *abstract state* is found, the *stop-start* event injected within the *special event sequence* might close the just

opened *NavigationDrawer*, which could be the only way to access those activities not yet explored, leading to the previous *abstract state*. In this situation, it would be desirable to press that button another time without waiting for all the other events to be generated.
For this reason, every event $e \in \Sigma$ has a probability $Prob(e)$ to be triggered from a certain *abstract state*. An event already injected has less likelihood to be generated than another one not yet triggered and this probability can be adjusted with the parameter $\epsilon \in [0, 1]$.

Definition 4.2.3. Let:

- $q = (a, E) \in Q$ be an *abstract state*;
- $e \in E \subseteq \Sigma$ be an event that can be triggered from q;
- $N \subseteq E$ be the set of events not yet triggered from q;
- $\epsilon \in [0, 1]$ be a parameter.

Then, $Prob(e)$ is the probability of e to be triggered from q and it is defined as follows:

$$Prob(e) = \begin{cases} \epsilon * \frac{1}{|E|} & \text{if } e \notin N \\ \epsilon * \frac{1}{|E|} + (1 - \epsilon) * \frac{1}{|N|} & \text{if } e \in N \end{cases}$$

The exploration stops when the fixed number of transitions, which is defined a priori by the user, is reached.

4.3 Implementation

After choosing to extend *DroidBot* [13], its source code was analyzed more accurately to better understand the functionalities needed to develop the proposed technique. In particular, besides the entire exploration strategy, oracles and other types of events have been implemented.

4.3.1 Events

By default, *DroidBot* [13] already provides a set of events to interact with the device. Such events are abstracted using Python classes, each of which implements a different version of the *send*() method to correctly trigger a specific type of event on the device, using different Android Debug Bridge (*ADB*) [19] commands.
Among all these events, the only ones used for the developed exploration strategy are the following:

- *TouchEvent*, used to simulate a tap;
- *LongTouchEvent*, used to simulate a longer tap;
- *SetTextEvent*, used to write text inside an editable view;

- *ScrollEvent*, used to simulate a swipe;
- *KeyEvent*, used to simulate a press on a navigation button (e.g. "Back" or "Home").

Since the default version of *DroidBot* [13] does not provide the possibility to generate *stop-start* events, which are essential to reveal *Data Loss* problems, the screen rotation event has been implemented. Although a single orientation change of the device screen is sufficient to generate a *stop-start* event, the latter does not allow to have comparable activity GUI states. In fact, the GUI objects in "portrait" mode are different from the same ones in "landscape" mode, making it impossible to determine whether a *Data Loss* failure has occurred. For this reason, it has been necessary to generate two orientation changes with the aim of both generating a *stop-start* event and allowing the comparison of the activity GUI states.
The double orientation change of the device screen has been abstracted with the *DoubleRotationEvent* class, which allows to atomically execute two consecutive screen rotation events. Like in all the other events, the *send*() method uses *ADB* commands to trigger these configuration changes. These commands are executed in the following order:

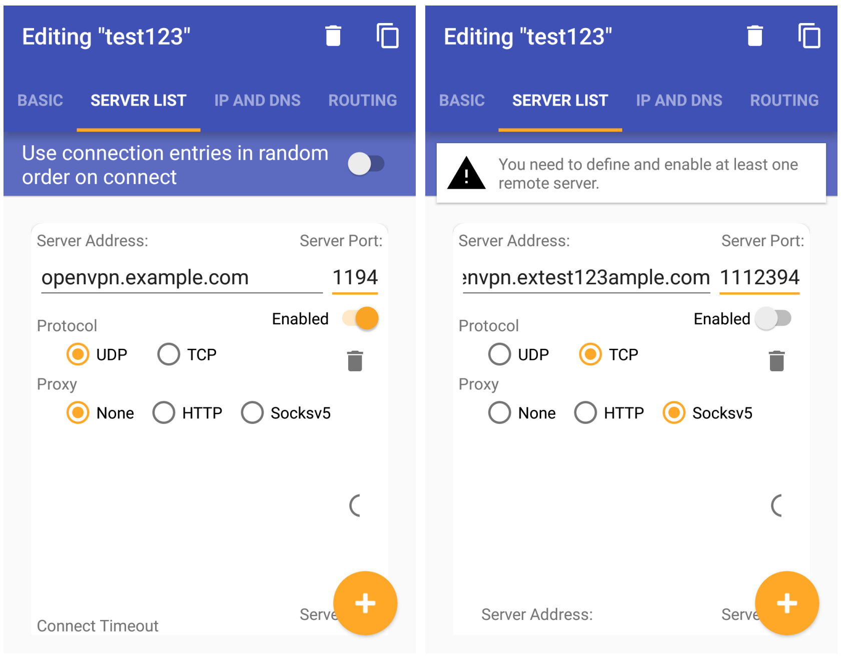
1. adb shell settings put system user_rotation 1
2. adb shell settings put system user_rotation 0

The first command sets the device orientation screen mode to "landscape", while the second one sets it back to "portrait".
The key element to maximize the *Data Loss* failure revelation is the *FillAllEvent* class. Given the set of views that compose the activity GUI state at a certain moment, a *FillAllEvent* first filters the views, selecting only the ones that are editable and clickable, such as *EditTexts* or *ToggleButtons*, and then triggers the corresponding events on them, as shown in Figure 4.2. For instance, a *TouchEvent* is generated on a *RadioButton* or a *TouchEvent* followed by a *SetTextEvent* is sent to an *EditText*. In the latter case, the *SetTextEvent* writes "test123" inside an *EditText* in order to cover the cases in which it is allowed to write only numeric values or only not numeric ones. Its *send*() method implementation is shown in Algorithm 1.
The *FillAllEvent* should be "safe" because it is not supposed to change the current *abstract state*. Therefore, views belonging to the *Button* class are not considered as clickable, because they may change the current *abstract state* with high probability.
Since the default implementation of the *ScrollEvent* class allowed only to specify the starting coordinates of a swipe event, this event has been extended with the possibility to specify also its ending coordinates. Thanks to this extension, a swipe can be performed starting from one side of the screen to the opposite one, allowing to improve even more both the *Data Loss* failure revelation and the exploration strategy. For example, a swipe from the bottom to the top of the device screen can be useful for those activities whose GUI objects are not all visible at the same time (see Figure 4.3), or a swipe from the leftmost side to the rightmost side of the

Algorithm 1 Algorithm to generate and send the events used by a *FillAllEvent*

```
$events \leftarrow \emptyset$
$views \leftarrow$ getViewsFromDevice()
for all $view \in views$ do
    if $view \in EditableClasses$ then
        $events \leftarrow events \cup$ SetTextEvent(“test123”, $view$)
    else if $view \in ClickableClasses$ then
        $events \leftarrow events \cup$ TouchEvent($view$)
    end if
end for
for all $event \in events$ do
    $event$.send()
end for
```

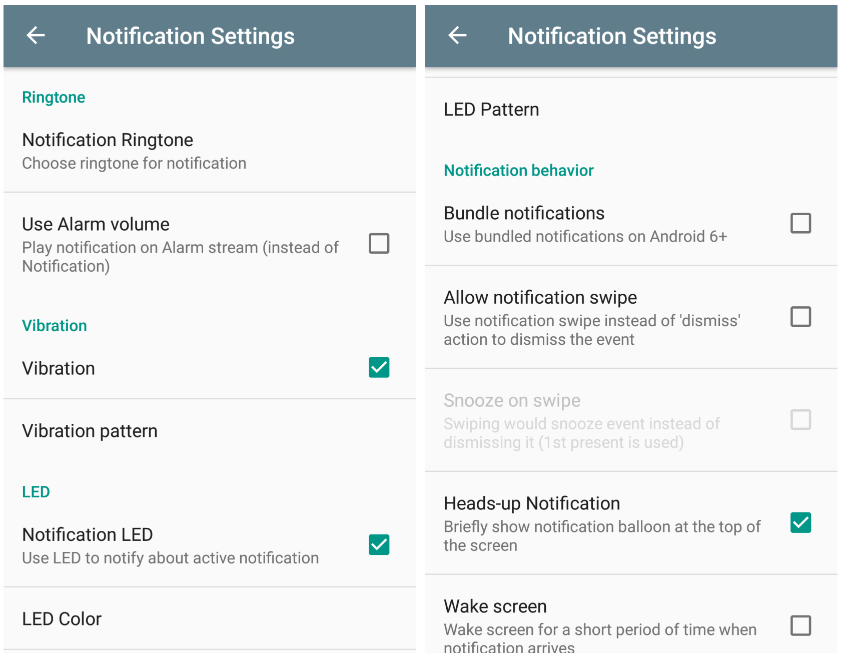
screen, or vice versa, can reveal a *NavigationDrawer*, which may not be otherwise accessible (see Figure 4.4), increasing the activity coverage.
In conclusion, all the events have been provided with a *createUniqueCode*() method, which internally uses the hash function MD5 to generate an alphanumeric sequence that uniquely identifies them. This method has been implemented in two versions: the first one generates the code based on the properties of the target view, while the second one bases itself only on the properties of the event class, as there are event classes without any view attached (e.g., the *DoubleRotationEvent* class).


Editing "test123"
BASIC | SERVER LIST | IP AND DNS | ROUTING
Use connection entries in random order on connect
Server Address: openvpn.example.com
Server Port: 1194
Enabled [✓]
Protocol: [✓ UDP] TCP
Proxy: [✓ None] HTTP Socksv5
Connect Timeout
Serve

Editing "test123"
BASIC | SERVER LIST | IP AND DNS | ROUTING
You need to define and enable at least one remote server.
Server Address: ɘnvpn.extest123ample.com
Server Port: 1112394
Enabled
Protocol: UDP [✓ TCP]
Proxy: None HTTP [✓ Socksv5]
Server Address:
Serve

(a) Before a *FillAllEvent* (b) After a *FillAllEvent*

Figure 4.2: A *FillAllEvent* writes inside every *EditText* and presses every *ToggleButton* and *RadioButton* in OpenVPN for Android v0.7.5



Notification Settings

Ringtone

Notification Ringtone
Choose ringtone for notification

Use Alarm volume ☐
Play notification on Alarm stream (instead of Notification)

Vibration

Vibration ✓

Vibration pattern

LED

Notification LED ✓
Use LED to notify about active notification

LED Color

Notification Settings

LED Pattern

Notification behavior

Bundle notifications ☐
Use bundled notifications on Android 6+

Allow notification swipe ☐
Use notification swipe instead of 'dismiss' action to dismiss the event

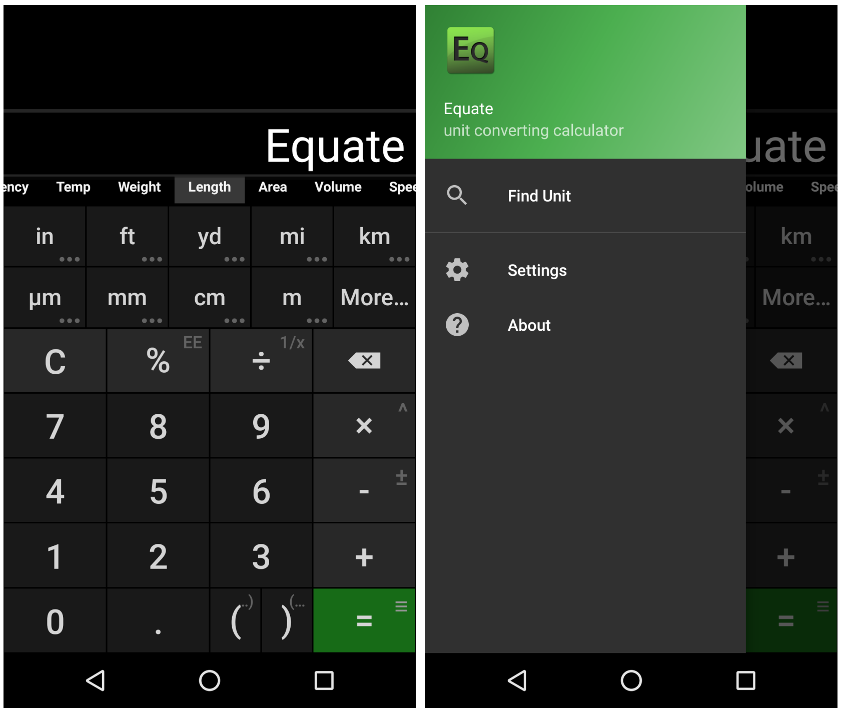
Snooze on swipe ☐
Swiping would snooze event instead of dismissing it (1st present is used)

Heads-up Notification ✓
Briefly show notification balloon at the top of the screen

Wake screen ☐
Wake screen for a short period of time when notification arrives

(a) Before a *ScrollEvent* (b) After a *ScrollEvent*

Figure 4.3: A *ScrollEvent* from the bottom to the top in Calendar Notifications v3.14.159 shows the views that first were hidden



(a) Before a *ScrollEvent* (b) After a *ScrollEvent*

Figure 4.4: A *ScrollEvent* from the leftmost side to the rightmost side opens a *NavigationDrawer* in Equate v1.6

```
[
    ...,
    {
        'content_description': None,
        'resource_id': None,
        'text': 'Editing "test123"',
        'visible': True,
        'checkable': False,
        'children': [],
        'size': '720*81',
        'checked': False,
        'temp_id': 4,
        'selected': False,
        'child_count': 0,
        'content_free_signature': '[class]android.widget.TextView[resource_id]None',
        'is_password': False,
        'parent': 3,
        'focusable': False,
        'editable': False,
        'focused': False,
        'clickable': False,
        'class': 'android.widget.TextView',
        'scrollable': False,
        'package': 'de.blinkt.openvpn',
        'long_clickable': False,
        'view_str': 'a209096d5371003c3865b683290f620f',
        'enabled': True,
        'bounds': [[48, 115], [768, 196]],
        'signature': '[class]android.widget.TextView[resource_id]None[text]Editing "test123"[enabled,,]'
    },
    ...
]
```

Figure 4.5: The list of Python dictionaries describing the GUI state of the activity shown in Figure 4.2a.

4.3.2 Oracles

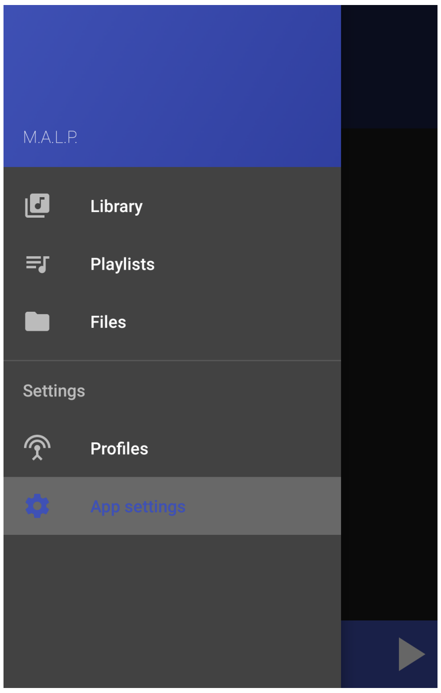
Data Loss failures can cause application crashes and changes in the GUI state of an activity, as described in section 2.3.

Whereas the detection of app crashes can be performed analyzing the outputs of *Logcat* [20], whose purpose is to dump system messages, changes in the GUI state of an activity after a *stop-start* event can not be detected with the same mechanisms, but require specific oracles.

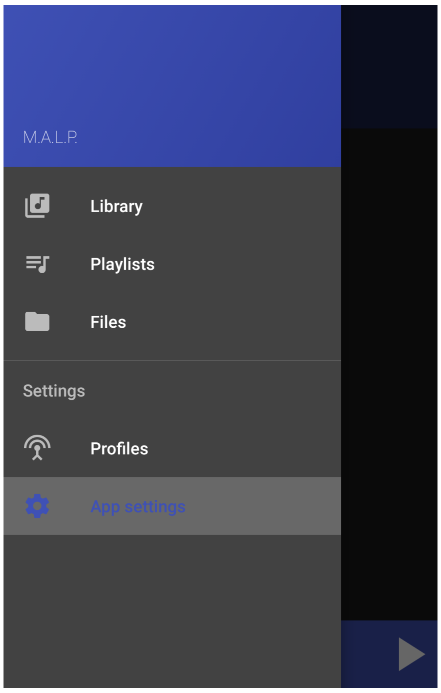
Since changes in the view properties that compose an activity may manifest themselves either with GUI failures or not, two types of oracles have been implemented, each of which takes and uses two different snapshots of an activity GUI state to determine whether a *Data Loss* failure has occurred.

The *property-based oracle* covers the cases in which changes in the view properties do not reflect themselves in graphical changes directly visible to the users. The snapshot of this oracle is a list of Python dictionaries, which provides a description of both the view hierarchy and the set of view properties, as illustrated in Figure 4.5, where only a view has been reported to avoid overloading the discussion. In order to detect *Data Loss* failures, it simply compares the snapshots taken before and after a *DoubleRotationEvent*. Figure 4.6 shows a *Data Loss* failure that can be detected only by using the *property-based oracle*. In particular, from how it can be deduced from the two figures, the *Data Loss* failure has not manifested itself graphically: the images are identical. However, what has changed in their snapshots is the value contained in the property "content_description".

Instead, the *screenshot-based oracle* aims to detect those activity GUI state changes that manifest themselves graphically without directly affecting the view properties or the view hierarchy. In contrast to the other oracle, this one considers the



(a) Before a *DoubleRotationEvent*



(b) After a *DoubleRotationEvent*

```
[
    ...,
    {
        'content_description': 'Close navigation drawer',
        ...
    },
    ...,
]
```

(a) Before a *DoubleRotationEvent*

```
[
    ...,
    {
        'content_description': 'Open navigation drawer',
        ...
    },
    ...,
]
```

(b) After a *DoubleRotationEvent*

Figure 4.6: A *Data Loss* failure that can be detected only by using the *property-based oracle* in MALP 3d31062

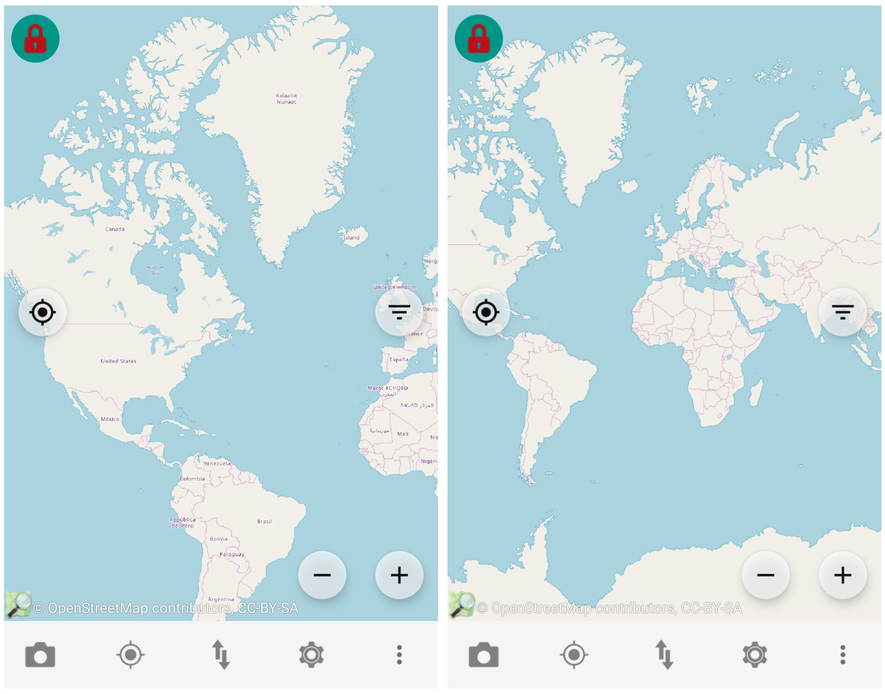
screenshot of the device screen as a snapshot and uses Algorithm 2 to detect *Data Loss* failures. More specifically, the *Screenshot* class encapsulates this algorithm, which takes in input a threshold value and two images representing the screenshots before and after a *DoubleRotationEvent*, which are obtained using the *ADB* command "adb shell exec-out screencap -p". The two images are first converted in gray scale and then cropped. The first operation speeds up the comparison procedure while the second one has the purpose of decreasing the number of false positives generated. In fact, the crop operation shrinks the image size in order to discard the top and the bottom of the device screen, as they display information

Algorithm 2 Algorithm to compare two images used by the *Screenshot* class

```
Input: $img1, img2$: two image snapshots of an activity GUI state
Input: $threshold$: threshold value used for the comparison
Output: $true$ if $img1$ and $img2$ are equal, $false$ otherwise

Require: $img1$ and $img2$ are the same size
Require: $0 \leq threshold \leq 1$
  $img1 \leftarrow$ convertToGrayScale($img1$)
  $img2 \leftarrow$ convertToGrayScale($img2$)
  $img1 \leftarrow$ cropImage($img1$)
  $img2 \leftarrow$ cropImage($img2$)
  $thresholdInPixels \leftarrow$ height($img1$) $*$ wight($img1$) $*$ $threshold$
  $differentPixels \leftarrow$ countDifferentPixels($img1$, $img2$)
  if $differentPixels < thresholdInPixels$ then
    return $True$
  else
    return $False$
  end if
```

that varies over time, such as the battery level or the time. In conclusion, the images are compared pixel by pixel and they are equal if the number of different pixels is less or equal than a value that depends on the threshold. Note that the threshold value is necessary to avoid false positives generated by the blinking of the cursor, or other similar scenarios, and by default it is set to 0.0002.
Figure 4.7 shows an example of *Data Loss* failure that can be detected only by using the *screenshot-based oracle*. In fact, the lists of Python dictionaries, describing the activity GUI states, remain unchanged before and after a *DoubleRotationEvent*, in contrast to the screenshots, which show a loss in the zoom position.
As already mentioned, these oracles can detect different types of *Data Loss* failure and, hence, it is necessary to simultaneously use them in correspondence of each *stop-start* event. In particular, a *Data Loss* failure is detected if one of them fails.


© OpenStreetMap contributors, CC-BY-SA
© OpenStreetMap contributors, CC-BY-SA

(a) Before a *DoubleRotationEvent* (b) After a *DoubleRotationEvent*

Figure 4.7: A *Data Loss* failure that can be detected only by using the *screenshot-based oracle* in Vespucci Osm Editor v10.2

4.3.3 Exploration strategy

The exploration strategy tries to maximize both the activity coverage and the number of *Data Loss* failure revelations. As described in section 4.2, the GUI model abstracts the concept of activity GUI state using *abstract states*, which have been implemented with the *AbstractState* class.
An *AbstractState* represents a specific GUI state of an activity and it is identified by the activity name as well as by the ordered set of events that can be triggered from the abstracted activity GUI state. The set of events is obtained by parsing

```
[
    ...,
    {
        'unique_code': 'e9694813e8c57b3aa33621e4e5cc3c9e',
        'triggered': False
    },
    {
        'unique_code': '920d005452363205c8a13700f2aa9f9d',
        'triggered': True
    },
    ...
]
```

Figure 4.8: Example of how events are stored inside an *AbstractState*

the same list of Python dictionaries used by the *property-based oracle* (see Figure 4.5). More precisely, it is passed in input to Algorithm 3, which analyzes it and generates the ordered set of events in output. As shown in Figure 4.8, an *AbstractState* keeps track of which events have already been generated, by marking them

Algorithm 3 Algorithm to create the events from an activity GUI state

```
Input: $views$: a Python dictionary describing an activity GUI state
Output: $events$: an ordered set of events

for all $view \in views$ do
    if $view$ is not visible $\vee$ $view$ is not enabled then
        $views \leftarrow views - view$
    end if
end for
$events \leftarrow \emptyset$
for all $view \in views$ do
    if $view$ is clickable then
        $events \leftarrow events \cup$ TouchEvent($view$)
    end if
    if $view$ is scrollable then
        $events \leftarrow events \cup$ ScrollEvent($view$, UP)
        $events \leftarrow events \cup$ ScrollEvent($view$, DOWN)
        $events \leftarrow events \cup$ ScrollEvent($view$, LEFT)
        $events \leftarrow events \cup$ ScrollEvent($view$, RIGHT)
    end if
    if $view$ is checkable then
        $events \leftarrow events \cup$ TouchEvent($view$)
    end if
    if $view$ is longclickable then
        $events \leftarrow events \cup$ LongTouchEvent($view$)
    end if
    if $view$ is editable then
        $events \leftarrow events \cup$ SetTextEvent($view$)
    end if
end for
$events \leftarrow events \cup$ ScrollEvent($view$, FULLLEFT)
$events \leftarrow events \cup$ ScrollEvent($view$, FULLRIGHT)
$events \leftarrow events \cup$ KeyEvent(BACK)
$events \leftarrow events \cup$ DoubleRotationEvent()
return $events$
```

as either *triggered* or *not triggered*. For reasons of efficiency, such events are stored with their corresponding unique codes obtained using their *createUniqueCode*() method. In addition, the *AbstractState* class also provides two methods: the first one returns the entire set of event codes, while the second one is used to get just the codes of the events not yet triggered. As soon as all the events in an *AbstractState* are marked as *triggered*, all of them are newly labelled as *not triggered*.
The entire exploration strategy is represented by the *DataLossPolicy* class, which generates events used both to explore the application under test and to reveal *Data Loss* problems detected using the oracles described in subsection 4.3.2. Figure 4.9 shows the work flow of the developed policy. Moreover, Algorithm 4 shows how the

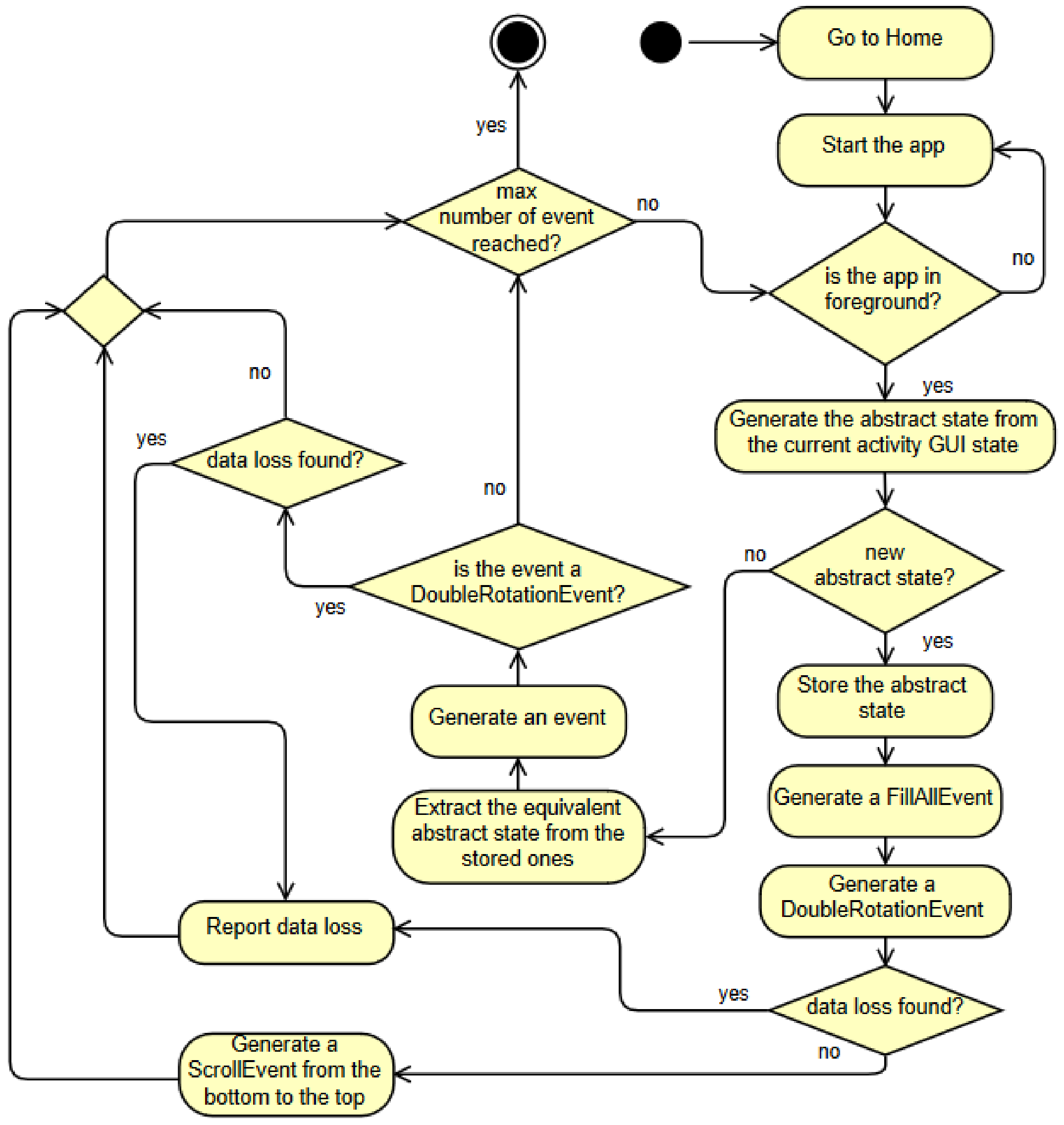

Go to Home
Start the app
is the app in foreground?
no
yes
Generate the abstract state from the current activity GUI state
new abstract state?
no
yes
Store the abstract state
Generate a FillAllEvent
Generate a DoubleRotationEvent
data loss found?
yes
no
Extract the equivalent abstract state from the stored ones
Generate an event
is the event a DoubleRotationEvent?
no
yes
data loss found?
no
yes
Report data loss
Generate a ScrollEvent from the bottom to the top
max number of event reached?
yes
no

Figure 4.9: *DataLossPolicy* diagram

event sequences are generated and how the oracles are used to detect *Data Loss* failures after each *DoubleRotatoinEvent*. The *specialEvents* variable is used to correctly handle the *special event sequence* generation. Moreover, the exploration ends when the fixed number of events to inject is reached. Instead, Algorithm 5 shows how the event to be triggered is chosen. First, it checks whether the application is in the foreground and eventually generates specific events to push it on top of the activity stack. It also creates the *AbstractStates* used for the event generation and stores them. If such *AbstractState* has never been encountered during the exploration, the policy starts to inject a *special event sequence*, otherwise its equivalent one is fetched from the *AbstractStates* stored in order to use the information about the events already triggered from it, thus generating a new event.

Algorithm 4 *DataLossPolicy* algorithm

Input: $maxNumOfEvents$: the fixed number of events to be triggered

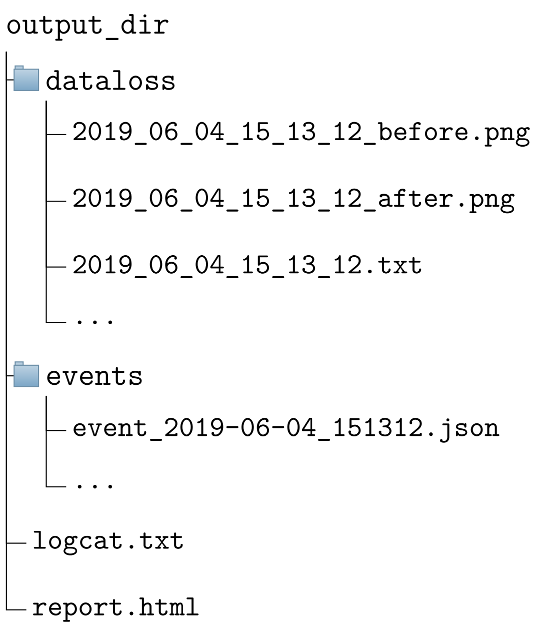
```
$count \leftarrow 0$
while $count < maxNumOfEvents$ do
    if $count == 0$ then
        $event \leftarrow$ KeyEvent(HOME)
    else if $count == 1$ then
        $event \leftarrow$ Intent("start initial activity")
    else
        $event \leftarrow$ nextEvent()
    end if
    if $event ==$ DoubleRotationEvent then
        $screenBefore \leftarrow$ takeScreenshotFromDevice()
        $viewsBefore \leftarrow$ getViewsFromDevice()
    end if
    execute($event$)
    if $event ==$ DoubleRotationEvent then
        $screenAfter \leftarrow$ takeScreenshotFromDevice()
        $viewsAfter \leftarrow$ getViewsFromDevice()
        if $screenBefore \neq screenAfter \vee viewsBefore \neq viewsAfter$ then
            print Data Loss found!
            $specialEvents \leftarrow false$
            saveAsImg($screenBefore$, $screenAfter$)
            saveAsTxt($viewsBefore$, $viewsAfter$)
        end if
    end if
    $count \leftarrow count + 1$
end while
```

Algorithm 5 Algorithm to choose the next event to be triggered

```
if $app \notin activityStack$ then
    return Intent(“start initial activity”)
else if $app \neq foregroundApp$ then
    return KeyEvent(BACK)
end if
$currentActivity \leftarrow$ getCurrentActivity()
$possibleEvents \leftarrow$ getPossibleEvents()
$abstractState \leftarrow$ AbstractState($currentActivity, possibleEvents$)
if $abstractState \notin abstractStatesFound$ then
    $abstractStatesFound \leftarrow abstractStatesFound \cup abstractState$
    $specialEvents \leftarrow true$
    return FillAllEvent
else
    $abstractState \leftarrow$ extract($abstractState, abstractStatesFound$)
end if
if $lastEvent ==$ FillAllEvent then
    return DoubleRotationEvent
else if $lastEvent ==$ DoubleRotationEvent $\wedge$ $specialEvents == true$ then
    $specialEvents \leftarrow false$
    return ScrollEvent(FULL_DOWN)
end if
$num \leftarrow$ Uniform(0, 1)
if $0 \leq num \leq \varepsilon$ then
    $events \leftarrow$ getAllEvents($abstractState$)
else
    $events \leftarrow$ getEventsNotYetTriggered($abstractState$)
end if
$event \leftarrow$ extractRandomEvent($events$)
return $event$
```

4.3.4 Report generation

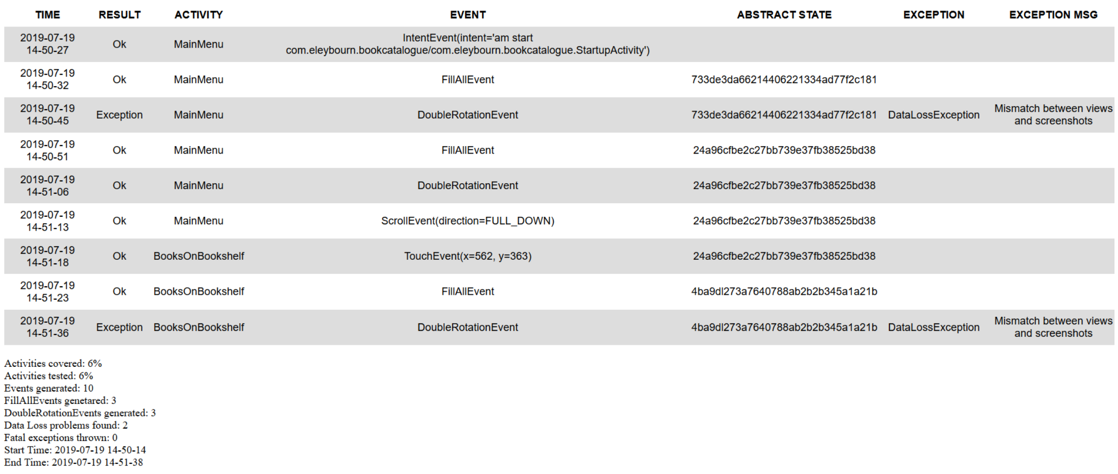
At the end of the execution, a folder containing useful information about the performed analysis is generated. The structure of the output directory is illustrated in Figure 4.10. The “dataloss” folder contains a set of images and text files, which are created whenever one of the two oracles fails. The images correspond to the snapshots of the *screenshot-based oracle* and they are named “year_month_day_hour _minute_second_moment.png”, where “moment” can be either “before” or “after”, depending on whether the screenshot was taken before or after a *stop-start* event. Instead, the text files contain the snapshots used by the *property-based oracle* and they are named “year_month_day_hour_minute _second_views.txt”. This information is useful to analyze the results of the execution. In fact, they can be used to understand why a *Data Loss* failure was detected at a certain time.
The *events* folder contains JSON files, each of which represents an abstraction of



```
output_dir
├─ dataloss
│   ├─ 2019_06_04_15_13_12_before.png
│   ├─ 2019_06_04_15_13_12_after.png
│   ├─ 2019_06_04_15_13_12.txt
│   └─ ...
├─ events
│   ├─ event_2019-06-04_151312.json
│   └─ ...
├─ logcat.txt
└─ report.html
```

Figure 4.10: Output directory

a specific event generated during the exploration. If the user selects the replay exploration policy, which is included in the default version of *DroidBot* [13], these files will be used to generate the same exact event sequence of a previous execution.
The *logcat.txt* file contains all the system messages provided by *Logcat* [20], including the stack traces of the fatal exceptions thrown. It can be useful to understand the nature of the crashes observed and to analyze whether they were caused by *Data Loss* faults.
Finally, the *report.html* file contains the list of all the events generated during the execution and, for all of them, useful information is reported, as shown in Figure 4.11.



TIME	RESULT	ACTIVITY	EVENT	ABSTRACT STATE	EXCEPTION	EXCEPTION MSG
2019-07-19 14-50-27	Ok	MainMenu	IntentEvent(intent='am start com.eleybourn.bookcatalogue/com.eleybourn.bookcatalogue.StartupActivity')			
2019-07-19 14-50-32	Ok	MainMenu	FillAllEvent	733de3da66214406221334ad77f2c181		
2019-07-19 14-50-45	Exception	MainMenu	DoubleRotationEvent	733de3da66214406221334ad77f2c181	DataLossException	Mismatch between views and screenshots
2019-07-19 14-50-51	Ok	MainMenu	FillAllEvent	24a96cfbe2c27bb739e37fb38525bd38		
2019-07-19 14-51-06	Ok	MainMenu	DoubleRotationEvent	24a96cfbe2c27bb739e37fb38525bd38		
2019-07-19 14-51-13	Ok	MainMenu	ScrollEvent(direction=FULL_DOWN)	24a96cfbe2c27bb739e37fb38525bd38		
2019-07-19 14-51-18	Ok	BooksOnBookshelf	TouchEvent(x=562, y=363)	24a96cfbe2c27bb739e37fb38525bd38		
2019-07-19 14-51-23	Ok	BooksOnBookshelf	FillAllEvent	4ba9dl273a7640788ab2b2b345a1a21b		
2019-07-19 14-51-36	Exception	BooksOnBookshelf	DoubleRotationEvent	4ba9dl273a7640788ab2b2b345a1a21b	DataLossException	Mismatch between views and screenshots

Activities covered: 6%
Activities tested: 6%
Events generated: 10
FillAllEvents genetared: 3
DoubleRotationEvents generated: 3
Data Loss problems found: 2
Fatal exceptions thrown: 0
Start Time: 2019-07-19 14-50-14
End Time: 2019-07-19 14-51-38

Figure 4.11: Example of report.html of Book Catalogue v5.2.0-RC3a

CHAPTER 5

Experimental evaluation

Once the implementation phase was completed, in order to evaluate its effectiveness in detecting *Data Loss* failures in Android applications, the technique described in this thesis work has been tested with the apps contained in the benchmark provided by Riganelli et al. [1]. The benchmark includes 110 known *Data Loss* problems affecting 48 buggy Android apps (56 considering all their versions) and *Appium* [8] test cases useful for replaying and understanding such problems, where 98 of them also implement *JUnit* [7] oracles to automatically detect the corresponding failures. In addition, it contains further 58 *Data Loss* problems already reported to the developers, but not provided with the test cases necessary for their replication.
This chapter justifies the choices made to initialize the evaluative experiments, summarizes the data obtained from such experiments and highlights the emerged strengths and weaknesses of the proposed technique.

5.1 Study setup

All the experiments have been conduced on a Genymotion v3.0.2 Android emulator executed on a Linux machine. More precisely, the emulator was a Google Nexus 5 running Android 6.0 API 23 with 4 processors and 2 GB of RAM, while the Linux machine had an Intel Core I5-560M processor, 8 GB of RAM and ran Ubuntu 18.04.
The experimentation work on the apps of the benchmark required the setting of three parameters:

1. The number of runs for every app;
2. The amount of time to be allocated for each run;
3. The value of the ϵ parameter.

The choice for the number of runs and for the amount of time to be allocated for each run has been made relying on the study conduced by Wang et al. [18], who performed an experimentation work to evaluate the main state-of-the-art test case generation tools on industrial Android apps. In their study, they carried out their experiments allocating 3 runs of 3 hours for each application in order to "compensate potential influence brought by randomness during testing".
Conforming to their decisions, since the stopping criteria for the exploration strategy can only be specified in terms of number of events to be triggered and since the emulator took about 4.8 seconds to complete the generation of a single event, every application of the benchmark has been tested with 3 runs of 2250 events.

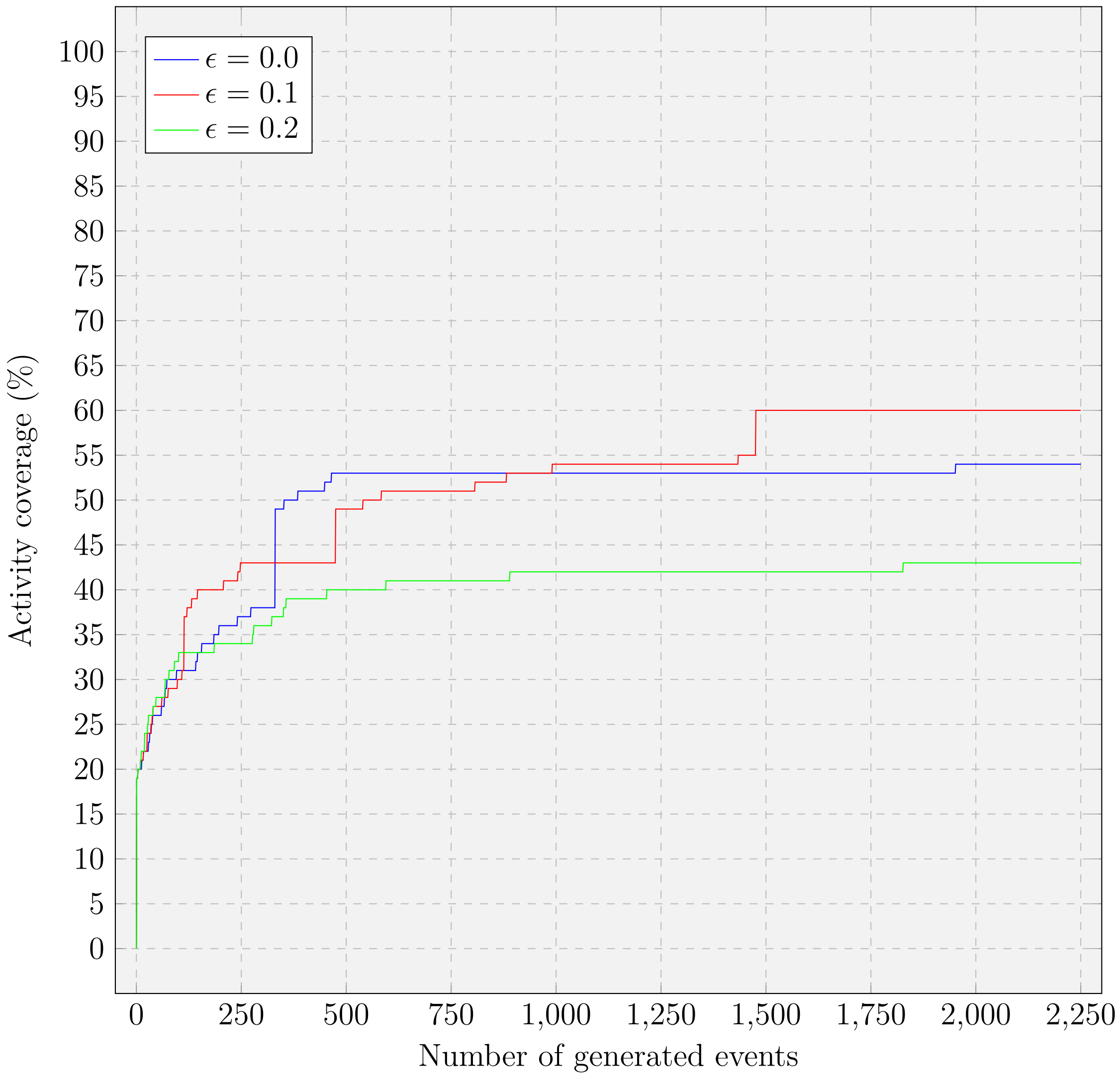


Figure 5.1: Trend of average activity coverage (in %) to varying of the ϵ parameter on Equate v1.6, Calendar Notifications v3.14.159 and Twidere v3.7.3

To decide the value of the ϵ parameter, three apps, able to represent all the applications of the benchmark in terms of number of activities, have been first selected and then tested. First of all, all the apps have been grouped based on their number of activities and just the ones of smallest, medium and biggest size have been considered. Later, a random app has been extracted from each class, obtaining the following sample:

- Equate v1.6 with 2 activities;
- Calendar Notifications v3.14.159 with 13 activities;
- Twidere v3.7.3 with 52 activities.

Afterwards, different experiments have been performed on the aforementioned applications with the purpose of choosing the best value of the ϵ parameter, which would have led to the highest activity coverage, where an experiment was composed by a total of 9 runs: 3 runs of 2250 events for each application with a fixed value of the ϵ parameter.
At the end of each experiment, the average activity coverage has been computed considering the results of all the 9 runs composing it. If the average value of the activity coverage of a specific experiment was less than the value obtained from the previous one, then the value of the corresponding ϵ would have been rejected and the one of the previous experiment accepted.
As shown in Figure 5.1, the first experiment set $\epsilon = 0.0$ and the average activity coverage achieved was 52%, the second one set $\epsilon = 0.1$ and it achieved 60% of average activity coverage, while the last one set $\epsilon = 0.2$, achieving 44% of average activity coverage. Since the last experiment had led to a worse average value of activity coverage than the previous one, the evaluation of the technique was carried out with $\epsilon = 0.1$.
In conclusion, the three parameters required to start the experimentation work on the 48 Android apps of the benchmark were the following:

1. The number of runs for every app: 3;
2. The amount of time to be allocated for each run: 3 hours (2250 events);
3. The value of the ϵ parameter: 0.1.

5.2 Experimentation work

In order to measure the effectiveness of the developed technique in terms of ability to detect the *Data Loss* problems reported in the benchmark as well as new ones, all the applications have been tested using the parameters described in section 5.1.
Before launching their executions, the apps have been granted with the permissions for accessing to the protected resources of the device (e.g., contacts, files or camera) needed for their correct functioning.
The default version of *DroidBot* [13] already provided a mechanism for translating JSON files, which must be written following specific structures, to ordered event sequences. Thanks to this functionality, since many apps required to login, different JSON files containing information about usernames and passwords have been written to inject the specific event sequences in order to complete the authentication. An example of a JSON file is shown in Figure 5.2. The applications that required JSON files are the following:

- Conversations v1.14.0 and v1.23.8
- K-9 Mail v5.010, v5.207 and v5.401
- Octodroid v4.0.3 and v4.2.0
- QuasselDroid v0.11.5
- Tusky for Mastodon v1.0.3
- Twidere v3.7.3

To launch the experiments on the selected apps, *DroidBot* [13] was executed with the command reported in Figure 5.3.

```json
{
    "views": {
        "login_username": {
            "resource_id": ".*et_username_main",
            "class": ".*EditText"
        },
        "login_password": {
            "resource_id": ".*et_password_main",
            "class": ".*EditText"
        },
        "login_button": {
            "text": "Login",
            "class": ".*TextView"
        }
    },
    "states": {
        "login_state": {
            "views": ["login_username", "login_password", "login_button"]
        }
    },
    "operations": {
        "login_operation": [
            {
                "event_type": "set_text",
                "target_view": "login_username",
                "text": "unimibtest"
            },
            {
                "event_type": "set_text",
                "target_view": "login_password",
                "text": "unimib123456"
            },
            {
                "event_type": "touch",
                "target_view": "login_button"
            }
        ]
    },
    "main": {
        "login_state": ["login_operation"]
    }
}
```

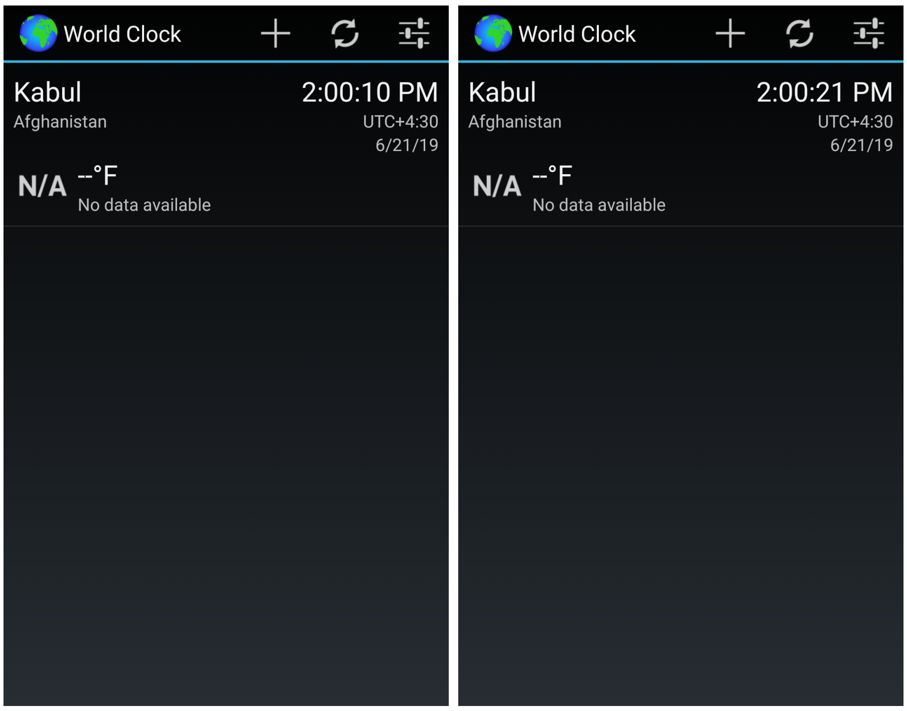
Figure 5.2: The JSON file used to test OctoDroid v4.0.3

```
droidbot -a appname.apk -o outdir -policy data_loss -epsilon 0.1 -count 2250 -grant_perm [-script file.json]
```

Figure 5.3: The shell command used to start the execution of *DroidBot* [13] on the appname.apk app

Definition 5.2.1. A *false positive* occurs in one of the following situations:

- The screenshot obtained after the *DoubleRotationEvent* is black, which indicates that the emulator was not responding;
- The screenshot after the *DoubleRotationEvent* was taken while the device screen was still rotating, which indicates that the emulator had not finished to process the adb command yet;
- Either the screenshots or the views before and after the *DoubleRotationEvent* are different, but such differences could not be considered as *Data Loss* failures. Figure 5.4 shows an explanatory example of such cases.



(a) Before a *DoubleRotationEvent* (b) After a *DoubleRotationEvent*

Figure 5.4: A *false positive* detected in World Clock & Weather v1.8.6

Definition 5.2.2. A *true positive* is a *Data Loss* failure detected by the technique that is a real manifestation of a *Data Loss* problem.

Once the execution phase was completed, that is, all the apps were successfully tested, the obtained raw data have been processed and classified with the following criteria:

1. Every *Data Loss* failure detected by the technique was labelled as *true positive* or as *false positive*;
2. Every *Data Loss* failure labelled as *true positive* was further labelled as:
    - *Benchmark* if it belonged to the benchmark;

    - *Online* if it did not belong to the benchmark but it had already been reported to the developers;
    - *New* otherwise.

3. Every *Data Loss* failure was grouped into the corresponding *activity* from which it was detected, thus avoiding considering the same *Data Loss* failures more than once;

4. Every *activity* was labelled as:

    - *False positive* if all its *Data Loss* failures had been labelled as *false positives*;
    - *Known* if at least one of its *Data Loss* failures had been labelled as *benchmark* or *online*;
    - *New* otherwise.

In conclusion, the data processed with the aforementioned criteria have been further analyzed in order to figure out the strengths and weaknesses of the proposed technique resulting from the experiments conduced.

5.3 Results

The results of the evaluation study are shown in Table 5.1. Every row represents the merged results of the 3 runs of an application with its corresponding version. The columns of the table are organized as follows:

- *App name*, which reports the app name;
- *Release*, which reports the version of the app;
- *Activities*, which reports the total number of app activities;
- *Total Buggy Activities*, which reports the total number of buggy app activities combining the activities labelled as *known* with the ones labelled as *new*;
- *Activity Coverage Average (Total)*, which reports both the average activity coverage and the total activity coverage achieved, where the total activity coverage is the ratio between the number of activities found in all the runs and the total number of the app activities;
- *Data Loss Benchmark Found Average (Total)*, which reports both the average number and the total number of the *Data Loss* failures labelled as *benchmark*;
- *Data Loss Online Found Average (Total)*, which reports both the average number and the total number of the *Data Loss* failures labelled as *online*;

- *Activities False Positives Found Average (Total)*, which reports both the average number and total number of the activities found and labelled as *false positives*;
- *New Buggy Activities Found Average (Total)*, which reports both the average number and the total number of the activities found and labelled as *new*;
- *Activities Crashed After A Stop-Start Event Average (Total)*, which reports both the average number and the total number of activities crashed after a *DoubleRotationEvent*.

App name	**Release**	**Activities**	**Total Buggy Activities**	**Activity Coverage Average (Total)**	**Data Loss Benchmark Found Average (Total)**	**Data Loss Online Found Average (Total)**	**Activities False Positives Found Average (Total)**	**New Buggy Activities Found Average (Total)**	**Activities Crashed After A Stop-Start Event Average (Total)**
Amaze File Manager	v3.1.0-beta.1	4	4	100% (100%)	3/5 (3/5)	2/2 (2/2)	0 (0)	2 (2)	1 (1)
AntennaPod	v1.5.2.0	16	5	33% (44%)	5/7 (5/7)	2/11 (2/11)	1 (1)	3 (3)	0 (0)
BeeCount	v2.4.7	8	7	96% (100%)	1/3 (1/3)	1/5 (1/5)	1 (1)	4 (4)	0 (0)
BookCatalogue	v5.2.0-RC3a	35	20	66% (71%)	5/7 (6/7)	-	0 (0)	11 (14)	1 (1)
Calendar Notification	v3.14.159	13	7	67% (69%)	3/3 (3/3)	1/1 (1/1)	0 (0)	4 (4)	1 (1)
Conversations	v1.14.0	21	7	41% (57%)	0/1 (0/1)	-	0 (0)	4 (6)	1 (1)
Conversations	v1.23.8	23	9	40% (57%)	1/1 (1/1)	-	0 (0)	6 (8)	0 (0)
CycleStreets	v3.5	11	6	55% (55%)	1/1 (1/1)	-	0 (0)	5 (5)	0 (0)
Diary	v1.26	3	3	100% (100%)	1/2 (2/2)	-	0 (0)	2 (2)	0 (0)
DNS66	v0.3.3	5	3	100% (100%)	1/1 (1/1)	-	0 (0)	2 (2)	0 (0)
Document Viewer	v2.7.9	9	3	48% (56%)	0/1 (0/1)	-	1 (1)	2 (2)	0 (0)
Easy xkcd	v6.0.4	9	6	74% (78%)	1/1 (1/1)	-	0 (0)	5 (5)	2 (2)
Equate	v1.6	2	2	100% (100%)	2/2 (2/2)	1/1 (1/1)	0 (0)	1 (1)	1 (1)
Etar Calendar	v1.0.10	12	2	42% (42%)	4/5 (4/5)	2/5 (3/5)	0 (0)	0 (0)	0 (0)
Firefox Focus	v4.0	6	4	44% (50%)	0/1 (0/1)	-	0 (0)	3 (3)	0 (0)
Flym	v1.3.4	6	5	83% (83%)	0/1 (0/1)	-	0 (0)	4 (4)	0 (0)
Gadgetbridge	v0.25.1	20	3	28% (30%)	1/1 (1/1)	-	2 (3)	2 (2)	1 (1)
K-9 Mail	v5.010	28	10	41% (46%)	0/1 (0/1)	-	0 (0)	7 (9)	1 (1)
K-9 Mail	v5.207	27	9	51% (63%)	1/1 (1/1)	-	0 (0)	7 (8)	1 (1)
K-9 Mail	v5.401	29	9	54% (59%)	1/1 (1/1)	-	0 (0)	7 (8)	1 (1)
KISS Launcher	v2.25.0	2	2	100% (100%)	1/1 (1/1)	-	0 (0)	1 (1)	0 (0)
Loop Habit Tracker	v1.6.2	7	5	71% (71%)	2/6 (2/6)	-	1 (1)	0 (0)	0 (0)
MALP	3d31062	2	1	100% (100%)	1/1 (1/1)	-	0 (0)	0 (0)	1 (1)
MALP	v1.1.0	4	1	33% (50%)	2/4 (3/4)	-	0 (0)	0 (0)	1 (1)
Mgit	v1.5.0	10	9	97% (100%)	1/1 (1/1)	-	1 (1)	7 (8)	1 (1)
MTG Familiar	v3.5.5	2	1	50% (50%)	0/1 (1/1)	-	0 (0)	0 (0)	0 (0)
Notepad	v2.3	3	2	67% (67%)	1/1 (1/1)	-	0 (0)	1 (1)	0 (0)
OctoDroid	v4.0.3	44	21	56% (66%)	2/2 (2/2)	-	0 (0)	16 (19)	1 (1)
OctoDroid	v4.2.0	46	22	44% (57%)	0/1 (1/1)	-	1 (2)	17 (21)	0 (0)
Omni Notes	v5.4.3	17	4	29% (35%)	0/1 (0/1)	-	0 (0)	3 (3)	0 (0)
OpenTasks	v1.1.13	9	6	78% (78%)	1/1 (1/1)	-	0 (0)	5 (5)	0 (0)
OpenVPN for Android	v0.7.5	13	6	46% (46%)	1/1 (1/1)	-	0 (0)	5 (5)	1 (1)
PassAndroid	v3.3.3	14	3	36% (36%)	2/3 (3/3)	8/8 (8/8)	0 (0)	0 (0)	0 (1)
Periodic Table	v1.1.1	3	3	100% (100%)	2/2 (2/2)	-	0 (0)	1 (1)	0 (0)
Port Knocker	v1.0.8	6	2	50% (50%)	2/3 (2/3)	0/1 (0/1)	0 (0)	0 (0)	0 (0)
Prayer Times	v3.6.6	22	9	32% (36%)	3/7 (6/7)	1/2 (1/2)	0 (0)	4 (6)	1 (1)
QuasselDroid	v0.11.5	5	3	40% (40%)	1/1 (1/1)	-	0 (0)	2 (2)	1 (1)
QuickLyric	v2.1	4	3	75% (75%)	1/1 (1/1)	-	0 (0)	2 (2)	0 (0)
Simple Draw	v3.1.5	7	3	86% (86%)	0/1 (0/1)	-	0 (0)	2 (2)	0 (0)
Simple File Manager	v2.6.0	8	6	88% (88%)	1/1 (1/1)	-	0 (0)	3 (5)	1 (1)
Simple File Manager	v3.2.0	8	6	75% (75%)	1/1 (1/1)	-	0 (0)	4 (5)	0 (0)
Simple Gallery	v1.50	11	5	48% (55%)	1/4 (2/4)	1/7 (1/7)	0 (0)	3 (3)	0 (0)
Simple Solitaire	v2.0.1	7	3	93% (100%)	0/1 (1/1)	-	0 (0)	1 (2)	1 (1)
Simpletask	v10.0.7	11	7	67% (73%)	1/1 (1/1)	-	0 (0)	6 (6)	0 (0)
SMS Backup Plus	v1.5.11-Beta18	7	2	14% (14%)	1/1 (1/1)	-	0 (0)	1 (1)	0 (0)
Syncthing	v0.9.5	9	9	93% (100%)	3/5 (4/5)	2/2 (2/2)	1 (1)	5 (6)	1 (1)
Taskbar	v3.0.3	21	2	26% (29%)	2/2 (2/2)	12/13 (13/13)	1 (1)	1 (1)	1 (1)
Tasks Astrid To-Do List Clone	v6.0.6	45	9	19% (27%)	0/1 (0/1)	-	1 (1)	6 (8)	1 (1)
Tusky for Mastodon	v1.0.3	12	8	78% (83%)	1/1 (1/1)	-	0 (0)	6 (7)	3 (3)
Twidere	v3.7.3	52	7	14% (17%)	0/1 (0/1)	-	0 (0)	6 (6)	0 (0)
Vespucci Osm Editor	v10.2	19	7	42% (47%)	1/1 (1/1)	-	1 (1)	5 (6)	0 (0)
Vlille Checker	v4.4.0	6	3	67% (67%)	1/1 (1/1)	-	0 (0)	2 (2)	0 (0)
WiFiAnalyzer	1.9.2	4	3	75% (75%)	3/3 (3/3)	-	0 (0)	1 (1)	0 (0)
World Clock & Weather	v1.8.6	4	4	100% (100%)	0/1 (0/1)	-	0 (0)	3 (3)	0 (0)
TOTAL	**-**	**713**	**311**	**62% (66%)**	**- (82/110)**	**- (36/58)**	**1 (14)**	**5 (230)**	**1 (27)**

Table 5.1: Results of the experiments

5.3.1 Activity coverage

The results reported in Table 5.1 and in Figure 5.5 show that the developed technique achieved an average value of 66% for the total activity coverage and 62%

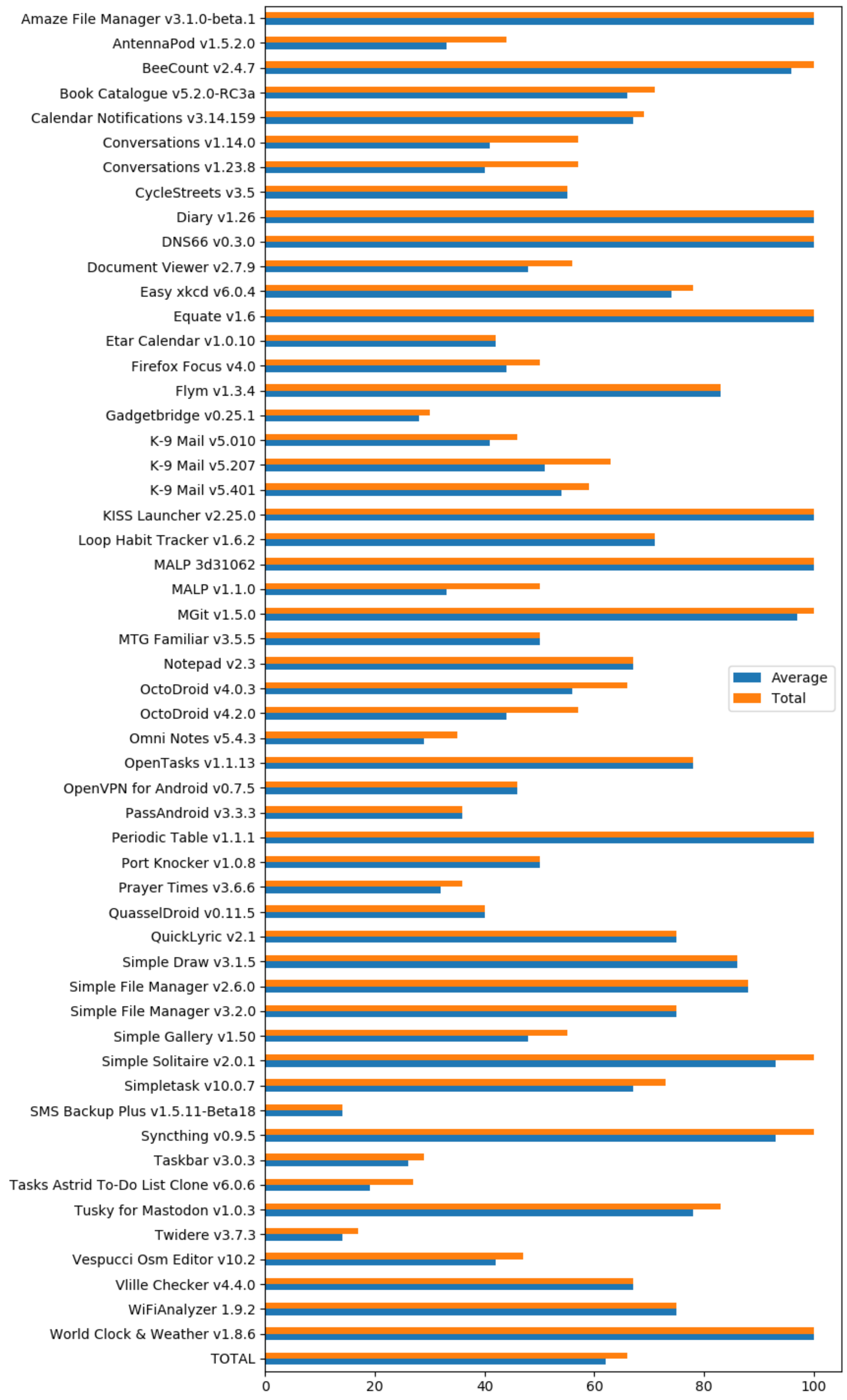


Figure 5.5: Activity coverage reached for each app

for the average activity coverage. Unfortunately, although QuasselDroid v0.11.5 and Conversations v1.14.0 required to login and the corresponding JSON files had been written, some problems during the authentication phase arose and thus they were not completely tested, negatively affecting the activity coverage.

5.3.2 Data Loss problems

As shown in Table 5.1, the technique found 82 out of 110 *Data Loss* problems of the benchmark and 36 out of 58 *Data Loss* problems not belonging to the benchmark but already reported to the developers.
Figure 5.6 shows that all the already known *Data Loss* problems affecting 32 apps out of 56 were detected during the experimentation. Instead, for 9 apps out of 56 no *Data Loss* failures already known were found.
After a detailed analysis of the remaining 50 *Data Loss* problems to find, the latter have been divided into 3 classes based on the reasons why these failures had not been discovered during the exploration of the apps:

- *Complex sequence*: the *Data Loss* failure required a specific event sequence to be found that was not generated, even though the current implementation of the technique allowed it;
- *Preliminary actions required*: the *Data Loss* failure could have been found with the current implementation of the technique, but preliminary operations were required;
- *Improvements required*: the *Data Loss* failure was not found because the current implementation of the technique did not allow to generate the required event sequence.

Figure 5.7 shows the distribution of the remaining 50 *Data Loss* failures not found into the 3 aforementioned classes: 41 of them were not found because they required complex event sequences, 4 of them required preliminary actions and 5 of them required improvements in the technique. For example, the *Data Loss* failure reported in the issue 439 of the benchmark in OmniNotes v.5.4.3 was classified as a *complex sequence*, as it would have been necessary to create a protected note with a password, unlock it and, then, inject a *DoubleRotationEvent.* Although this event sequence could have been generated with the current implementation of the technique, the probability that this would have happened was very low. The *Data Loss* failure reported in the issue 44 of the benchmark in Document Viewer v2.7.9 required to open a book or a document, change to another page and rotate the device screen, but, since it was necessary to have a book or a document already downloaded and imported in the app, it was classified as *preliminary actions required.* Instead, the *Data Loss* problem reported in the issue 654 of the benchmark in Tasks Astrid To-Do List Clone v6.0.6 required to temporarily exit from the app and take a picture with the device camera. Since the current implementation of the technique was not supposed to generate events outside the application boundaries, it was impossible to detect this failure without improving the exploration

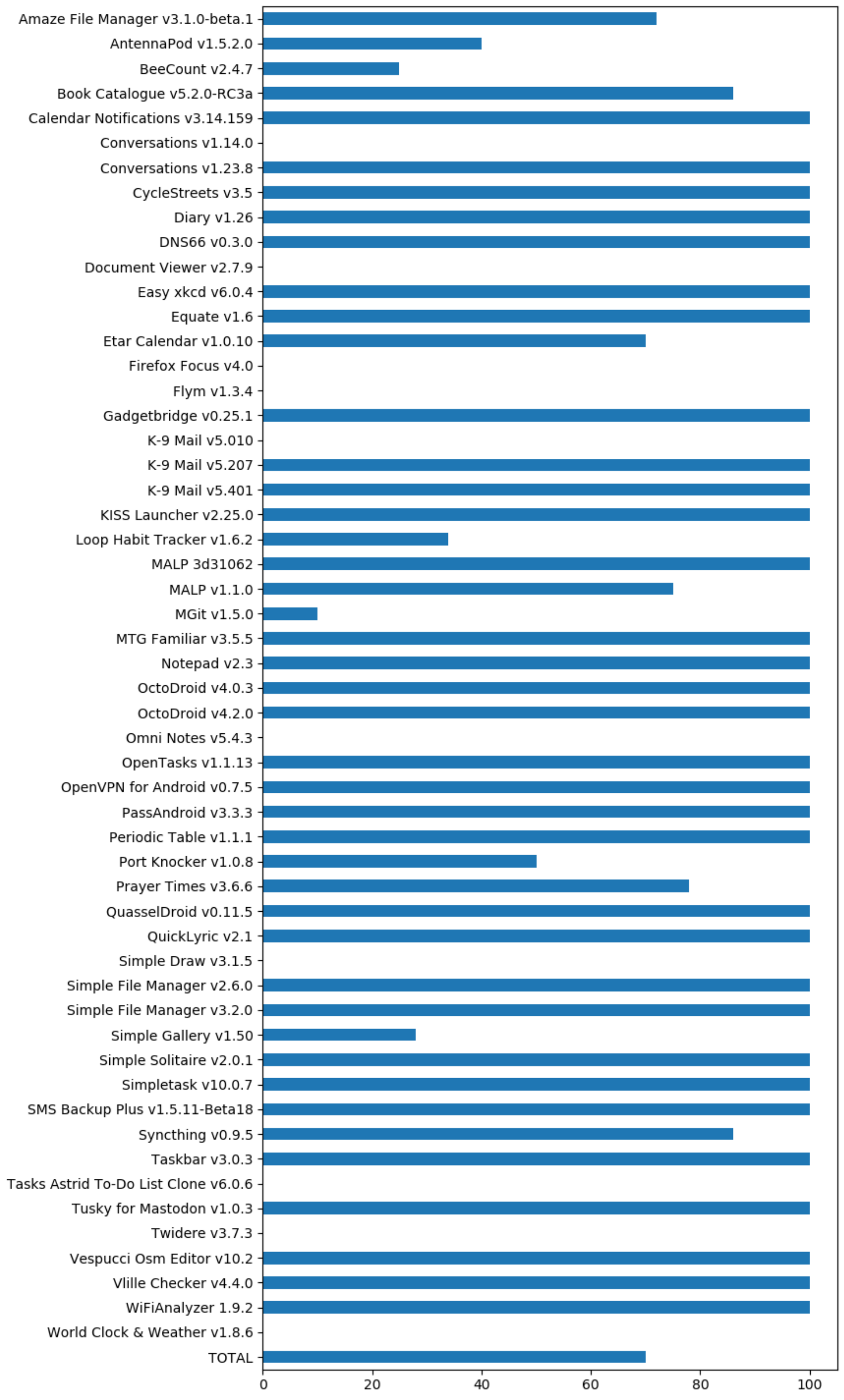


Figure 5.6: Percentage of *Data Loss* failures labelled as *benchmark* or *online* detected for each app

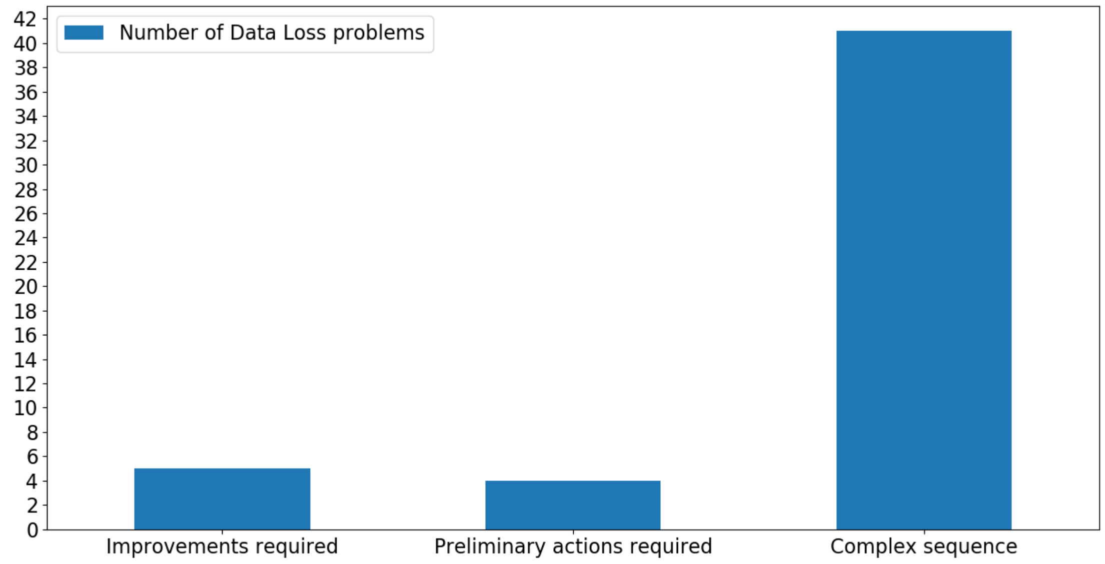


Figure 5.7: Classification of the undiscovered *Data Loss* problems labelled as *benchmark* or *online*

strategy, allowing it to take also steps outside the app under test. For this reason, this *Data Loss* problem was classified as *improvement required*.
In addition, the results demonstrate that the developed technique managed to detect many new buggy activities. Initially, as shown in Table 5.1 and in Figure 5.8, the total number of known buggy activities were only 81 out of 713, which corresponds to 12% of the total number of activities. At the end of the experimentation phase, this number increased up to 230 out of 713, which corresponds to 47% of the total number of activities. For example, before the experimentation, in OctoDroid v4.2.0 just 1 activity was known to be buggy while at the end of the analysis this number increase up to 22 buggy activities.
In conclusion, even though the benchmark does not include any *Data Loss* failures leading to application crashes, the technique discovered many of them. In particular, 23 apps out of 56 registered at least a crash after a *DoubleRotationEvent*. For example, in Tusky For Mastodon v1.0.3, 3 activities out of 12 manifested *Data Loss* problems causing application crashes.

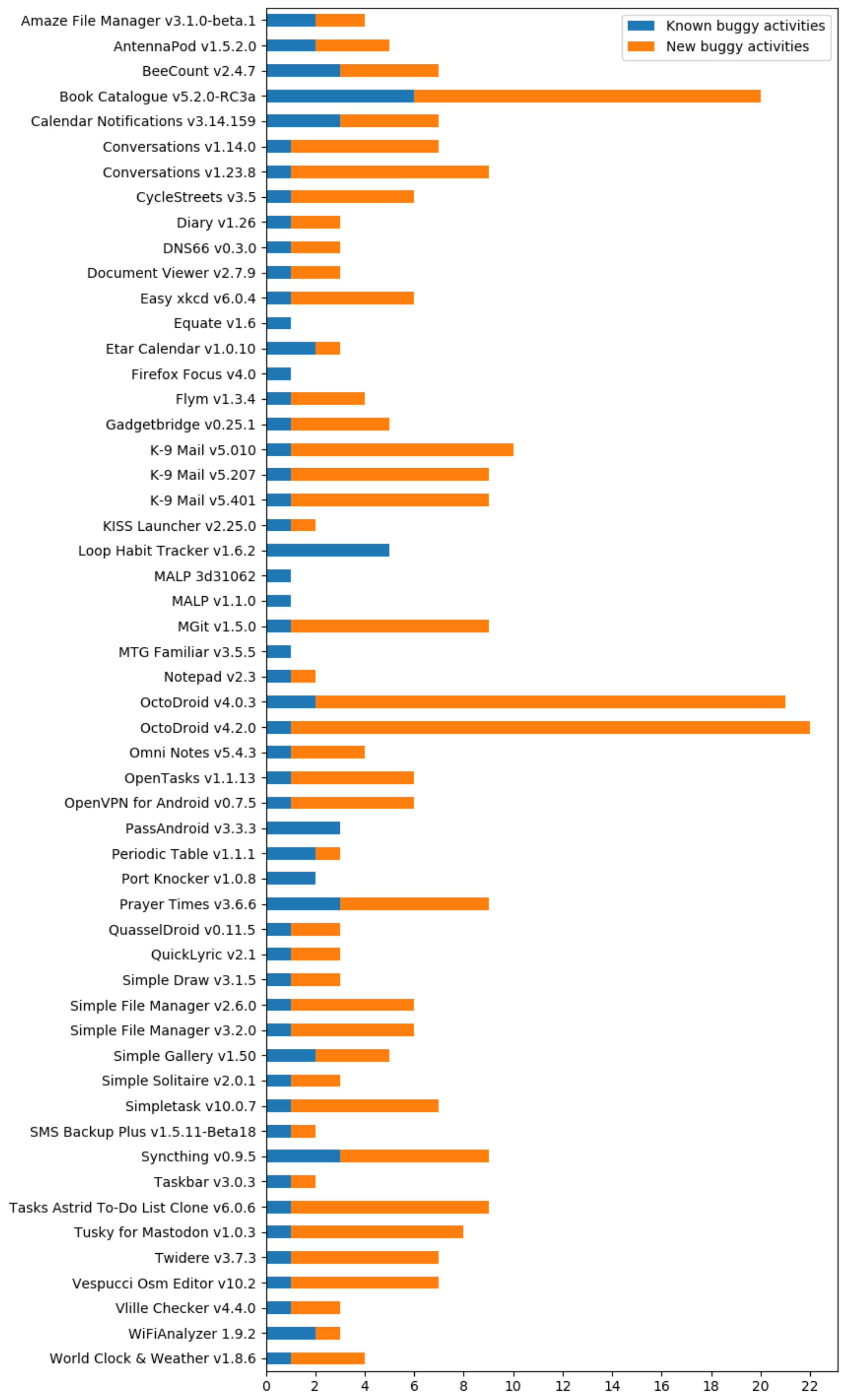


Figure 5.8: Number of buggy activities for each app

5.3.3 Oracles

Since 118 out of 168 (72 %) *Data Loss* failures already known were found, this demonstrates that both the exploration strategy and the oracles had correctly been implemented.
The implemented oracles are effective in detecting *Data Loss* failures: 7 out of 12 *Data Loss* failures of the benchmark, for which it was not possible to implement *JUnit* [7] assertions in their associated *Appium* [8] test case, have been detected. This indicates that, in these cases, the developed oracles are better than the oracles created using *JUnit* [7] in detecting *Data Loss* problems.
Figure 5.9 shows the total number of *Data Loss* failures labelled as *true positives* detected for each application and divided into the corresponding oracles from which they were detected. It also confirms that the two types of implemented oracles are complementary, as previously mentioned in subsection 4.3.2. In fact, as also shown in Figure 5.10, there are *Data Loss* failures detected only by the *screenshot-based oracle* (1246 out of 13605) and others only by the *property-based oracle* (2417 out of 13605).
A similar analysis was also conduced for the *Data Loss* failures labelled as *false positives* and the results are shown in Figure 5.11 and in Figure 5.12. Even in this case, the majority of the readings shows that the *Data Loss* failures were detected by both the two oracles (1180 out of 1645). The remaining 372 and 93 *false positives* were respectively detected by the *screenshot-based oracles* and the *property-based oracle*. What is emerged from a detailed analysis on the *false positives* detected only by the *property-based oracle* is that also the screenshots before and after the *DoubleRotationEvent* were different but were treated as equal because the threshold value set for the comparison between the two images was still high, that is, 0.0002. This indicates that if the threshold value was set to 0.0, there would have not been *false positives* detected only by the *property-based oracle*.

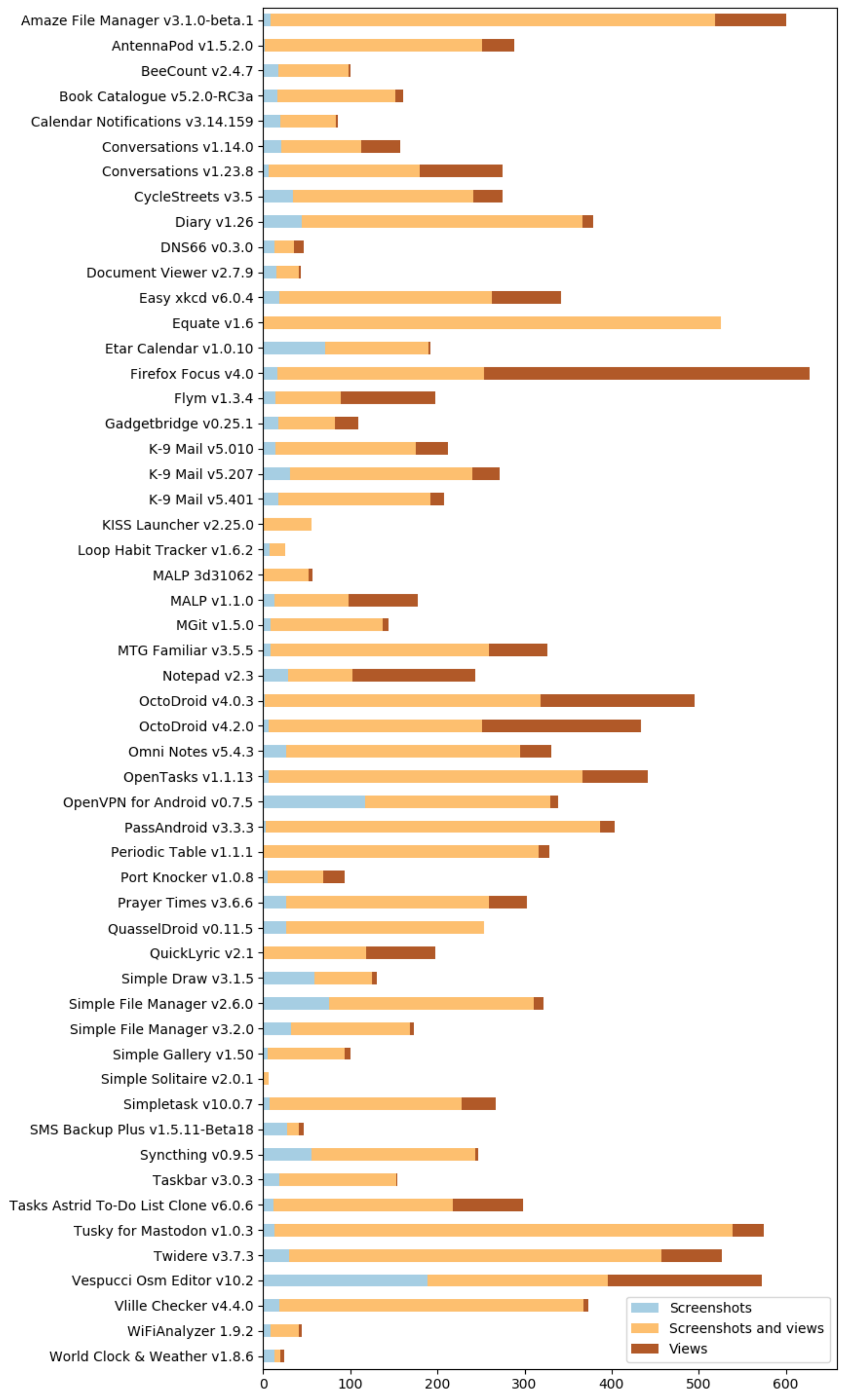


Figure 5.9: Number of *Data Loss* failures labelled as *true positives* detected for each app

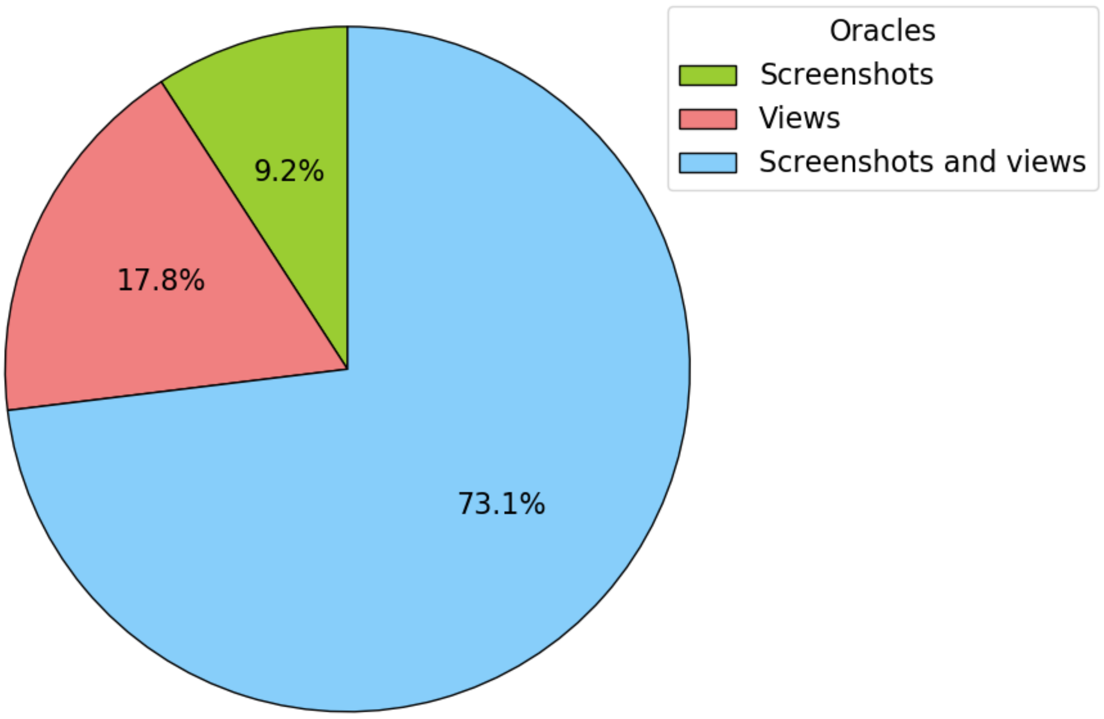


Figure 5.10: Percentage of *Data Loss* failures labelled as *true positives* detected by every oracle

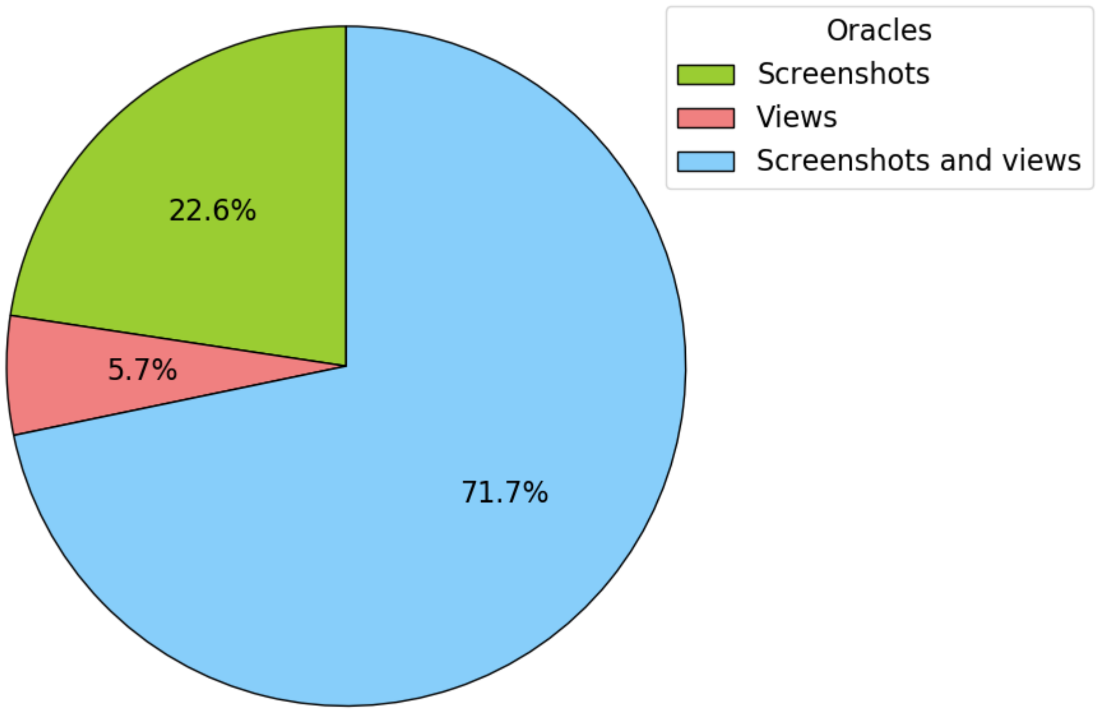


Figure 5.11: Percentage of Data Loss failures labelled as *false positives* detected by every oracle

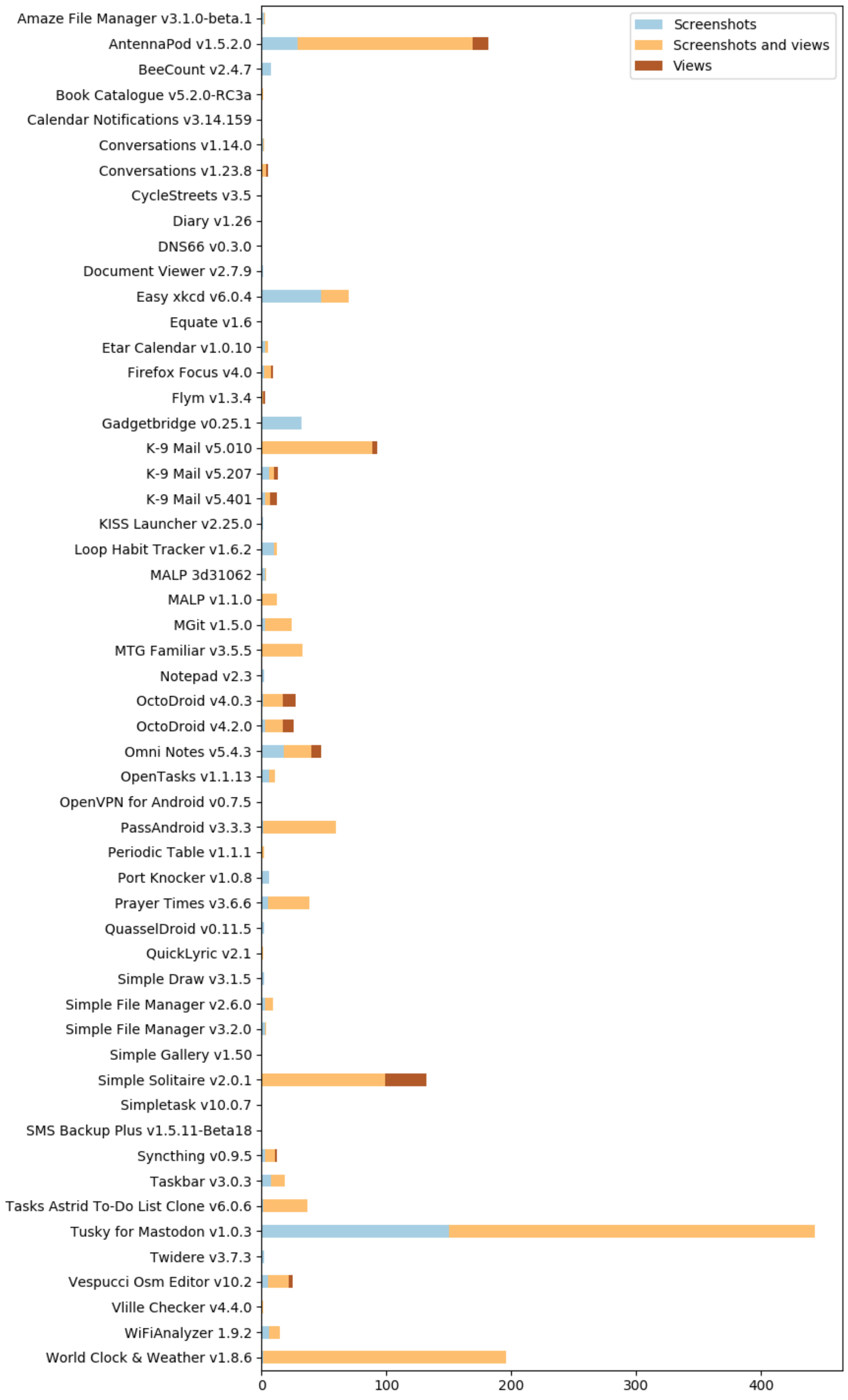


Figure 5.12: Number of the *Data Loss* failures labelled as *false positives* detected for each app

CHAPTER 6

Conclusions

This thesis work has presented the concept of *Data Loss* problem in Android applications, describing how they can alter the expected behaviours of the affected apps and underlining how pervasive they are. Then, after presenting the state of the art of Android testing, it has highlighted the necessity of a specific tool capable of automatically detecting these types of problems and, hence, it has introduced a novel technique able to achieve this goal as well as an evaluation study to measure its effectiveness.
More specifically, a *Data Loss* failure occurs when the variables of the application lose their values after a *stop-start* event and it can manifest itself both in GUI failures and in application crashes. These problems can be introduced in an Android app when the developers do not implement properly the logic for saving and restoring the activity states in correspondence of *stop-start* events.
Nowadays, although these types of problems affect many Android apps, there are no tools able to automatically detect them. In fact, the main state-of-the-art test case generation tools are only able to observe application crashes because they do not implement specific oracles in order to detect different failures.
To address this problem, this thesis has introduced a novel technique capable of detecting *Data Loss* failures while automatically exploring the apps. This technique has been integrated in *DroidBot* [13], a state-of-the-art test case generation tool, and it uses a *model-based* exploration strategy in order to explore the apps under test, generating events based on a GUI model constructed at runtime. It heuristically injects *stop-start* events and it uses two different oracles to detect *Data Loss* problems by comparing the activity GUI states before and after such events.
The evaluative study presented in this thesis shows the effectiveness of the proposed technique in detecting *Data Loss* failures. This study involved 48 buggy Android apps contained in the benchmark provided by Riganelli et al. [1] and the results demonstrated that the aforementioned technique managed to detect 82 out of 110 *Data Loss* problems of the benchmark, 36 out of 58 *Data Loss* problems

already reported to the developers and many new ones, some of which also caused application crashes.
The results obtained during the evaluation study confirm the effectiveness of the implemented oracles in detecting *Data Loss* failures, but the exploration strategy still needs different improvements.
In future work, since some *Data Loss* failures can not be detected with the current implementation of the technique, the exploration strategy must be further improved allowing to detect them.
Finally, thanks to this thesis work, from now on it will be possible to detect *Data Loss* failures affecting Android applications using this automatic test case generation technique.

Bibliography

[1] Oliviero Riganelli, Marco Mobilio, Daniela Micucci, and Leonardo Mariani. “A Benchmark of Data Loss Bugs for Android Apps”. In: *Proceedings of the International Conference on Mining Software Repositories (MSR)*. 2019.

[2] Statista. *Number of available applications in the Google Play Store from December 2009 to December 2018*. 2019. URL: `https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/`.

[3] Android Developers. *Bundle*. URL: `https://developer.android.com/reference/android/os/Bundle.html`.

[4] Oliviero Riganelli, Daniela Micucci, and Leonardo Mariani. “Healing Data Loss Problems in Android Apps”. In: *Proceedings of the IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 2016.

[5] Domenico Amalfitano, Vincenzo Riccio, Ana C. R. Paiva, and Anna Rita Fasolino. “Why does the orientation change mess up my Android application? From GUI failures to code faults”. In: *Software Testing, Verification and Reliability* 28.1 (2018), e1654.

[6] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. “Systematic Execution of Android Test Suites in Adverse Conditions”. In: *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 2015.

[7] *JUnit*. URL: `https://junit.org/junit5/`.

[8] *Appium*. URL: `http://appium.io/`.

[9] *Robotium*. URL: `https://www.robotium.org/`.

[10] *Espresso*. URL: `https://developer.android.com/training/testing/espresso`.

[11] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. “Automated Test Input Generation for Android: Are We There Yet? (E)”. In: *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2015.

[12] Android Developers. *UI/Application Exerciser Monkey.* URL: `https://developer.android.com/studio/test/monkey`.

[13] Yuanchun Li, Yang Ziyue, Guo Yao, and Chen Xiangqun. "DroidBot: A Lightweight UI-guided Test Input Generator for Android". In: *Proceedings of the 39th International Conference on Software Engineering Companion.* 2017.

[14] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. "Guided, Stochastic Model-based GUI Testing of Android Apps". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering.* 2017.

[15] Tanzirul Azim and Iulian Neamtiu. "Targeted and Depth-first Exploration for Systematic Testing of Android Apps". In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications.* 2013.

[16] Ke Mao, Mark Harman, and Yue Jia. "Sapienz: Multi-objective Automated Testing for Android Applications". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis.* 2016.

[17] Razieh Nokhbeh Zaeem, Mukul R. Prasad, and Sarfraz Khurshid. "Automated Generation of Oracles for Testing User-Interaction Features of Mobile Apps". In: *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation.* 2014.

[18] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. "An empirical study of Android test generation tools in industrial cases". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering.* 2018.

[19] Android Developers. *Android Debug Bridge (adb).* URL: `https://developer.android.com/studio/command-line/adb`.

[20] Android Developers. *Logcat command-line tool.* URL: `https://developer.android.com/studio/command-line/logcat`.