UNIVERSITÀ DEGLI STUDI DI NAPOLI
FEDERICO II

UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

PH.D. THESIS

IN

INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

ENHANCING AUTOMATED GUI EXPLORATION TECHNIQUES FOR ANDROID MOBILE APPLICATIONS

VINCENZO RICCIO

TUTOR: PROF. ANNA RITA FASOLINO

COORDINATOR: PROF. DANIELE RICCIO

XXXI CICLO

SCUOLA POLITECNICA E DELLE SCIENZE DI BASE
DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE

Alla mia famiglia.

Abstract

Mobile software applications ("apps") are used by billions of smartphone owners worldwide. The demand for quality to these apps has grown together with their spread. Therefore, effective techniques and tools are being requested to support developers in mobile app quality engineering activities.

Automation tools can facilitate these activities since they can save humans from routine, time consuming and error prone manual tasks. Automated GUI exploration techniques are widely adopted by researchers and practitioners in the context of mobile apps for supporting critical engineering tasks such as reverse engineering, testing, and network traffic signature generation. These techniques iteratively exercise a running app by exploiting the information that the app exposes at runtime through its GUI to derive the set of input events to be fired.

Although several automated GUI exploration techniques have been proposed in the literature, they suffer from some limitations that may hinder them from a thorough app exploration.

This dissertation proposes two novel solutions that contribute to the literature in Software Engineering towards improving existing automated GUI exploration techniques for mobile software applications.

The former is a fully automated GUI exploration technique that aims to detect issues tied to the app instances lifecycle, a mobile-specific feature that allows users to smoothly navigate through an app and switch between apps. In particular, this technique addresses the issues of crashes and GUI failures, that consists in the manifestation of unexpected GUI states. This work includes two exploratory studies that prove that GUI failures are a widespread problem in the context of mobile apps.

The latter solution is a hybrid exploration technique that combines automated GUI exploration with capture and replay through machine learning. It exploits app-specific knowledge that only human users can provide in order to explore relevant parts of the application that can be reached only by firing complex sequences of input events on specific GUIs and by choosing specific input values.

Both the techniques have been implemented in tools that target the Android Operating System, that is today the world's most popular mobile operating system. The effectiveness of the proposed techniques is demonstrated through experimental evaluations performed on real mobile apps.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Over the last decade, the number of smartphone users has considerably increased. This number is steadily growing and is forecast to surpass 2.5 billion in 2019[1]. This is causing a constant demand for new software applications running on mobile devices, commonly referred to as *mobile apps*. As of the month of March 2018, both Android and iOS users had the opportunity to choose from more than 2 million apps[2].

Mobile technology has radically changed the lifestyle of billions of people around the world. More and more people are using mobile apps for several hours every day, entrusting them their sensitive data and carrying out a large variety of activities through them, including critical tasks. Thus, the demand for quality to mobile apps has grown together with their spread. End users require mobile apps to be reliable, robust, efficient, secure and usable. Failures exposed by an app may have a negative impact on the user experience and lead users to look for another app that offers the same features. As a consequence, software developers should give proper consideration to the quality of their mobile apps by adopting suitable quality assurance solutions, such as testing.

Several techniques and tools are currently available for testing a mobile app before it is placed on the market [1]. Test automation tools can facilitate mobile app testing activities since they save humans from routine, time consuming and error prone manual tasks [2].

The research community has devoted great interest to the mobile app testing field in the last years. Several testing approaches have been proposed to assess different quality aspects of mobile applications [3], such as functionality [4, 5, 6, 7, 8, 9], performance [10, 11], security [12, 13, 14, 15], responsiveness [16], accessibility [17], and energy consumption [18, 19, 20].

For testing a mobile app, it is possible to extend and adapt existing testing techniques designed for Event-Driven Systems (EDSs). In event-based testing of EDSs, the behaviour of the system under test is checked with input consisting of specific

[1] https://www.statista.com/statistics/330695/
[2] https://www.statista.com/statistics/276623/

sequences of events, *i.e.* significant state changes [21], that sample its input space [22, 23].

Mobile apps can be considered as EDSs, since they are able to analyze and react to different types of events. Those event types include events triggered on the Graphical User Interface (GUI), events related to the external environment and sensed by device sensors (such as temperature, GPS, accelerometer, gyroscope), events generated by the device hardware platform (such as battery and other external peripheral ports, like USB and headphones), and events generated by other apps running on the mobile device [24].

Existing techniques for automated analysis of mobile apps behavior implement exploration strategies that usually exploit an iterative approach that is based on sending input events to a running app through its User Interface (UI) until a termination criterion is satisfied [25]. These techniques use the information that an app exposes at runtime through its GUI to derive the set of input events to be fired; events may be chosen either randomly [26, 27, 28, 29] or according to a more systematic GUI exploration strategy [30, 31, 32, 33, 34].

In the following, these techniques are referred to as Automated GUI Exploration Techniques (AGETs) [35]. Automated approaches for exploring the behavior of event-driven software applications showed to be extremely useful in several other contexts besides testing, such as reverse engineering [36] and network traffic generation and analysis [37, 38]. Even some major cloud services providers like Amazon[3] and Google[4] are currently offering testing services that exploit AGETs to mobile app developers.

Although AGETs provide a viable approach for automatically exercising mobile apps, they suffer from some limitations that may hinder them from a thorough app exploration.

A limitation intrinsic in the automated exploration approaches consists of not being able to replicate human-like interaction behaviors. In fact, some app features need to be exercised by exploiting app-specific knowledge that only human users can provide. As a consequence, these techniques often fail in exploring relevant parts of the application that can be reached only by firing complex sequences of input events on specific GUIs and by choosing specific input values [39, 40].

Another challenge for AGETs applied to the mobile domain is to target mobile-specific features. Mobile apps have several peculiarities compared to traditional software applications that have to be taken into account by testing techniques and tools, e.g. new interaction gestures, limited resources, new development concepts (e.g. context awareness), the diversity of devices and their characteristics [41].

In particular, the small size of mobile devices introduced the need to have on the screen one single focused app at a time. Mobile OSs define a peculiar lifecycle for app instances in order to manage them transparently to the users who can navigate

[3] https://aws.amazon.com/it/device-farm/
[4] https://console.firebase.google.com/

through an app and switch between apps without losing their progress and data; at the same time, it allows not to waste the limited resources of mobile devices, such as memory and battery.

As pointed out by recent work, this feature is often overlooked by mobile app developers and testers, causing several app issues, *e.g.* crashes if the user receives a phone call while using the app, huge consumption of system resources such as memory or energy when the user is not actively using the app, loss of progress and data if the user leaves the app and returns to it at a later time or simply changes the device orientation while using the app [4, 42, 43, 6, 28, 44, 45, 7, 46, 47, 48, 49, 50]. Therefore, AGETs and testing processes should devote particular attention to verify the correct runtime app state change management by exercising the apps through mobile-specific events, such as sending an application to the background and resuming it, receiving a call, or changing the orientation of the device. In particular, this thesis pays special attention to crashes and GUI failures that consist in the manifestation of unexpected GUI states [51, 52].

This dissertation aims to contribute to the Software Engineering literature by providing solutions to overcome these limitations of AGETs in the mobile domain. The proposed solutions target Android, the world's most popular mobile Operating System (OS). The Android OS commercial success along with its open-source nature, led the researchers to focus on this mobile platform. Thus, most of the techniques and tools presented in literature are implemented and evaluated in the Android context.

1.2 Thesis Contributions

This Thesis work contributes to the literature in Software Engineering towards improving the automated GUI exploration techniques for mobile software applications. More specifically, it includes the following contributions:

- Two exploratory studies that aim at investigating the diffusion of GUI failures due to the mobile-specific event of changing the screen orientation, their key characteristics, and the faults causing them. The former study addresses the context of open-source Android apps, while the latter one considers very popular Android apps from Google Play Store. The studies exploit a novel classification framework that distinguishes three main classes of GUI failures. All the failures reported in the studies have been classified and made available in publicly shared documents.

- A fully automated event-based testing technique that explores Android apps for detecting issues tied to the Android Activity lifecycle, i.e. GUI failures and crashes. The technique has been implemented in a tool whose binaries has been made available for free download.

- A hybrid GUI exploration technique that combines automated GUI exploration with capture and replay to manage GUIs that need to be solicited by specific

user input event sequences to allow the exploration of parts of the app that cannot be reached otherwise. These GUIs are automatically detected by exploiting a machine learning approach and exercised by leveraging input event sequences provided by the user.

This dissertation includes material from the following research papers already published in peer-reviewed journals or conferences:

- Domenico Amalfitano, Vincenzo Riccio, Ana Cristina Ramada Paiva, and Anna Rita Fasolino (2018). Why does the orientation change mess up my Android application? From GUI failures to code faults. *Software Testing, Verification and Reliability, 28(1)*. Wiley. doi:10.1002/stvr.1654. [53]

- Domenico Amalfitano, Vincenzo Riccio, Nicola Amatucci, Vincenzo De Simone, and Anna Rita Fasolino (2019) Combining Automated GUI Exploration of Android apps with Capture and Replay through Machine Learning. *Information and Software Technology, 105(1)*. Elsevier. doi:10.1016/j.infsof.2018.08.007. [35]

- Vincenzo Riccio, Domenico Amalfitano, and Anna Rita Fasolino (2018). Is This the Lifecycle We Really Want? An Automated Black-Box Testing Approach for Android Activities. (In press) In the *Proceedings of The Joint Workshop of 4th Workshop on UI Test Automation and 8th Workshop on TESting Techniques for event BasED Software (INTUITESTBEDS 2018)*. ACM. [54]

1.3 Thesis Outline

The remainder of this dissertation is organized as follows:

- Chapter 2 reports the background of this work. It provides an overview of the Android platform. Moreover, it introduces the automated GUI exploration techniques for Android apps;

- Chapter 3 describes two exploratory studies that investigate the GUI failures exposed in Android apps by the mobile-specific event of changing the screen orientation. I carried out these studies with the support of my research group and of Prof. Ana C. R Paiva from the University of Porto (FEUP). The studies involved both open-source and apps from Google Play that were specifically tested exposing them to orientation change events;

- Chapter 4 shows the automated GUI exploration technique I designed with the support of some members of my research group to detect issues tied to the Android Activity lifecycle. The chapter includes an experimental evaluation involving real Android apps that shows the effectiveness of the proposed approach in finding GUI failures and crashes tied to the Activity lifecycle;

- Chapter 5 proposes a novel hybrid GUI exploration technique I designed and implemented in collaboration with my research group. It combines automated GUI exploration with capture and replay and leverages machine learning to pragmatically integrate these two approaches. The chapter includes an experimental evaluation showing the benefits introduced the hybridization in the app exploration.
- Chapter 6 reports conclusive remarks and possible future work.

Chapter 2

Background

2.1 The Android Platform

Mobile platforms introduced innovative features compared to traditional software applications, e.g., new interaction gestures, new development concepts such as context awareness, the ability to target multiple devices with different characteristics [41]. These features made mobile apps successful but introduced new challenges to software developers and testers.

Introduced by Google in 2007, Android is today the world's most popular mobile operating system (OS). Suffice it to say that Android accounted for around 88 percent of all smartphone sales to end users worldwide in the second quarter of 2018[5].

In this Section, I will introduce the Android platform and some of its features that are relevant for this disseration.

2.1.1 The Android Platform Architecture

Android platform architecture is designed as a stack of components. The stack layout, as defined by the official Android Developer Guide[6], is shown in figure 2.1.

The foundation of the Android platform is a modified version of the Linux Kernel that is responsible of providing the core services of the system. This kernel has been in widespread use for years, and is used in millions of security-sensitive environments. It has been adopted by Android to take advantage of key security features such as process isolation and the user-based permissions model[7]. Moreover, it allows device manufacturers to develop hardware drivers for a well-known kernel.

The layered architecture allows the developers to build portable apps by simply reusing core, modular system components and services. This happens thanks to the abstraction of the inner mechanisms and of the hardware details through standard interfaces.

[5] https://www.statista.com/statistics/266136/
[6] https://developer.android.com/guide/platform/
[7] https://source.android.com/security/overview/kernel-security.html

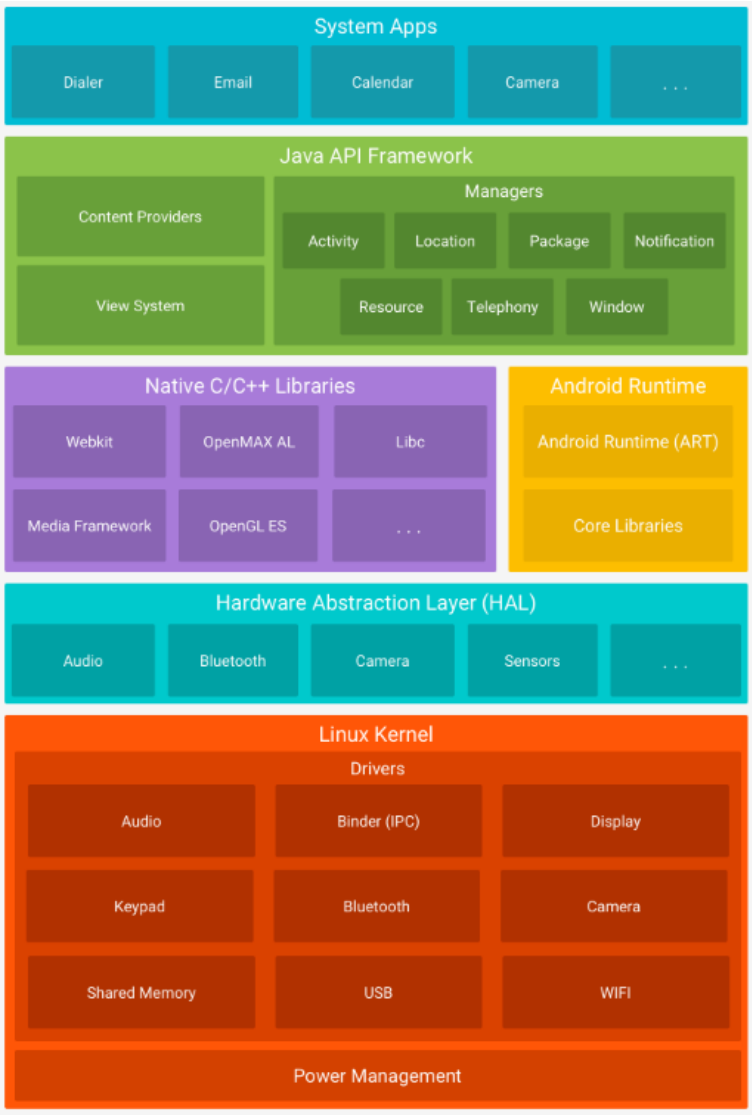


FIGURE 2.1: The Android Platform Architecture

2.1.2 Android App Components

Unlike traditional applications, Android apps do not contain any particular "main" method. An Android app can have multiple app components that are the entry points through which the system or a user can enter an app. Therefore, app components are the essential building blocks of Android apps.

There are four different types of app components: Activities, Services, Broadcast receivers, and Content providers. Each type serves a distinct purpose and has a distinct lifecycle that defines how the component is created and destroyed.

Activities, Services, and Broadcast Receivers are activated by asynchronous messages called Intents sent at runtime. This allows an app component to request actions from other components, whether they belong to the same app or another one.

In the following, I describe the four types of app components.

2.1.2.1 Activity

Activities are the entry points for interacting with the user. An Activity can be seen as a single GUI through which the users can access the features offered by the app.

An app has a *launcher* Activity that is the Activity that is shown to the user when the app is started.

Each Activity in an app is independent from the others. However, any of the Activities of an app can be started by other Activities that can even belong to a different app, if allowed. This enables a cohesive user experience by allowing user flows also between different apps.

2.1.2.2 Service

A Service is a general-purpose entry point that keeps an app running in the background to perform long-running operations or to perform work for remote processes. Therefore, it does not provide a user interface.

Another app component can thus start and interact with a Service without blocking the user interaction with computationally intensive operations.

2.1.2.3 Broadcast Receiver

A Broadcast Receiver is a component that allows an app to respond to system-wide broadcast announcements. They are another entry into the app outside of a regular user flow. Therefore, the system can deliver broadcasts to registered apps even if they aren't currently running.

Broadcasts can be initiated either by the system (e.g. a broadcast announcing that the battery is low) or by apps, for example, to let other apps know that some data has been downloaded to the device and is available to them.

Although broadcast receivers don't display a user interface, they may create a status bar notification to alert the user when a broadcast event occurs.

2.1.2.4 Content Provider

A Content Provider is a component that provides a standard interface to manage app data stored in a persistent storage location, such as a local SQL database or a remote repository.

The provided data can be shared among different apps or private to a specific one. The Android system offers a set of Content Providers such as for Contacts and Messages.

To access an app data, apps must have the needed permissions and the URI of the Content Provider. Therefore, the data of an app can be queried or modified by other apps through the content provider, if the content provider allows it.

2.1.3 Android Activity Lifecycle

An Activity is implemented as a subclass of the `Activity` class, defined in the Android Framework. The Activity instances exercised by the user are managed as an *Activity stack* by the Android OS. A user usually navigates through, out of, and back to an app but only the Activity at the top of the stack is active in the foreground of the screen. To ensure a smooth transition between the screens, the other Activities are kept in the stack. This allows the user to navigate to a previously exercised Activity without losing its progress and information. The system can decide to get rid of an Activity in background to free up memory space.

To provide this rich user experience, Android Activities have a proper lifecycle, transitioning through different states. Figure 2.2 shows the Activity lifecycle as it is illustrated in the official Android Developer Guide[8]. The rounded rectangles represent all the states an Activity can be in; the edges are labeled with the callback methods that are invoked by the Android platform when an Activity transits between states.

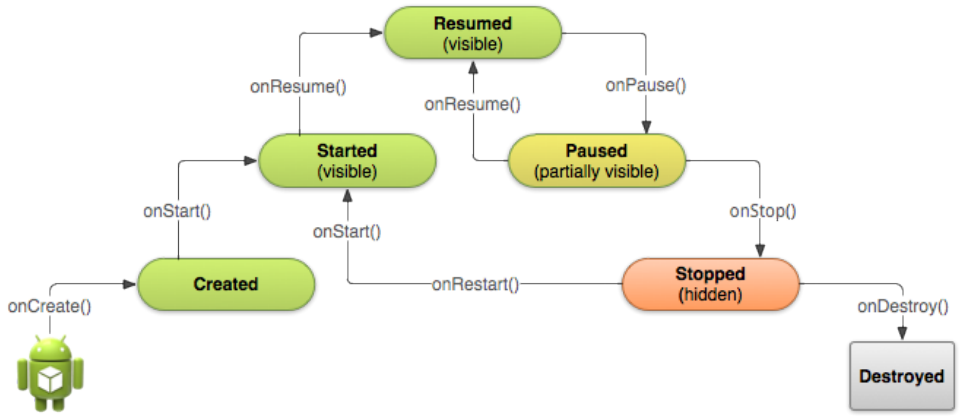


FIGURE 2.2: The Android Activity Lifecycle

The Activity visible in the foreground of the screen and interacting with the user is in the `Resumed` state, either it is created for the first time or resumed from the `Paused` or `Stopped` states.

When an Activity has lost focus but is still visible (e.g., a system modal dialog has focus on top of the Activity), it is in the `Paused` state; in this state, the app usually maintains all the user progress and information.

When the user navigates to a new Activity, the previous one is put in the `Stopped` state; it still retains all the user information but it is no longer visible to the user.

However, when an Activity is in `Paused` or `Stopped` states, the system can drop it from memory if the system resources are needed by other apps and therefore the Activity transits to the `Destroyed` state.

When it is displayed again to the user, the Activity is restarted and its saved state must be restored.

[8] https://developer.android.com/reference/android/app/Activity.html

The Android framework provides 7 callback methods that are automatically invoked as an Activity transits to a new state. They can be overridden by the developer to allow the app to perform specific work each time a given change of the Activity state is triggered. These callbacks are:

- *onCreate():* is called when the Activity is created for the first time. It is used by the programmer to perform the fundamental setup of the Activity such as binding data to lists and instantiating class-scope variables.
- *onStart():* makes the Activity visible to the user. This method is where the app initializes the code that maintains the UI.
- *onRestart():* is called if the Activity was stopped and is being re-displayed to the user, i.e. the user has navigated back to it.
- *onResume():* is called right before the Activity starts to interact with the user. It can be used by the programmer to begin animations or open exclusive-access devices, such as the camera.
- *onPause():* is called when the user is leaving the Activity, i.e. the Activity is going in the background. The programmer can override this method to pause or adjust operations that should not continue while the Activity is paused but are expected to resume shortly.
- *onStop():* is called when the Activity is no longer visible to the user. In this method, the programmer should release or adjust resources that are not needed while the app is not visible to the user, e.g. pause animations or switch from fine-grained to coarse-grained location updates. Moreover, the programmer should override this method to perform relatively CPU-intensive shutdown operations such as saving information to a database.
- *onDestroy():* is called before the Activity is destroyed, either because the user explicitly dismisses it or it is being destroyed by the system because more resources are needed by Activities with an higher priority or the system is temporarily destroying the Activity due to a configuration change (such as device rotation). The programmer should override this method to release all resources that have not yet been released by earlier callbacks.

When an Activity is destroyed due to normal app behavior, such as when the user presses the Back button, the Activity instance is lost; this behavior matches the user's expectations. However, if the system destroys the Activity due to a configuration change or the lack of memory, the user expects to preserve progress and data. Therefore, although the actual Activity instance is gone, the system remembers that it existed. In this case, when the user navigates back to the Activity, the system creates a new instance of that Activity using a set of saved data that describes the state of the Activity when it was destroyed. The saved data that the system uses to restore

the previous state is called the *instance state* and is a collection of key-value pairs stored in a *Bundle* object. When the Activity instance is destroyed and then recreated, the state of the layout is restored to its previous state with no code required by the programmer. However, the Activity might have more state information that should be restored, such as member variables that track the user's progress in the Activity. To this aim, the programmer can override the following callback methods:

- *onSaveInstanceState:* is called as the Activity begins to stop, so the Activity can save state information to an instance state bundle. The default implementation of this method saves transient information about the state of the Activity's layout, such as the text in an editable field or the scroll position of a list. To save additional instance state information for the Activity, the programmer must override this callback method and add key-value pairs to the Bundle object that is saved in the event that the Activity is destroyed unexpectedly.

- *onRestoreInstanceState():* is called right after the onStart() method. This method allows to recover the saved instance state from the Bundle that the system passes to an Activity that is recreated after it was previously destroyed.

The awareness of Android Framework features can help the programmers to develop dependable apps that behave the way users expect. Instead, if the developers do not take into proper account the lifecycle of the Activity components, their apps may show several issues, e.g. crashes if the user receives a phone call while using the app, huge consumption of system resources such as memory or energy when the user is not actively using it, loss of progress and data if the user leaves the app and returns to it at a later time or changes the device orientation while using the app.

The official Android Developer Guide stresses the relevance of the Activity lifecycle feature and warns the developers of the threats it introduces in several sections; therefore, it provides recommendations and guidelines to help programmers in the correct handling of the Activity lifecycle[9].

Despite this, several works in the literature have pointed out that mobile apps, including industrial-strength ones, suffer from issues that can be attributed to Activity lifecycle mishandling [4, 42, 43, 6, 28, 44, 45, 7, 46, 47, 48, 49, 50]. Zein *et al.* [3] performed a systematic mapping study of mobile application testing techniques involving 79 papers and identified possible areas that require further research. Among them, they emphasized the need for specific testing techniques targeting Activity lifecycle conformance.

For this reason, in this dissertation I will study the issues introduced by the Activity lifecycle mishandling and propose testing techniques and tools able to detect them.

[9] https://developer.android.com/guide/

2.1.4 Android Graphical User Interface

The GUI state rendered by an app is particularly relevant to this dissertation. In fact, automated GUI exploration techniques usually exploit the information that an app exposes at runtime through its GUI to derive the set of possible input events that can be fired. Moreover, I will address the issue of GUI failures, i.e. the manifestation of unexpected GUI states, exposed by events that exercise the Activity lifecycle.

In Android, the structure of an app GUI is defined as a `UI Layout`. The Android framework provides a variety of pre-built UI components such as structured layout objects and UI controls that allow the developers to build the GUI for their apps[10].

All of the views in a app screen are arranged in a single hierarchy of `View` and `ViewGroup` objects. The Android View Hierarchy, as described in the official Android Developer Guide[11], is depicted in figure 2.3.

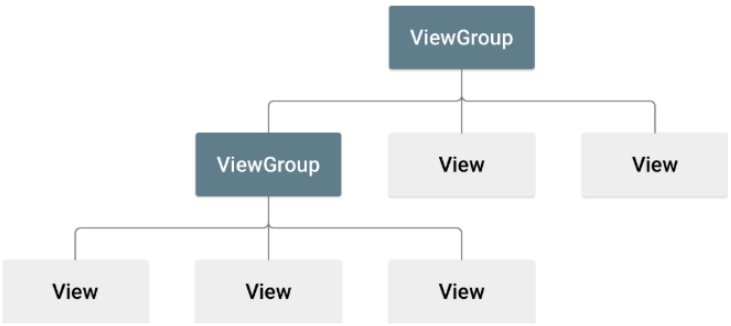


FIGURE 2.3: The Android View Hierarchy

A View object usually draws on screen an interactive object. Android defines a number of specialized subclasses of views that act as controls or are capable of displaying text, images, or other content. Therefore, View objects can be one of many predefined subclasses, such as `Button` or `TextView`. Moreover, a user can implement a custom View by overriding some of the View standard methods defined by the framework[12].

Instead a ViewGroup is an invisible container that defines the UI layout structure. It can contain View and other ViewGroup objects. The ViewGroup objects may present different layout structure and can be one of many predefined types, such as `LinearLayout` or `ConstraintLayout`, or be customized by developers.

A layout can be declared:

- in XML files. UI elements can be declared in XML language by using the XML vocabulary provided by Android that corresponds to the predefined View classes and subclasses. Declaring the UI layout in XML allows to separate the presentation of the app from the code that controls its behavior. Using XML

[10] https://developer.android.com/guide/topics/ui/
[11] https://developer.android.com/guide/topics/ui/declaring-layout
[12] https://developer.android.com/reference/android/view/View

files also eases the definition of different layouts for different screen sizes and orientations.

- in the app code. The View and ViewGroup objects can be defined programatically and the layout will be instantiated at runtime. However, the app's default layouts can be declared in XML, but their features can be modified programatically.

2.2 Automated GUI Exploration of Android Apps

Event-Driven Architecture is a software architectural pattern that is usually applied to design and implement applications where loosely coupled software components and services are able to notify of and to react to significant state changes (*events*) [21].

Mobile apps can be considered as Event-Driven Systems that are able to analyze and react to different types of events. Those event types include events triggered on the Graphical User Interface (GUI), events related to the external environment and sensed by device sensors, events generated by the device hardware platform (such as battery, USB, headphones and other external peripheral ports), and events generated by other apps running on the mobile device [24].

For assessing the quality of mobile apps, it is possible to extend and adapt existing automated techniques designed for GUI-based Event-Driven Systems, such as Automated GUI Exploration Techniques (AGETs). These techniques have been already adopted to automatically assess the quality of GUI-based desktop applications [55, 56, 57] and Web applications [58, 59]. AGETs iteratively explore the behavior of the running application by exploiting the information that it exposes at runtime through its GUI to derive the input events to be fired.

2.2.1 Generic GUI Exploration Algorithm for Android Apps

In an earlier work [25], my research group analyzed a set of 13 Android testing techniques implementing AGETs and abstracted in a general framework the characteristics of the different GUI exploration approaches. They presented a unified algorithm that abstracts the workflow of existing automated GUI exploration techniques. The novel exploration techniques I will propose in Chapters 4 and 5 extend this algorithm.

The unified GUI exploration algorithm foresees the iterative execution of the sequential activities of Current GUI Description, Input Event Sequence Planning, Input Event Sequence Execution and Termination Condition Evaluation until a predefined Termination Condition is met. The workflow of the automated GUI exploration is described by the UML Activity diagram shown in Fig. 2.4 where the Activity states describe the steps of the algorithm.

Each app exploration is started by the *App Launch* step that installs and launches the app on an Android device.

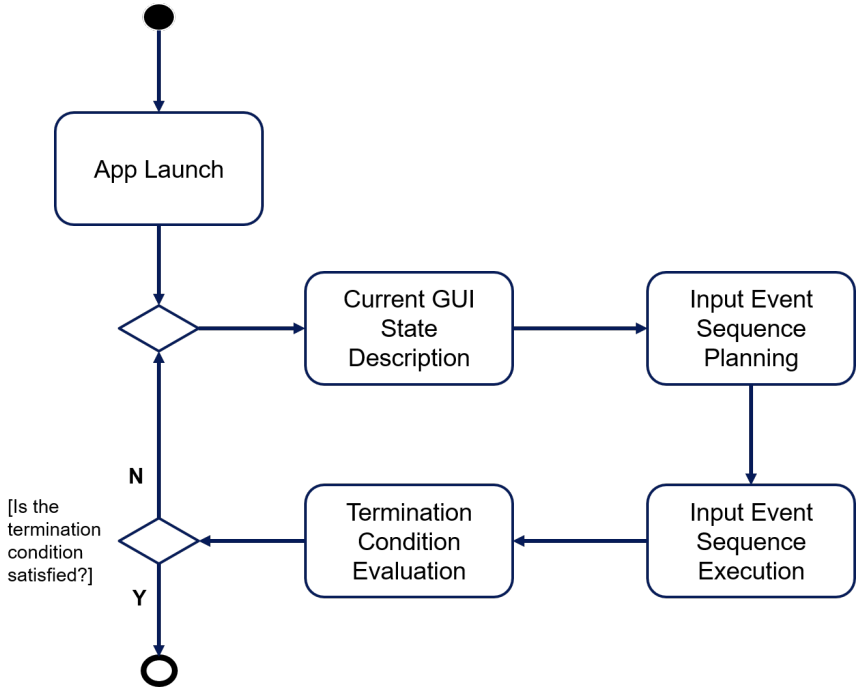


FIGURE 2.4: The Automated GUI Exploration

In the *Current GUI Description* step, a representation of the GUI state currently exposed by the app is inferred according to an abstraction criterion [60].

The *Input Event Sequence Planning* step selects the next input event sequence to fire among all the ones triggerable on the current GUI. The selection of the input event sequence to run is done according to a scheduling strategy. This strategy defines how the GUI is explored and spans from pure random exploration [26, 27, 28, 29] to systematic exploration strategies that rely on an app model [30, 31, 32, 33, 34].

In the *Input Event Sequence Execution* step the next planned input event sequence is actually executed.

Finally, the *Termination Condition Evaluation* step evaluates whether the termination condition is met and the exploration can be stopped. The termination condition may be based on aspects of the exploration process, such as the number of events that have been fired or the amount of time spent testing, or it may be based on some adequacy measurement that determines whether the app has been sufficiently explored, e.g. statement coverage criterion [61].

2.2.2 Challenges to Automated GUI Exploration of Android Apps

AGETs have been successfully adopted in the context of mobile apps for supporting critical engineering tasks such as testing [2], reverse engineering [36], network traffic generation and analysis [37, 38], performance and energy consumption analysis [18].

Although AGETs provide a viable approach for automatically exercising Android mobile apps, they suffer from some limitations that may hinder them from a thorough app exploration.

Some AGETs have been presented in the literature to address the issues that can be attributed to Activity lifecycle mishandling such as crashes [6, 7] or resource leaks [46]. Only one of them [28] addressed GUI failures, that consist in the manifestation of an unexpected GUI state. However, their authors only considered the issues tied to the orientation change event, potentially neglecting the ones tied to other events that exercise the Activity lifecycle. GUI failures tied to the Activity lifecycle are currently a widespread problem in the context of mobile apps, as evidenced by the exploratory studies I will describe in Chapter 3. Therefore, in Chapter 4 I will present a novel fully-automated testing technique that adopts a GUI exploration strategy that systematically exercises the lifecycle of app Activities to expose GUI failures and crashes.

Another limitation intrinsic in automated exploration approaches consists of not being able to replicate human-like interaction behaviors. In fact, some app features need to be exercised by exploiting app-specific knowledge that only human users can provide. As an example, these techniques often fail in exploring relevant parts of the applications that can be reached only by firing complex sequences of input events on specific GUIs and by choosing specific input values [39, 40]. In this dissertation, I refer to these GUIs as *Gate GUIs*. There may be several types of Gate GUIs in real apps, such as GUIs in which the users need to enter their credentials in order to create a new account or to access to functionality offered by the app to authenticated users only, GUIs that require the users to correctly configure the settings of services they intend to use through the app, or GUIs that request the users to scan a valid QR code through the device camera to access to particular app features. In Chapter 5, I will present a novel hybrid GUI exploration technique that exploits the human involvement in the automated exploration process to overcome the limitations introduced by Gate GUIs.

Chapter 3

Exploring GUI Failures due to Orientation Changes

In this Chapter, I investigate the failures exposed in mobile apps by the mobile-specific event of changing the screen orientation. I focus on GUI failures that consist in unexpected GUI states rendered by the apps. These failures should be avoided in order to improve the apps quality and to ensure better user experience. I propose a classification framework that distinguishes three main classes of GUI failures due to orientation changes and exploit it in two studies that investigate the impact of such failures in Android apps. The studies involved both open-source and industrial-strength apps that were specifically tested exposing them to orientation change events. The results showed that more than 88% of these apps were affected by GUI failures, some classes of GUI failures were more common than others, and some GUI objects were more frequently involved. The app source code analysis allowed me to identify 6 classes of common faults causing specific GUI failures.

3.1 Introduction

Mobile apps are event-driven systems able to analyze and react to events additional to those available for traditional desktop or Web applications, such as receiving a phone call, changing the state of the network connections or changing the orientation of the device. Therefore, mobile app developers should devote particular attention to verify the behavior of their apps when subjected to mobile-specific events.

Among these types of events, the orientation change deserves special attention. It is a peculiar event in mobile platforms that results in the switch of the running app between portrait and landscape layout configurations. Moreover, it causes the Activity instance currently on screen to be destroyed and then recreated in the new configuration according to the Activity lifecycle described in Section 2.1.3. Android guidelines recommend that, when the orientation change event occurs, the app adapts itself to the new layout, avoiding memory leaks, and preserving its state and any significant stateful transaction that was pending. Unfortunately, the implementation of these recommendations is not straightforward and introduces programming challenges to Android programmers. Several works in the literature have pointed

out that many mobile apps actually crash or show failures that can be attributed to orientation change mishandling [4, 42, 43, 6, 28, 44, 45, 7, 46, 47, 48, 49, 50].

GUI failures are a relevant class of failures that may disrupt the user experience. They consist in the manifestation of an unexpected GUI state [51, 52]. If an Android app does not correctly handle orientation change events, it may expose several types of GUI failures, e.g. unexpected GUI objects may appear in wrong positions, objects may be rendered with wrong properties, or important objects may be missing from the GUI. A GUI failure may involve different types of GUI objects and there may be object types that are more likely to be involved than others. These failures may be caused either by application logic errors, or by faults in the code that uses Android-specific programming features. As a consequence, studying this type of GUI failures and classifying them according to their characteristics may be useful both for defining testing techniques able to detect them, and for preventing the introduction of code faults causing them.

In this Chapter, I propose a novel framework for classifying GUI failures. This classification framework is exploited by two different exploratory studies that aimed at investigating their diffusion, the key characteristics, and possible faults causing them. I carried out these studies with the support of members of my research group and of Prof. Ana C.R. Paiva from the University of Porto (FEUP). The former study addressed the context of open-source Android apps, while the latter one considered popular Android apps belonging to the Google Play Store. In both studies, the apps testing led to the detection of a considerable number of GUI failures due to the orientation change. These failures were validated, classified, and made available in publicly shared documents. In the former study, we also analyzed the source code of a subset of applications exposing the most frequent types of failure and thus discovered six classes of common faults causing them made by Android developers.

This work contributes to the Android community in several ways. It may help in the definition of a fault model specific to Android apps in order to develop testing techniques that can allow developers to find faults in apps before release, especially in the parts of the code that use new programming features [62]. Moreover, it can enable the definition of additional mutation operators specific to Android apps and, possibly, of static analysis techniques suitable for early bug detection. Lastly, the descriptions of GUI failures provided by the exploratory studies may be exploited to evaluate and compare the effectiveness of different testing techniques and tools.

The remainder of the Chapter is structured as follows. Section 3.2 presents some examples of real GUI failures due to orientation changes that motivated us to explore this issue. Section 3.3 illustrates the framework we defined for characterizing GUI failures due to orientation changes. Section 3.4 presents the first exploratory study we performed for finding GUI failures in real Android open source applications, classifying them, and discovering common faults causing some of these failures. Section 3.5 reports a second exploratory study that aimed at finding and classifying

DOC GUI failure in popular Android apps from Google Play Store. Section 3.6 discusses the threats that could affect the validity of the exploratory studies. Section 3.7 provides related work. Section 3.8 finally draws the conclusions and presents future work.

3.2 Motivating Examples

In this section, I present 4 examples of GUI failures due to screen orientation changes in mobile apps. I found these failures by manually testing 4 different real mobile applications. In particular, I solicited the apps by using the Double Orientation Change (DOC) event that consists in a sequence of two consecutive orientation change events. To detect a GUI failure, I compared the GUIs before and after this event. I used the DOC event because the application of a single orientation change may not be sufficient to detect GUI failures, as some minor differences in GUI content or views are indeed acceptable between landscape and portrait orientations[13], and the GUI state of the app may differ after a single orientation change event. After a second consecutive orientation change, the GUI content and layout should be the same as before the first orientation change, otherwise there is a GUI failure [42, 43].

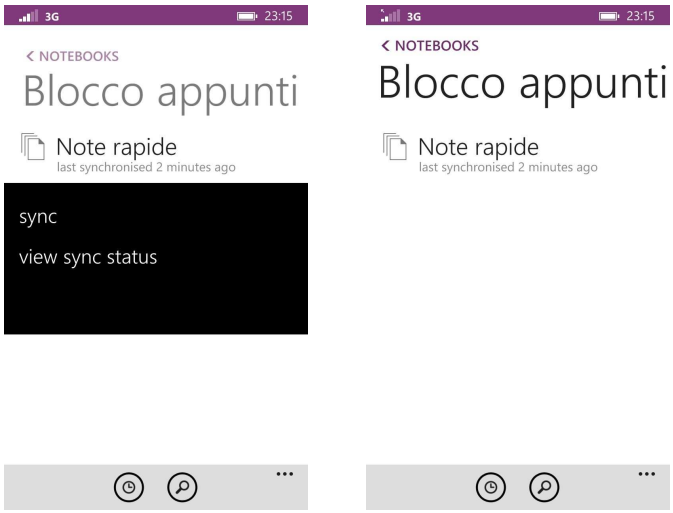
The first example of GUI failure occurs in the digital note-taking application OneNote running on Windows 10 Mobile OS. Figure 3.1 shows this failure. In this case, when the user performs a long press on a notebook in the list, a contextual menu appears displaying the actions that can be performed on the selected document such as syncing it, as shown in Figure 3.1(a). After the DOC, the contextual menu disappears as reported in Figure 3.1(b). In this case the application does no longer provide the features for managing the selected notebook.

The second failure occurs on the Gmail app version 6.8.130963407 running on a device equipped with Android 6.0. This failure is shown in Figure 3.2. If the user performs a long press on a mail in the list and selects "Other Options" in the application bar, then an action overflow menu appears. The menu displays the actions that can be performed on the selected mail such as moving it or reporting it as spam, see Figure 3.2(a). After a double orientation change of the device the menu disappears from the user interface, as shown in Figure 3.2(b)).

This kind of failures also occur in lesser-known apps such as Agram, an Android application that displays anagrams in English. Figure 3.3 shows a failure exposed by Agram version 1.4.1. If the user chooses to create random words, a Dialog appears prompting the number of words he wants to generate (see Figure 3.3(a)). When the user changes the orientation of the device twice, the dialog disappears and a list of random words is rendered on the screen as shown in Figure 3.3(b).

The last example regards a failure exposed by the mail client application preinstalled in iOS version 9.3.1. This failure is shown in Figure 3.4. The user can select one or more incoming messages he wants to manage, as reported in Figure 3.4(a).

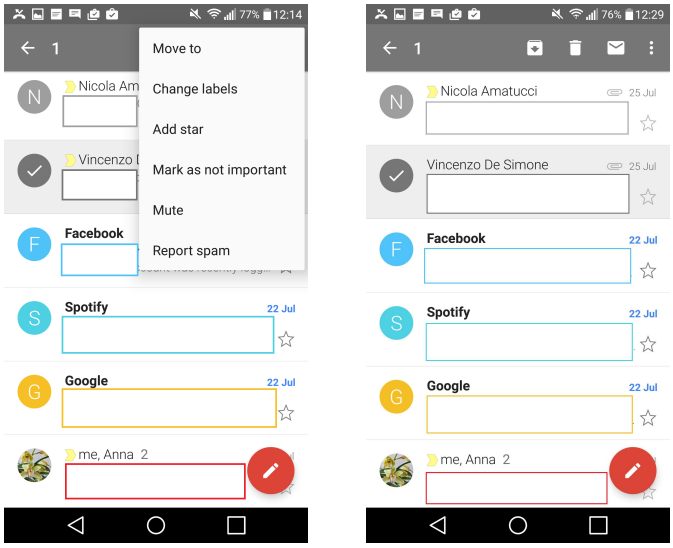
[13] https://developer.android.com/docs/quality-guidelines/core-app-quality



(a) GUI before the two orientation changes

(b) GUI after the two orientation changes

FIGURE 3.1: GUI failure exposed by OneNote running on Windows 10 Mobile OS



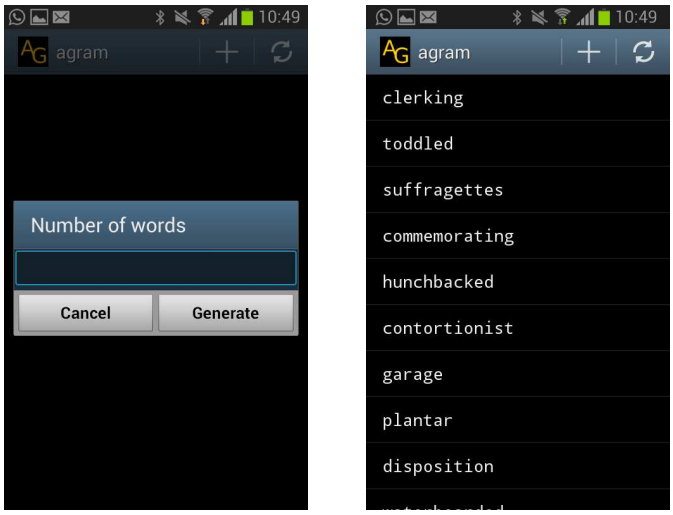
(a) GUI before the two orientation changes

(b) GUI after the two orientation changes

FIGURE 3.2: GUI failure exposed by Gmail running on Android OS

After the double orientation change of the device, the application does not preserve the mail selection made by the user, as shown in Figure 3.4(b). As a consequence, the UI widgets allowing to handle the selected mails (*i.e.*, Cancel, Mark, Move and Delete) disappear and different ones appear on the user interface.

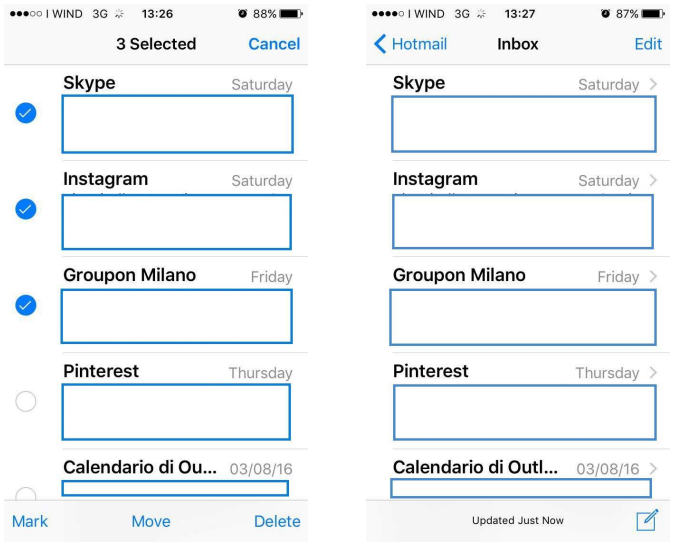
As these examples show, the observed GUI failures concerned apps from all the major mobile platforms, i.e., Android, iOS, and Windows. They affected even popular applications and applications usually bundled in mobile devices as pre-installed software. Even if such failures may not be considered as critical as app crashes,



(a) GUI before the two orientation changes

(b) GUI after the two orientation changes

FIGURE 3.3: GUI failure exposed by Agram running on Android OS



(a) GUI before the two orientation changes

(b) GUI after the two orientation changes

FIGURE 3.4: GUI failure exposed by the iOS mail client app

their effects may have a negative impact on the user experience and contribute to poor user ratings. These examples suggest that this problem may be relevant and widespread and worth to be further investigated.

3.3 The DOC GUI Failure Classification Framework

In GUI testing, a GUI failure can be defined as a runtime manifestation of an unexpected GUI state [52]. When an app is exercised by a DOC, the expected GUI

state should be the same as before the DOC, unless the GUI specifications prescribe a different behavior.

As a consequence, any discrepancy we observe between the GUI state before the DOC (referred to as *start state*) and the GUI state after the DOC (referred to as *end state*) may represent the manifestation of a GUI failure.

As the examples in Section 3.2 showed, GUI failures may involve different types of GUI objects and may manifest themselves in diverse ways. Hence, we decided to classify these failures in terms of 2 attributes called *scope* and *mode*, respectively. The *scope* attribute represents the type of GUI object involved in the manifested GUI failure. More precisely, the *scope* of a failure can be one of the types of GUI objects used to implement the app GUI in the considered mobile platform P. For instance, the GUI object types offered by the Android platform include Button, ContextMenu, Dialog, TextView, *etc.*. More in general, $S(P)$ indicates the set of possible GUI object types offered by the platform P.

The *mode* attribute indicates how the failure manifested itself in the GUI end state. In accordance with the IEEE Standard Classification for Software Anomalies [63] and other GUI failure classification models proposed in the literature [64], this attribute can assume one of the 3 following values:

- *Extra*: Some GUI Objects are present that should not be. In this case, there are one or more objects appearing in the *end state* of the GUI that were not present in the *start state*;
- *Missing*: Some GUI Objects are absent that should be present. This failure happens when there are one or more objects contained in the *start state* that are no longer present in the *end state*;
- *Wrong*: Some GUI Objects are displayed in an incorrect state. This failure happens when one or more objects of the *start state* are contained in the *end state* but look different.

Using these 2 attributes, any DOC GUI failure can be characterized by a couple $(mode, scope)$, where:

- $mode \in M = \{Extra, Missing, Wrong\}$.
- $scope \in S(P)$.

Figure 3.1 provides an example of a GUI implemented in the Windows Phone Toolkit platform. This GUI presents a failure that can be characterized by the couple (Missing, Context Menu), because the GUI after the DOC misses the Context Menu shown in the former GUI.

Figure 3.2 and Figure 3.3 show two examples of DOC GUI failures I observed in the Android context. Figure 3.2 provides another example of Missing mode GUI failure involving an ActionOverflowMenu object, whereas Figure 3.3 presents a more

complex case where the same DOC event triggered two distinct failures, *i.e.*, a Missing failure and an Extra failure. The GUI after the DOC event presents indeed an extra ListView object and, at the same time, it misses the Dialog object that was rendered on the former GUI.

Figure 3.4 provides an example of 3 DOC GUI failures I observed in the iOS context. In accordance with the proposed classification framework, these failures can be characterized by the couple (Wrong, UITableView), since the *selected* property of the items in the UITableView changes state after the double orientation change.

In the following, I provide a set of definitions that can be used to formalize the DOC GUI failures and to classify them with respect to their *mode* and *scope* attributes.

3.3.1 GUI Objects

DOC GUI failures involve one or more GUI objects. A GUI object is a graphical element of the User Interface that is characterized by a set of properties, such as its type, position, size, background color, *etc.*, which vary with the type of the considered object.

Each object property assumes values that are drawn from a predefined set of values associated with that property. The set of properties of each object and the set of values each property may assume depend on the specific development framework used for implementing GUIs in the considered mobile platform.

When needed, the dot notation will be used for referring to the values assumed by the properties of a GUI object, i.e., the notation $o.p_i$ indicates the value v_j assumed by the property p_i of the object o.

Definition 1
P is the set of properties of all the GUI objects provided by a given mobile development framework (*i.e.*, Android SDK, iOS UIKit, Windows Mobile WPToolkit).

Definition 2
$\forall\ p_i \in P\ \exists!\ V_{p_i}$, where V_{p_i} is the set of all the possible values that p_i can assume.

Definition 3
A GUI object o is defined as:

$$o \triangleq \{(p_i, v_j) : p_i \in P, v_j \in V_{p_i}\}$$

Among all the properties of an object, the focus is on a subset that is crucial for identifying the object and defining its layout i.e., its type, position, and size. On the basis of the values assumed by these 3 properties, a GUI object is of a given type, is located on a precise position on the screen and occupies a specific area of the GUI. In the following, this subset is referred to as *fundamental GUI object properties* and indicated with P^*.

Definition 4
The set of fundamental GUI object properties P^* is defined as:

$$P^* \triangleq \{type, position, size\} \subset P$$

3.3.2 GUI State and State Transition

The GUI state is formally described as the set of GUI objects that it contains. In mobile applications, like in any other EDS, single events or event sequences may cause state transitions. The transition between two states, triggered by one or more events, is defined as a function that associates two states. The double orientation change is a sequence of two consecutive orientation change events. These concepts are explained by the following definitions:

Definition 5
GUI State S is defined by the set of GUI objects it contains.

$S \triangleq \{o_1, o_2, ..., o_n\}$.

Definition 6
Given an Application Under Test (AUT), the set of its Application States (AS_{AUT}) is defined by all the GUI States the AUT can render to the user:

$AS_{AUT} \triangleq \{S_1, S_2, ..., S_n\}$

Definition 7
An event e is a function that associates two GUI States of an AUT.

$e : AS_{AUT} \rightarrow AS_{AUT}$

Definition 8
An event sequence es is a ordered sequence $es =< e_1, \ldots, e_n >$ of n events, $n \geq 1$. Formally, an event sequence is a function that associates two GUI states of an AUT, since it is sequentially triggered starting from an initial GUI state of the AUT and reaches a final GUI state of the AUT.

$es : AS_{AUT} \rightarrow AS_{AUT}$

Definition 9
The DOC is an event sequence consisting in 2 consecutive *orientationChange* events.

$DOC \triangleq < orientationChange, orientationChange >$

3.3.3 Equivalence and Similarity between GUI Objects

The formal definition of DOC GUI failures relies on the equivalence and similarity relations between GUI states. These relations in turn depend on the equivalence and similarity relations between GUI component objects.

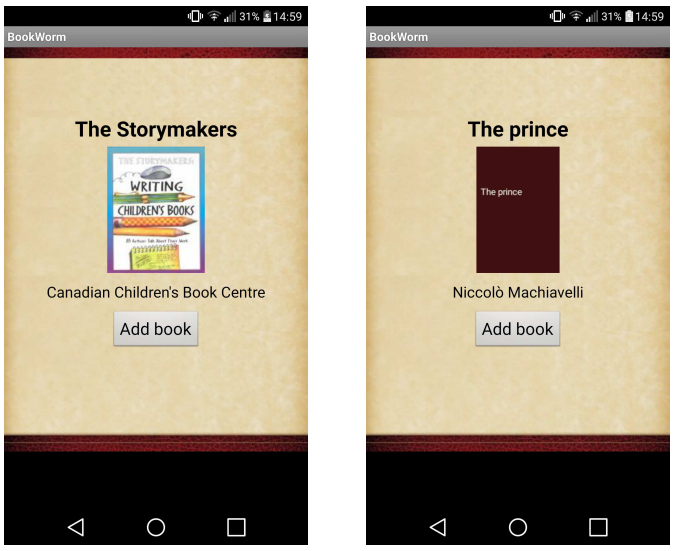
Two objects are considered *similar* if and only if their type, position, and size properties assume exactly the same values. Two objects are considered *equivalent* if and only if all their properties assume exactly the same values.

Two GUI states are considered *equivalent* if and only if for each object of the former GUI state there is exactly one equivalent object in the latter state, and at the same time for each object of the latter GUI state there is exactly one equivalent object in the former state.

Figure 3.5(a) and Figure 3.5(b) report two GUIs states exposed by the Bookworm app. They have two pairs of similar TextView objects since their 3 fundamental GUI object properties assume the same values, but these objects differ for their textual values, i.e. the TextViews of the GUI state in Figure 3.5(a) assume the textual value "The Storymakers" and "Canadian Children's Book Centre", whereas the equivalent objects in the GUI state in Figure 3.5(b) assume the value "The Prince" and "Niccolò Machiavelli", respectively.

Analogously, the ImageView objects in the two GUI states differ for the source value they assume, as it is evident from the different cover images they display. Instead, the Button object is equivalent among the two GUIs.

As a result, the considered GUI states are not equivalent since they include 3 not equivalent objects.



(a) (b)

FIGURE 3.5: Example of two GUI states that are similar but not equivalent according to the proposed classification framework

Definition 10
Two GUI objects o_i and o_j are similar when their fundamental properties values coincide, while the values of other properties may differ.

$$o_i \sim o_j \iff \begin{cases} \forall (p_u, v_t) \in o_i : p_u \in P^*, \exists! (p_q, v_k) \in o_j : p_u = p_q, v_t = v_k, \\ \forall (p_q, v_k) \in o_j : p_q \in P^*, \exists! (p_u, v_t) \in o_i : p_u = p_q, v_t = v_k \end{cases}$$

Definition 11
Two GUI objects o_i and o_j are equivalent when they have the same set of properties and each property assumes the same value in both objects.

$$o_i \cong o_j \iff \begin{cases} \forall (p_u, v_t) \in o_i, \exists! (p_q, v_k) \in o_j : p_u = p_q, v_t = v_k, \\ \forall (p_q, v_k) \in o_j, \exists! (p_u, v_t) \in o_i : p_u = p_q, v_t = v_k \end{cases}$$

Definition 12
The equivalence between two GUI States S_i and S_j is defined as follows:

$$S_i \cong S_j \iff \begin{cases} \forall o_t \in S_i, \exists! o_k \in S_j : o_t \cong o_k, \\ \forall o_k \in S_j, \exists! o_t \in S_i : o_k \cong o_t \end{cases}$$

3.3.4 DOC GUI Failures

The definitions presented so far can be exploited to formalize the different types of DOC GUI failures. Given the GUI start state `S` and the GUI end state `DOC(S)` reached after a DOC event, there is a DOC GUI failure if and only if S and DOC(S) are not equivalent.

Definition 13
Given a GUI State $S \in AS_{AUT}$, the double orientation change causes a DOC GUI failure if :

$$S \ncong DOC(S)$$

The definitions given in this Section are used to define the properties that can be checked to characterize a DOC GUI failure in terms of its mode and scope.

Definition 14
The GUI property that must be checked to assess the presence of an Extra GUI failure due to a DOC f_{DOC} is defined as follows:

$$\begin{array}{c} \exists\, o_j \in DOC(S) : o_j \nsim o_i, \\ \forall\, o_i \in S \end{array} \implies \begin{array}{c} \exists\, f_{DOC} : f_{DOC}.mode = Extra, \\ f_{DOC}.scope = o_j.type) \end{array}$$

Considering the GUI state in Figure 3.6(a) and the one in Figure 3.6(b) obtained after a DOC event, it can be noticed that there is a Dialog object appearing in the end state that is not present in the start state; thus, there is a DOC GUI failure having Extra mode and Dialog scope.

Definition 15
The GUI property that must be checked for assessing the presence of a Missing GUI failure due to a DOC f_{DOC} is defined as follows:

$$\begin{array}{l}\exists\, o_j \in S : o_j \nsim o_i, \\ \forall\, o_i \in DOC(S)\end{array} \Longrightarrow \begin{array}{c}\exists\, f_{DOC} : f_{DOC}.mode{=}Missing, \\ f_{DOC}.scope{=}o_j.type)\end{array}$$

Considering the GUI state in Figure 3.6(a) and the one in Figure 3.6(c) obtained after a DOC event, there are two Button objects in the start state that are no longer present in the end state; thus, there are 2 DOC GUI failures having Missing mode and Button scope.

Definition 16
The GUI property that must be checked for assessing the presence of a Wrong GUI failure due to a DOC f_{DOC} is defined as follows:

$$\begin{array}{l}\exists\, o_j \in S : \exists\, o(i) \in DOC(S), \\ o(j) \sim o(i)\,, o(j) \ncong o(i)\end{array} \Longrightarrow \begin{array}{c}\exists\, f_{DOC} : f_{DOC}.mode{=}Wrong, \\ f_{DOC}.scope{=}o_j.type)\end{array}$$

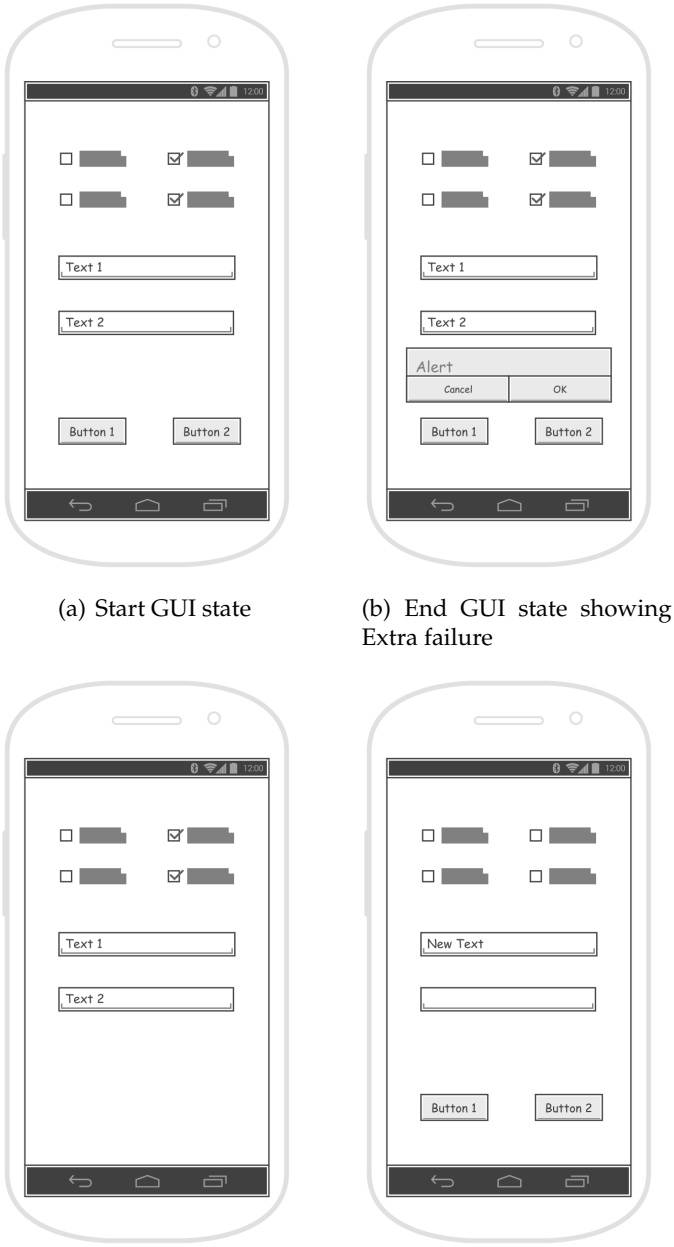
Considering the GUI state in Figure 3.6(a) and the one in Figure 3.6(d) obtained after a DOC event, there are 2 EditText objects in the start state that are still contained in the end state but have a different text value; thus, there are 2 DOC GUI failures having Wrong mode and EditText scope.

3.4 Exploratory Study of Open-Source Apps

The examples in Section 3.2 showed that DOC GUI failures affect mobile apps of all the major mobile platforms. This work investigates such failures in the context of Android apps due to the great success of this platform and to the large availability of apps in markets and open-source repositories.

The first study aims at exploring and classifying DOC GUI failures occurring in real Android apps. In this study, we considered open source apps, which allow the access to their source code. The study aimed at achieving the following goals:

G1 - to verify the spread of DOC GUI failures among real Android mobile apps.
G2 - to characterize the detected DOC GUI failures.
G3 - to find possible common faults causing DOC GUI failures.



(a) Start GUI state

(b) End GUI state showing Extra failure

(c) End GUI state showing Missing failure

(d) End GUI state showing Wrong failure

FIGURE 3.6: Example of Extra, Missing and Wrong GUI failures

To carry out this study, we followed an experimental procedure based on 5 steps: Objects selection, Apps testing, DOC GUI failures validation, DOC GUI failures classification, Common faults Identification.

3.4.1 Objects Selection

In this step, we selected a sample of apps from a repository of open source Android apps. We chose to consider F-Droid[14], a well-known repository of Free and Open

[14] https://f-droid.org/

Source Software (FOSS) applications for the Android platform. F-Droid offers direct access to apps' source code and to their developers through code repository and issue tracker. It has been used in many other studies on Android testing proposed in the literature [2, 9, 8] and contains a growing number of applications belonging to different categories.

In the selection activity, we required the apps to be candidate to expose a DOC GUI failure by allowing the orientation change of the screen. We also required the availability of the app developers, in order to contact them and receive their feedback about the DOC GUI failures detected in the study. Finally, we needed their opinion to discard the detected DOC GUI failures that were instead a manifestation of an expected GUI behavior, i.e. a feature of the analyzed application. To this aim, we used the three inclusion criteria listed below to select an object application from F-Droid:

1. **Issue tracker availability:** the app should be linked to its issue tracker;
2. **Active developers:** the app should have been updated in the last 12 months since the study started. In this way I, felt confident that the selected application was still maintained by its developers;
3. **Orientation change enabled:** the app should have at least one activity that provides both portrait and landscape screen layouts.

Table 3.1 shows how the dataset was built. When the study was performed, the F-Droid repository contained 2,030 apps, but only 1,807 of them provided an issue tracker. Among these 1,807 apps, 762 were updated in the last 12 months. Of the 762 apps, 685 allowed orientation changes. Finally, the dataset was built by randomly selecting the 10% of the 685 that met the inclusion criteria. Table 3.2 lists name, version and category of the 68 apps in the dataset that cover 14 of the 17 categories provided by F-Droid.

TABLE 3.1: Dataset 1 construction

Criteria	**# apps**
Apps in F-Droid	2030
Provide an issue tracker	1807
Updated in the last 12 months	762
Allow Orientation Change	685
Dataset	68

3.4.2 Apps Testing

The apps in the dataset were tested to find DOC GUI failures. To this aim, we exploited a test amplification strategy in which new test cases were obtained starting from an initial set of existing test cases. The approach of enhancing existing test

TABLE 3.2: Dataset 1

App	Version	Category	App	Version	Category
A Time Tracker	0.21	Time	**Odyssey**	1.1.0	Multimedia
A2DP Volume	2.11.11	Multimedia	**OpenFood**	0.4	Sports & Health
AFWall+	2.8.0	Security	**ownCloud News**	0.9.9.2	Internet
agram	1.4.1	Reading	**Padland**	1.3.2	Writing
Alarm Klock	2.2	Time	**PassAndroid**	3.3.2	Reading
Amaze File Manager	3.1.1	System	**Periodical**	0.3	Science & Education
AntennaPod	1.5.2.0	Multimedia	**Pinpoi**	1.4.3	Navigation
BankDroid	1.9.10.6	Money	**Port Knocker**	1.0.9	Security
BeeCount	2.4.1	Writing	**Prayer Times**	3.6.3	Time
Berlin-Vegan	2.0.7	Navigation	**Primary**	0.1	Science & Education
Blitzmail	0.6	Internet	**ReGex**	1.3.1	Games
Cache Cleaner	2.2.0	System	**Ruler**	1.0.1	Multimedia
Calendar Notifications Plus	1.3.21	System	**Shorty**	1.06	Multimedia
Chibe	1	Time	**Sieben**	1.9	Sports & Health
Colorpicker	1	System	**Silectric**	1.2.01	Money
Currency	1.04	Money	**Simple Dilbert**	4.2	Reading
DNS66	0.2.1	Internet	**Simple Solitaire**	2.0.1	Games
DroidShows	7.3.1	Multimedia	**Slide**	5.5.4	Reading
Etar	1.0.8	Time	**SpaRSS**	1.11.8	Reading
ExprEval	1	Science & Education	**StageFever**	1.0.9	Multimedia
File Manager	1.24	System	**SteamGifts**	1.5.2	Internet
FOSDEM companion	1.4.2	Time	**Step and Height counter**	1.2	Sports & Health
Gallery	1.48	Multimedia	**Stringlate**	0.9.3	Development
ImapNotes2	4.9	Internet	**SyncThing**	0.9.1	Internet
Iven News Reader	3.0.2	Reading	**Tap'n'Turn**	2.0.0	System
LeafPic	0.5.2	Multimedia	**Taskbar**	3.0.2	System
Legeappen	3	Sports & Health	**Transdroid Torrent Search**	3.7	Internet
Loop Habit Tracker	1.6.2	Sports & Health	**Transistor**	2.1.7	Multimedia
Lyrically	0.5	Multimedia	**Unit Converter Ultimate**	4.2	Science & Education
Malp	1.1.1	Multimedia	**uNote**	1.1.4	Writing
Mather	0.3.0	Science & Education	**Weather**	3.4	Internet
Network Monitor	1.28.10	Connectivity	**Who Has My Stuff?**	1.0.25	Money
NewPipe	0.8.7	Multimedia	**WiFi Analyzer**	1.6.5	Connectivity
NewsBlur	5.0.0b3	Reading	**World Clock & Weather**	1.8.5	Time

cases in the domain of Android mobile applications has been used by Zhang and Elbaum [65] and by Adamsen *et al.* [43].

12 master students from the University of Naples were involved to obtain the initial set of test cases. The students had been trained about automatic testing of GUI based software applications by techniques of Capture & Replay. Each student was asked to record a number of test cases sufficient to cover the features provided by four or five applications of the Dataset so that each application was tested by a single student. The students had one semester to accomplish their tasks. The students exploited the Robotium Recorder tool[15] for recording GUI test cases.

I automatically amplified the test cases recorded by the students by injecting after each user event a snippet of Robotium code that fires a double orientation change and checks the presence of the three DOC GUI failure modes through appropriate assertions. These assertions are based on the definitions given in 3.3.4. The code reported in the Listing 3.1 shows an example of an amplified test case for the *A Time Tracker* app. After each event recorded by the users, such as *EVENT 1* and *EVENT 2*, the test case is amplified by adding the code that describes the start GUI state (*Get GUI state before DOC*), fires the double orientation (*DOUBLE ORIENTATION*), describes the end GUI state (*Get GUI state after DOC*) and matches the two descriptions (*CHECK ASSERTIONS*).

[15] https://robotium.com/products/robotium-recorder

After this activity, the amplified test cases were replayed for testing the apps on a real LG G4 H815 device equipped with Android 6.0. We collected the GUI failures automatically detected by these test cases.

LISTING 3.1: Example of amplified test case

```
package com.markuspage.android.atimetracker.test;

import android.test.ActivityInstrumentationTestCase2;
import com.robotium.solo.*;
import com.markuspage.android.atimetracker.Tasks;


public class AmplifiedTest extends ActivityInstrumentationTestCase2<Tasks> {
   private Solo solo;

   ...
   public void testRun() {

      //EVENT 0: Wait for the main activity
      solo.waitForActivity(com.markuspage.android.atimetracker.Tasks.class, 2000);

      //Get GUI state before DOC
      GUIbefore=describeGUI();

      //DOUBLE ORIENTATION
      solo.setActivityOrientation(Solo.LANDSCAPE);
      solo.sleep(5000);
      solo.setActivityOrientation(Solo.PORTRAIT);
      solo.sleep(5000);

      //Get GUI state after DOC
      GUIafter=describeGUI();

      //CHECK ASSERTIONS
      Missing.add(disappearing(GUIbefore,GUIafter))
      Extra.add(appearing(GUIbefore,GUIafter))
      Wrong.add(changingState(GUIbefore,GUIafter))

      //EVENT 1: Click on "OK"
      solo.clickOnText("OK");

      //Get GUI state before DOC
      GUIbefore=describeGUI();

      //DOUBLE ORIENTATION
      solo.setActivityOrientation(Solo.LANDSCAPE);
      solo.sleep(5000);
      solo.setActivityOrientation(Solo.PORTRAIT);
      solo.sleep(5000);

      //Get GUI state after DOC
      GUIafter=describeGUI();

      //CHECK ASSERTIONS
      Missing.add(disappearing(GUIbefore,GUIafter))
      Extra.add(appearing(GUIbefore,GUIafter))
      Wrong.add(changingState(GUIbefore,GUIafter))

      //EVENT 2: Open the Action Overflow Menu
```

```
    solo.sendKey(Solo.MENU);
    //Get GUI state before DOC
    GUIbefore=describeGUI();

    //DOUBLE ORIENTATION
    solo.setActivityOrientation(Solo.LANDSCAPE);
    solo.sleep(5000);
      solo.setActivityOrientation(Solo.PORTRAIT);
    solo.sleep(5000);

    //Get GUI state after DOC
    GUIafter=describeGUI();

    //CHECK ASSERTIONS
    Missing.add(disappearing(GUIbefore,GUIafter))
    Extra.add(appearing(GUIbefore,GUIafter))
    Wrong.add(changingState(GUIbefore,GUIafter))
    ...
}
```

3.4.3 DOC GUI Failure Validation

This step was performed with the aim of obtaining a set of unique and validated DOC GUI failures.

I analyzed with the collaboration of a research fellow the collected GUI failures in order to remove the *duplicate* ones and obtain a list of unique DOC GUI failures. Two GUI failures are assumed as duplicate GUI failures when they are characterized by the same scope and mode, have equivalent start states and equivalent end states, and affect the same app. As an alternative, all the detected GUI failures could have been reported to developers, giving them the responsibility to decide whether or not two or more failures were actually duplicate GUI failures. However, the former way of counting has been preferred to not annoy the developers by flooding them with multiple similar (if not identical) requests.

Then, we assessed whether each failure was actually the manifestation of an incorrect GUI state rather than an intended behavior of the application. To this aim, I consulted the developer of the apps and opened an issue for each failure on the F-Droid issue tracker. Each issue contained a description of the observed DOC GUI failure and a sequence of events leading to it. The developers' answers received by the end of the study were analyzed in order to have the evidence about the issues that were accepted or rejected. The failures whose issues had been accepted by the developers were considered as validated.

Table 3.3 reports the data collected at the end of the validation activity. For each application are reported the number of the detected unique DOC GUI Failures, the number of accepted issues (giving the number of validated failures), the number of the issues that were not accepted by the developers, and the number of issues for which the developers did not provide any answer. As data show, a total of 439 unique GUI failures were detected in 59 open source applications, whereas 9 out of

68 applications did not show any DOC GUI failure. Altogether 298 issues over 439 were accepted by the developers. 61 issues were not accepted by the developers as failures, whereas 80 issues did not receive any answer and remained as pending. Among the 298 accepted failures, only 7 were already known to their developers.

The answers given by developers who refused the issues were analyzed. In many cases, the developers claimed that there was no way to avoid the reported issues, since they were due to the default behavior of the Android framework. A few developers' answers were ambiguous and did not clearly state whether the issue was accepted or not. In these borderline cases, the failures were not counted as accepted failures in order to avoid that the experimental results were biased by a personal and possibly misleading interpretations of the developers' answers.

As a result, the data reported in Table 3.3 can be considered as a lower bound.

3.4.4 DOC GUI Failure Classification

In this step, each validated GUI failure has been characterized on the basis of its mode and scope, thus obtaining different classes of GUI failures. For each class of DOC GUI failure, we evaluated how many times it occurred and the number of applications that exposed it. Table 3.4 reports the results of this classification.

As the Table shows, the study revealed 13 classes of Missing GUI failures that involved 13 different types of GUI objects. It found 19 classes of Wrong GUI failures, involving 19 different types of GUI objects, and just 3 classes of Extra GUI failures.

The Missing and Wrong classes of failures were the most frequent ones, with overall 192 and 101 occurrences of failures respectively, whereas the Extra failures occurred fewer times, i.e. only 5 times over 298.

Considering the types of GUI objects involved in failures, it can be observed that there are GUI object types more involved in failures than other ones, i.e., Dialog (146 occurrences), ListView (28 occurrences), ScrollView (21 occurrences), and TextView (14 occurrences).

As to the number of affected apps, some failure classes such as (Missing, Dialog), (Wrong, ListView), and (Wrong, ScrollView) affected more applications than others, since each of them recurred in more than 10 different apps. The (Missing, Dialog) failure type occurred 141 times over 298 in 34 applications. The (Wrong, ListView) failure appeared 27 times in 16 apps, whereas the (Wrong, ScrollView) involved 13 applications with 19 occurrences.

Details about the dataset and the detected GUI failures have been made publicly available[16]; each reported GUI failure is provided with a link to the issue opened in the app bug tracker.

[16] https://docs.google.com/spreadsheets/d/1k8IbndKH9K-9kmTGI9Wnc-FPlrOOEJ2dCRd8Uqu9JGk/edit?usp=sharing

TABLE 3.3: DOC GUI Failures found in open source apps

App	#DOC GUI Failures	#Accepted Issues	#Not Accepted	#Not Answered
A Time Tracker	10	10	0	0
A2DP Volume	10	10	0	0
AFWall+	8	8	0	0
agram	9	9	0	0
Alarm Klock	2	2	0	0
Amaze File Manager	18	0	18	0
AntennaPod	20	20	0	0
BankDroid	4	0	0	4
Bee Count	7	7	0	0
Berlin-Vegan	0	0	0	0
Blitzmail	0	0	0	0
Cache Cleaner	0	0	0	0
Calendar Notifications Plus	10	0	0	10
Chibe	2	0	0	2
Colorpicker	2	2	0	0
Currency	10	9	1	0
DNS66	1	1	0	0
DroidShows	0	0	0	0
Etar	18	3	0	15
ExprEval	1	1	0	0
File Manager	9	9	0	0
FOSDEM companion	5	4	1	0
Gallery	8	8	0	0
ImapNotes2	3	0	0	3
Iven News Reader	1	0	0	1
LeafPic	9	9	0	0
Legeappen	13	13	0	0
Loop Habit Tracker	8	8	0	0
Lyrically	3	1	0	2
Malp	5	4	1	0
Mather	1	1	0	0
Network Monitor	0	0	0	0
NewPipe	12	0	0	12
NewsBlur	16	15	1	0
Odyssey	9	3	6	0
OpenFood	0	0	0	0
ownCloud News	7	7	0	0
Padland	8	8	0	0
PassAndroid	12	0	0	12
Periodical	4	4	0	0
Pinpoi	5	5	0	0
Port Knocker	5	1	0	4
Prayer Times	13	13	0	0
Primary	19	19	0	0
ReGex	4	3	1	0
Ruler	4	4	0	0
Shorty	1	1	0	0
Sieben	5	0	5	0
Silectric	4	2	0	2
Simple Dilbert	6	2	4	0
Simple Solitaire	2	2	0	0
Slide	0	0	0	0
SpaRSS	7	0	0	7
StageFever	1	1	0	0
SteamGifts	7	7	0	0
Step and Height counter	9	9	0	0
Stringlate	7	7	0	0
SyncThing	7	7	0	0
Tap'n'Turn	0	0	0	0
Taskbar	17	17	0	0
Transdroid Torrent Search	12	0	12	0
Transistor	0	0	0	0
Unit Converter Ultimate	2	2	0	0
uNote	9	9	0	0
Weather	11	0	11	0
Who Has My Stuff	6	0	0	6
WifiAnalyzer	10	10	0	0
World Clock & Weather	1	1	0	0
Total	**439**	**298**	**61**	**80**

TABLE 3.4: Classification of the DOC GUI failures found in open source apps

Failure Mode	Failure Scope	# Occurrences	# Involved Apps
Missing	Dialog	141	34
	Context Menu	11	6
	Action Overflow Menu	8	5
	Search View	7	6
	Text View	5	4
	Contextual Action Bar	5	4
	Button	4	3
	Edit Text	4	3
	OptionsMenu	2	2
	Spinner	2	1
	Toolbar	1	1
	List View	1	1
	Checkbox	1	1
Wrong	ListView	27	16
	ScrollView	19	13
	TextView	9	7
	Spinner	6	4
	Edit Text	6	4
	Number Picker	5	3
	Recycler View	4	3
	User Defined Widget	4	3
	Web View	4	3
	Dialog	4	2
	ActionMenuItem View	4	2
	Image View	2	2
	Time Picker	1	1
	Checkbox	1	1
	Button	1	1
	Date Picker	1	1
	HorizontalScrollView	1	1
	Navigation View	1	1
	TabHost	1	1
Extra	Button	2	1
	Scroll View	2	1
	Dialog	1	1

3.4.5 Common Faults Identification

In the previous step, 3 classes of DOC GUI failures, i.e. (Missing, Dialog), (Wrong ListView), and (Wrong, ScrollView) occurred most times and affected more than 10 apps.

- **Missing Dialog:** This failure consists in the disappearance of a Dialog object after a DOC event. A Dialog is a small window that prompts the user to make a decision or enter additional information. It does not fill the screen and is normally used for modal events that require users to take an action before they can proceed;
- **Wrong ScrollView:** This failure consists in the loss of the current state of a ScrollView object. A ScrollView contains and shows a list of GUI objects. Users

can scroll the list and see the items contained in it;

- **Wrong ListView:** This failure consists in the loss of the current state of a ListView object. A ListView shows a list of UI objects that can be vertically scrolled by the user, allowing it to be larger than the physical display. A ListView is filled using an adapter that pulls content from a source such as an array or database query and converts each item result into a view that's placed into the list.

Since the study resulted in a significant sample of failures of these 3 types, they have been investigated in order to assess whether the occurrences of the same types of failure had similar causes.

The source code of the Android apps exposing these failures was analyzed to detect the faults causing them. In this analysis, particular attention was paid to the mechanisms used by Android to manage the orientation changes. When an orientation change occurs, Android destroys the running Activity and then restarts it. The Activity lifecycle callback method `onDestroy()` is called, followed by `onCreate()`. The restart behavior is designed to adapt the app to the new layout configuration, without loss of user data or without disrupting the user experience.

This Android-specific feature must be taken into account by developers who should use the APIs provided by Android and follow the guidelines that describe the correct usage of the Android framework components. Analogously, testers should verify that the application handles properly the orientation change events.

As an example, Android guidelines prescribe that the control of a dialog GUI object (deciding when to show, hide, dismiss it) should be done through the API, not with direct calls on the dialog instances. Any violation of such guidelines may result in inconsistencies in the app behavior.

This analysis allowed to identify 6 classes of common faults that could be considered as errors in the usage of Android programming features, rather than mistakes related to the logic of a specific application.

These faults occurred multiple times, even in different apps, were often localized in the same category of Android code components, had the same characteristics, and could be solved by similar code fixes.

Hence, it is possible to describe each class of fault using a template made of the following characteristics:

- **Id:** an identifier used to refer to the common fault;
- **Associated failure:** the failures that are found to be caused by a fault that can be traced back to the described common fault;
- **Location:** Information about the Android app component where the fault can be detected;
- **Background:** this section contains general background information that is useful or interesting to better understand the common fault;

- **Description:** general characteristics of the considered common fault;
- **A possible fix:** this section explains an alternative solution that implement the same features intended by the developer but prevent the failures listed in the associated failure section;
- **Example:** an excerpt of code from a real app that contains an instance of the considered fault and a possible code fix.

In the following, I report the descriptions of the six fault classes.

3.4.5.1 Show method called on Dialog or its Builder

Id: SDB;
Associated failure: Missing Dialog;
Location: an object calling the static `show` method of the `Dialog` or the `AlertDialog Builder` classes;
Background: Android provides a specific guideline to deal with Dialog objects[17]; it states that Dialogs should be managed by the `DialogFragment` class, which ensures a correct handling of lifecycle events, such as when the user presses the Back button or changes the orientation of the screen[18];
Description: The app code contains calls to the public methods offered by the dialog object or its builder to show a dialog. This will correctly pop up the dialog on the screen but the dialog will disappear when the activity is destroyed and recreated due to orientation changes;
A possible fix: The developers can implement a class that extends `DialogFragment` and create the desired dialog in its `onCreateDialog()` callback method. They create an instance of this class and call `show()` on that object. The dialog appears but it will not disappear when the activity is destroyed and recreated due to orientation changes;
Example:
An example of this fault can be found in the `MainActivity` class of the app Periodical. When the user clicks on the Restore option in the action overflow menu, a dialog pops up but it disappears on orientation changes.

Listing 3.2 shows the relevant code. I highlighted in red the call to the `show()` method of the `AlertDialog` builder instance.
The green highlighted code shows a possible and effective fix; the same dialog is constructed in the `DialogFragment.onCreateDialog()` callback method.

[17] https://developer.android.com/guide/topics/ui/dialogs.html
[18] https://developer.android.com/reference/android/app/DialogFragment.html

LISTING 3.2: Example of a SDB Fault and a fix in the app Periodical

```
private void doBackup(){
     ...
-        final AlertDialog.Builder builder = new AlertDialog.Builder(this);
-        // The Builder class is used for convenient dialog construction...
-        builder.show()
+        DialogFragment backupAlert = new doBackupDialogFragment();
+        backupAlert.show(getSupportFragmentManager(), "backup");
}
...
+ public class doBackupDialogFragment extends DialogFragment {
+        @Override
+        public Dialog onCreateDialog(Bundle savedInstanceState){
+           AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
+           // The Builder class is used for convenient dialog construction...
+           return builder.create(); } }
```

3.4.5.2 Fragment created Twice on Activity restart

Id: FTA;
Associated failure: Missing Dialog, Wrong ListView;
Location: `onCreate` callback method of a class that extends `PreferenceActivity` or `PreferenceFragment` classes;
Background: To provide settings for the app, Android recommends to use the Preference API. Instead of using `View` objects to build the user interface, settings are built using various subclasses of the `Preference` class declared in an XML file. To load the preferences, the developer should call the method `addPreferencesFromResource()` during the `onCreate()` callback of a `PreferenceActivity` or, preferably, a `PreferenceFragment`;
Description: In the official tutorial on how to instantiate a `PreferenceFragment` within an activity, there is a faulty code snippet that creates a new `PreferenceFragment` each time the host activity is created. It results in a loss of state of the settings screen when the device is rotated. Our study detected that this code snippet is widely used among Android developers;
A possible fix: One simple solution is to add a check that determines whether the settings screen has already been created;
Example:
Listing 3.3 shows an example of this fault found in the `SettingsActivity` class of the app StageFever. When the user clicks on the Font Size of Notes options in the app settings, a dialog pops up. But it disappears on orientation changes. I added and highlighted in green a simple control that prevents the fragment `PrefsFragment` from being recreated if it has not been created for the first time but its state has been restored from the `savedInstanceState` bundle.

LISTING 3.3: Example of a FTA Fault and a fix in the app StageFever

```
@Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        prefs = PreferenceManager.getDefaultSharedPreferences(this);
+       if (savedInstanceState == null){
        getFragmentManager().beginTransaction()
                .replace(android.R.id.content, new PrefsFragment())
                .commit();
+       }
    }
```

3.4.5.3 Missing Id in XML Layout

Id: MIXL;
Associated failure: Wrong ScrollView;
Location: `ScrollView` element in layout XML files;
Background: In order for the Android system to restore the state of the views contained in an activity, each view must have a unique ID, supplied by the `android:id` attribute. Developers often rely heavily on visual editors to build layouts for their apps. The visual Layout Editor offered by Android Studio[19], the official IDE for Android platform development, allows the developers to build layouts by dragging widgets into a visual design editor instead of manually writing the layout XML. However, a `ScrollView` added to a layout via visual editor will miss the id attribute.
Description: The presence of a `ScrollView` element in the XML file describing the activity layout with no id attribute set can cause the loss of the ScrollView state, e.g. scroll position, when the user rotates the device;
A possible fix: To set an id attribute for the `ScrollView` element in the XML file describing the activity layout;
Example:
The XML code in Listing 3.4 defines the presence of a ScrollView in the layout of the event details fragment. When the user scrolls down the text that describes an event and then changes the orientation, the scroll position goes back to the top losing the effect of the user interaction. The green highlighted code is the fix of the developer that solved the bug and closed our issue[20] adding an id to the scrollview.

LISTING 3.4: Example of a MIXL Fault and a fix in the app FOSDEM

```
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
            xmlns:app="http://schemas.android.com/apk/res-auto"
            xmlns:tools="http://schemas.android.com/tools"
+           android:id="@+id/scrollview"
            android:layout_width="match_parent"
            android:layout_height="match_parent">
```

[19] https://developer.android.com/studio/index.html
[20] https://github.com/cbeyls/fosdem-companion-android/commit/b2e50f8e4dea7739f776373f1c3669ce70c2deb5

3.4.5.4 Aged Target SDK Version

Id: ATSDKV
Associated failure: Wrong ScrollView;
Location: `android:targetSdkVersion` attribute of `uses-sdk` element in the XML manifest file, i.e. `AndroidManifest.xml`;
Background: Every Android app must have a manifest XML file that provides essential information about the app itself to the Android system. The element `uses-sdk` has the `android:targetSdkVersion` attribute that is used to designate the API Level that the application targets;
Description: In the manifest file, the element `uses-sdk` has the `android:targetSdkVersion` value lower than 19. In this case, the app loses the `ScrollView` position on orientation change caused by a limitation of the framework version lower than 19. These limitations that have been fixed in the later versions of the Android SDK;
A possible fix: To set the `android:targetSdkVersion` value to 19 or higher in the manifest;
Example:
Listing 3.5 shows an excerpt from the Currency app manifest file that targets the version 17 of the Sdk. The implementation of `onSaveInstanceState` method of the `ScrollView` class in API versions lower than 19 does not retain the ScrollView position on configuration changes. Setting an API level to 19 or higher, as shown in the green highlighted code, fixes the issue as the ScrollView position is saved and restored directly by the system.

LISTING 3.5: Example of a ATSDKV Fault and a fix in the app Currency

```
<manifest
    ...
    package="org.billthefarmer.currency"
    ...>

  <uses-sdk
      android:minSdkVersion="14"
-     android:targetSdkVersion="17"
+     android:targetSdkVersion="19"
 />
```

3.4.5.5 List Adapter Not Set in onCreate Method

Id: LANSCM;
Associated failure: Wrong ListView;
Location: a class extending the `Activity` class where the list adapter setter method is called in a lifecycle callback method different from `onCreate()`;

Background: The ListView adapter binds source data to its layout. The adapter setter should be called in the `onCreate()` callback method that is responsible for retrieving and restoring the state of the list every time the activity is started or resumed;
Description: The adapter setter is called in a lifecycle method different from `onCreate()` and the developer does not explicitly restore the state of the list. The list state will be lost on orientation change, e.g. the position of the scrollable list is not preserved;
A possible fix: To call the adapter setter inside the `onCreate()` method;
Example:
An example of this fault can be found in the `ListProjectActivity` class of the app BeeCount. When the user scrolls down the list of projects and then changes the orientation, the scroll position goes back to the top loosing the effect of the user interaction. As shown in Listing 3.6, the `showData` method that sets the list adapter is called by the overridden `onResume` method.
To fix this issue, I simply moved the call to `showData` in the `onResume` method.

LISTING 3.6: Example of a LANSCM Fault and a fix in the app BeeCount

```
@Override
  protected void onCreate(Bundle savedInstanceState){
    ...
+    showData();
  }
    ...
@Override
  protected void onResume()
  {
    super.onResume();
-    showData();
  }
    ...
private void showData()
  {
    projects = projectDataSource.getAllProjects(prefs);
    adapter = new ProjectListAdapter(this, R.layout.listview_project_row, projects);
    setListAdapter(adapter);
  }
```

3.4.5.6 List Filled Through Background Thread

Id: LFTBT;
Associated failure: Wrong ListView;
Location: class that extends the helper class `AsyncTask` and calls the list adapter setter method;

Background: To fetch large data in the main UI thread can cause poor UI responsiveness or even Application Not Responding (ANR) errors. Thus, developers should fetch the data in another thread[21];
Description: Extending the helper class `AsyncTask` allows the developer to perform data fetching operations on a background thread and publish results on the main UI thread, i.e. setting the ListView adapter, without having to manipulate threads and/or handlers[22]. Still the developer is responsible for managing both the background thread and the UI thread through various activity or fragment lifecycle events, such as onDestroy() and configurations changes;
A possible fix: To use a Loader class, such as an AsyncTaskLoader, to load data from a data source for display in an Activity or Fragment. Loaders persist and cache results across configuration changes to prevent duplicate queries[23];
Example:
Listing 3.7 shows an example of this fault detected in the `lcd10Activity` class of the app LegenAppen. Each time the activity is created a `getChapterTask` asynchronous task is instantiated and executed to fetch the data and fill the ListView. This results in a loss of the ListView state, e.g. scroll position, on orientation changes.

LISTING 3.7: Example of a LFTBT Fault and a fix in the app LegenApp

```
public class Icd10Activity extends AppCompatActivity
+       implements LoaderManager.LoaderCallbacks<List<Item»
{
    @Override
    protected void onCreate(Bundle savedInstanceState){
        super.onCreate(savedInstanceState);
        ...
        mListView = (ListView) findViewById(R.id.icd10_list);
        ...
-       // Get chapters
-       GetChaptersTask getChaptersTask = new GetChaptersTask();
-       getChaptersTask.execute();
+       // Prepare the loader.  Either re-connect with an existing one,
+       // or start a new one.
+       getLoaderManager().initLoader(0, null, this);
    }...
-       private class GetChaptersTask
-       extends AsyncTask<Void, Void, SimpleCursorAdapter>{
-       @Override
-       protected SimpleCursorAdapter doInBackground(Void...  voids){
-       //the background thread performs a query and returns the results...
-       return new SimpleCursorAdapter(
-       mContext, R.layout.activity_icd10_list_item,
-       mCursor, fromColumns, toViews, 0);
-       }
```

[21] https://developer.android.com/guide/components/processes-and-threads.html
[22] https://developer.android.com/reference/android/os/AsyncTask.html
[23] https://developer.android.com/guide/components/loaders.html

```
-       ...
-       @Override
-       protected void onPostExecute(final SimpleCursorAdapter adapter){
-       //This method is invoked on the UI thread
-       //after the background computation finishes.  It takes
-       //the result of the background computation as a parameter.
-       mListView.setAdapter(adapter);
-       }
...
+       public Loader<List<Item» onCreateLoader(int id, Bundle args) {
+       // This LoaderManager callback is called
+       // when a new Loader needs to be created.
+       }
+       public void onLoadFinished(Loader<List<Item» loader, List<Item> data) {
+       // This callback method is called when a previously
+       // created loader has finished its load.
+       }
+       public void onLoaderReset(Loader<Cursor> loader) {
+       //This callback is called when a previously created loader is being reset
+       //(when you call destroyLoader(int) or when the activity or fragment
+       // is destroyed, and thus making its data unavailable.
+       }
```

3.4.5.7 Common Faults Summary

Table 3.5 summarizes the six classes of common faults found in this study and reports the Fault Acronym and the Name of each fault class.

Table 3.6 reports the relation among the 3 specific types of failures considered in the fault study and the 6 inferred fault classes.

Each element of the table reports the number of apps where the failure type occurred due to a common fault over the total number of apps affected by that failure type. As the Table shows, the SDB fault is the one that occurred mostly, since it involved the largest number (23) of apps. MIXL is particularly relevant since it caused (Wrong, ScrollView) failures in 77% (10/13) of the apps affected by this failure.

As Table 3.6 shows, the DOC GUI Failures were not always caused by the same fault. Different faults caused the same failure type in different apps. Moreover, the FTA fault originated two different failures type in different apps.

TABLE 3.5: Classes of common faults

Fault Acronym	Fault Name
SDB	Show method called on Dialog or its Builder
FTA	Fragment created Twice on Activity restart
MIXL	Missing Id in XML Layout
ATSDKV	Aged Target SDK Version
LANSCM	List Adapter Not Set in onCreate Method
LFTBT	List Filled Through Background Thread

TABLE 3.6: Relationships between DOC GUI failures and common faults

	SDB	FTA	MIXL	ATSDKV	LANSCM	LFTBT
Missing Dialog	23/34	10/34				
Wrong ScrollView			10/13	3/13		
Wrong ListView		10/16			3/16	4/16

3.4.6 Study Conclusion

At the end of the exploratory study, several interesting results were obtained allowing to reach the three goals of the study.

As for the first goal G1 of the study, the experimental results show that GUI failures due to orientation changes of the device are widespread among real Android apps. In fact, more than the 86% of the analyzed app sample exposed at least one DOC GUI failure. This datum suggests that the likelihood for app users to encounter these failures is very high, at least when they use open source apps.

It cannot be excluded that other testing techniques, different from the one used in this study, could expose even more failures. However, the percentage of apps affected by this problem is high enough to confirm the relevance of this topic to the community of Android developers and testers.

Regarding the second goal G2, the classification of GUI failures on the basis of their mode and scope show that most of the failures belong to the category of Missing objects and Wrong objects, with 64% and 34% of apps affected by them, respectively.

As for the scope of the failures, there are 4 types of GUI object that occurred more than 10 times and involved the 70% of the failures.

Lastly, it emerges that 3 types of failure, i.e., (Missing, Dialog), (Wrong, ListView), and (Wrong, ScrollView), are more widespread among the apps than the other ones, since each one of them affects more than 10 applications. This result provides a subset of failures worth to be further investigated.

As for the goal G3, the results show that a subset of observed types of failures were due to the same classes of faults that occurred in several different applications. The deeper analysis of the faults suggests that these classes of faults can be considered specific of Android apps, rather than isolated programming errors.

3.5 Exploratory Study of Industrial-Strength Apps

The first exploratory study presented in Section 3.4 provides evidence about the widespread diffusion of DOC GUI failures of different types in real Android apps coming from the open source world. Since I do not want to limit the validity of the conclusions solely to the context of open source apps, which are usually less mature than the ones available through the official app market, I present a second study. In this further study, the analysis is extended to the context of industrial-strength

apps selected from the official Android app market, Google Play[24], with the aim of reaching the same two goals G1 and G2 of the previous study. The study does not have the G3 goal of the former study, since it was not possible to access the source code of the considered apps.

In this study, it has been followed an experimental procedure very similar to the one performed in the previous study. Here, we executed 4 steps: Objects selection, Apps testing, DOC GUI failures validation, and DOC GUI failures classification.

3.5.1 Objects Selection

The objects of the study are Android apps belonging to the official Google Play Store that satisfied the following inclusion criteria. The app had to:

1. have more than 50 M installs;
2. have an average rating above 4 stars;
3. allow orientation change.

We used these requirements in order to select the most popular apps allowing the orientation change of the screen and having the best quality perceived by the users. We randomly selected 10 apps which met the proposed inclusion criteria. Table 3.7 lists name, version, number of installs and average rating of each selected app. These data are related to the period in which the study was performed.

TABLE 3.7: Dataset 2

App	Version	Category	Installs	Rating
App Lock	2.22.1	Tools	100 M	4.34
Dropbox	27.1.2	Productivity	500 M	4.40
Duolingo	3.39.1	Education	50 M	4.69
Gmail	6.11.27.141872707	Communication	1000 M	4.32
Pinterest	6.5.0	Social	100 M	4.57
Spotify Music	7.0.0.1369	Music & Audio	100 M	4.53
Twitter	6.27.1	News & Magazines	500 M	4.23
Waze	4.17.0.0	Maps & Navigation	100 M	4.56
Whatsapp	2.16.396	Communication	1000 M	4.42
Youtube	11.47.57	Video Players & Editors	1000 M	4.18

3.5.2 Apps testing

In this step, the object apps are tested by exploiting the same amplification technique used in the former study for testing the open source apps. 5 master students not involved in the previous study have been selected to obtain the initial set of test cases. The students recorded through the Robotium Recorder tool a number of test cases able to cover the features provided by the 10 applications under test. The test cases collected by the students were amplified in order to fire a double orientation change

[24] https://play.google.com/store/apps

and then to check the presence of the three DOC GUI failure modes through appropriate assertions after each recorded event, according to the approach presented in Section 3.4. The amplified test cases were launched on a real LG G4 H815 device with Android 6.0. Finally, the GUI failures automatically detected by these test cases have been collected.

3.5.3 DOC GUI failures validation

I manually analyzed the DOC GUI failures found in the previous step to remove the duplicate ones. Then the unique failures were reported to the Android customer support team offered by each app provider. In this case, there was no direct contact with the app developers so I had to interpret and answer the emails auto-generated by the app providers in order to validate the reported failures. I obtained a final answer only from the Dropbox and Pinterest support teams; both of them accepted the reported issues as failures.

Therefore, for validating the remaining DOC GUI failures I along with my research group decided to refer to the GUI Consistency Design Principle stating that: *"in a GUI, the same action should always yield the same result"* [66]. According to this principle, we checked the app behavior exhibited after the double orientation change in different points of the app, to verify whether it was inconsistent across the different parts of its GUI. If a GUI exposed a potential failure after the DOC event, such as a missing dialog, and we did not find the same behavior on different parts of the app GUI, then we deduced that the observed failure was a true positive, since it was a violation of the consistency principle.

Table 3.8 shows the DOC GUI failures found in the analyzed apps. The data show that all the 10 apps exposed more than one failure. Overall 140 DOC GUI failures were found. Waze and Pinterest are the applications where most of the failures were found.

TABLE 3.8: DOC GUI Failures found in popular Google Play apps

App	#Detected DOC GUI Failures
App Lock	2
Dropbox	7
Duolingo	11
Gmail	4
Pinterest	31
Spotify Music	12
Twitter	11
Waze	45
WhatsApp	3
Youtube	14
Total	**140**

3.5.4 DOC GUI failures classification

In this step, the failures were classified in terms of their mode and scope according to the proposed classification framework. Table 3.9 reports for each app the number of occurrences of the failure types they exposed.

Table 3.10 shows how many times each type of DOC GUI failure occurred and the number of applications that exposed it. The results obtained by this study are very similar to the ones of the former study; the Missing and Wrong mode failures were the most common types even in the most popular apps with 76 and 63 occurrences, respectively. Analogously, as for the involved object types, Dialogs, ListViews, ScrollViews and TextViews were the most frequent ones.

Details about the second dataset and the detected GUI failures have been made publicly available[25]. This document reports for each app its analyzed version, its Google Play Category, the failure types it exposed along with their occurrences, and a sequence of events able to trigger a specific failure type.

3.5.5 Study Conclusion

This second study obtained results very similar to the ones achieved by the former study.

As to the first goal G1, the experimental results confirmed that GUI failures due to orientation change are very frequent even in Android apps that are distributed through the official Android app market. As a consequence, it is possible to conclude that this problem is widespread in the field of Android apps and impact also industrial-strength applications.

Regarding the goal G2, the study indicated that Missing and Wrong are the most common DOC GUI failure modes also for the most popular Android applications. Considering the scope of the detected GUI failures, there are some types of GUI objects that occurred more frequently than others; more precisely, Dialogs, ListViews, ScrollViews and TextViews are the most involved types of GUI objects even among the most popular apps of the official Android market.

3.6 Threats to Validity

This section discusses the threats that could affect the validity of the results obtained in the exploratory studies.

3.6.1 Construct Validity

This aspect of validity reflects to what extent the operational measures that are studied really represent what the researcher has in mind and what is investigated according to the research question [67].

[25] https://docs.google.com/spreadsheets/d/1xhOudp3FBJq4MTHeRK4LWeqwRA9s_ltuh6PwkaIi-ZA/edit?usp=sharing

TABLE 3.9: Classification of the DOC GUI failures found in popular Google Play apps

App	Failure Mode	Failure Scope	Occurrences
App Lock	Missing	Action Overflow Menu	2
Dropbox	Wrong	ListView	1
	Missing	Modal Bottom Sheet	4
	Missing	Action Overflow Menu	2
Duolingo	Wrong	List View	3
	Wrong	Radio Button	1
	Wrong	Spinner	1
	Missing	Dialog	5
	Missing	Popup Menu	1
Gmail	Wrong	ListView	1
	Missing	Action Overflow Menu	2
	Missing	Context Menu	1
Pinterest	Wrong	ListView	1
	Extra	Tooltip	1
	Wrong	Dialog	1
	Wrong	Edit Text	8
	Wrong	GridView	1
	Wrong	RecyclerView	7
	Wrong	Spinner	1
	Wrong	Switch	2
	Wrong	Text View	3
	Missing	Dialog	3
	Missing	Image View	1
	Missing	Text View	1
	Missing	Sliding Up Panel	1
Spotify	Missing	Context Menu	5
	Missing	Action Overflow Menu	1
	Wrong	ListView	3
	Wrong	View Pager	1
	Wrong	Spinner	1
	Wrong	Web View	1
Twitter	Wrong	Search Widget	2
	Missing	Text View	1
	Missing	Action Overflow Menu	5
	Missing	Context Menu	1
	Wrong	Spinner	1
	Missing	Side Drawer	1
Waze	Missing	Modal Bottom Sheet	1
	Wrong	ImageView	1
	Wrong	List View	4
	Wrong	Scroll View	7
	Wrong	Time Picker	1
	Wrong	Web View	3
	Missing	Dialog	25
	Missing	Modal Bottom Sheet	2
	Missing	Time Picker Dialog	1
Whatsapp	Wrong	ScrollView	1
	Wrong	ListView	1
	Wrong	Image View	1
Youtube	Missing	Popup Menu	6
	Missing	Dialog	4
	Wrong	List View	3
	Wrong	Spinner	1

TABLE 3.10: Number of occurences of different DOC GUI failure types

Failure Mode	Failure Scope	# Occurrences	# Involved Apps
Missing	Dialog	37	4
	Action Overflow Menu	12	5
	Context Menu	7	3
	Modal Bottom Sheet	7	3
	Popup Menu	7	2
	Text View	2	2
	Image View	1	1
	List View	1	1
	Sliding Up Panel	1	1
	Time Picker	1	1
Wrong	ListView	17	8
	ScrollView	8	2
	EditText	8	1
	Recycler View	7	1
	Spinner	5	5
	Web View	4	2
	Text View	3	1
	Image View	2	2
	Search Widget	2	1
	Switch	2	1
	Dialog	1	1
	Grid View	1	1
	Radio Button	1	1
	Time Picker	1	1
	View Pager	1	1
Extra	Tooltip	1	1

In the studies we performed, since we did not have access to the requirements of each app, there was the risk that the GUI failures we detected could be not actual failures, but the manifestation of apps' expected behavior. This may be a possible threat to the construct validity of the studies. We mitigated this threat by exploiting GUI failures validation procedures.

In the first study, we relied on the developers' feedback. We opened an issue for each potential GUI failure and considered it a failure only when the app developers accepted that issue. This procedure makes me confident that all the GUI failures reported in the first study were actually failures.

In the second study, we used the violation of the UI Consistency Design Principle for validating the detected GUI failures. Although this procedure cannot definitely assure that they were aberrant app behaviors, it gave me additional evidence to assume it.

3.6.2 Internal Validity

This aspect of validity assesses that there are no uncontrolled variables of the experiment that had an effect on the outcome [67]. Such threats typically do not affect exploratory studies like the ones reported in this Chapter [68].

However, it cannot be excluded con-causes besides DOC events that triggered the observed GUI failures, e.g. the execution platform or the timing between consecutive events. A controlled experiment involving different Android OS versions, types of device, and time intervals between events should be carried out to further investigate this aspect.

3.6.3 External Validity

This aspect of validity is concerned with to what extent it is possible to generalize the findings to other contexts [67].

A possible threat to the generalizability of the experimental results could be the representativeness of the sample of Android apps.

In the first Study, we mitigated this threat by randomly selecting 68 open-source apps that had the orientation change enabled, active developers, and a issue tracker.

We did not limit the analysis to the open-source world and confirmed and strengthened our findings by considering industrial-strength apps in the second Study. In this second study, we randomly selected 10 apps from the official Google app market that allowed the orientation change, had more than 50 M installs, and had an average rating above 4 stars.

I cannot claim that these results generalize beyond the inclusion criteria we applied. Moreover, I cannot exclude that specific characteristics of the analyzed apps (such as the types of GUI widget they rely on, or their category) may have influenced the experimental results.

To further extend the validity of the study, a controlled experiment involving a larger set of apps with selected characteristics should be performed.

3.7 Related Work

In this work, I explored GUI failures in Android apps triggered by the orientation change mobile-specific event and analyzed source code bugs that cause them. There are many works in the literature that address event-based testing and mobile fault classification. In this Section, some of the most related ones are discussed.

3.7.1 Event-based mobile testing

Since mobile apps are event-driven systems, their behavior can be verified through inputs consisting in specific event sequences as stated by Belli *et al.* [22].

Several event-based testing techniques have been proposed in the literature to test mobile apps. These techniques span from random testing [8, 27, 61], to symbolic-execution-based test case generation [69, 70], ripping based [71, 30, 72], pattern-based testing [73, 74], model-based testing [4, 75] and combinations of model-based and combinatorial testing [76, 77].

Unlike the presented work, the main goal of most of these techniques is to maximize the code coverage or to find crashes in the apps under test. Other works instead aim at assessing specific quality aspects of mobile applications [3], such as performance [10], accessibility [17], security [14, 15], responsiveness [16], and energy consumption [19].

3.7.2 Testing apps through mobile specific events

Some recent works address the problem of testing a mobile application by mobile-specific events.

The work of Zaeem *et al.* [42] is based on the intuition that different mobile apps and platforms share a set of features referred to as User-Interaction Features and that there is a "general, common sense of expectation of how the application should respond to a given feature". The authors propose a technique for testing and generating oracles focusing on a subset of features, i.e. Double rotation, Killing and Restarting, Pausing and Resuming, Back button. They present `QUANTUM`, a framework that automatically generates a test suite to test the user-interaction features of a given app leveraging application agnostic test oracles. QUANTUM requires a user-generated GUI model of the app under test as input and provides as output a JUnit-Robotium test suite that exercises interaction features. Their initial experimentation of `QUANTUM` exposed a total of 22 real failures in 6 open-source Android apps, including 12 GUI failures due to orientation change.

Adamsen *et al.* [43] aim to improve the quality of apps by testing them under adverse conditions. They propose a technique and a tool named `THOR` that amplifies existing test cases injecting "neutral" event sequences that should not affect the functionality of the app under test and the output of the original test case. They focus on event sequences that are usually neglected in traditional testing approaches, including double orientation change events. Moreover, they provide a classification of the failures and bugs their technique is able to find. Among the four categories proposed in their classification, there are failures related to the GUI, *i.e.*, Unexpected Screen and Element disappears, that are similar to the ones I dealt with. They performed an experiment involving 4 real Android apps and the results showed that `THOR` was able to detect 66 distinct problems, the majority of which are due to events that cause a transition of the activity through the states Pause-Stop-Destroy-Create such as orientation change. Most of the failures detected by THOR belong to GUI category.

The results achieved both by [42] and [43] gave me a hint about the relevance of the problem addressed in this work, since several failures discovered by their techniques were GUI-related and exposed by the orientation change. While their work focused only on failure detection and classification, we also investigated the faults that cause a relevant part of these failures.

3.7.3 Android-specific fault classification

Several works in the literature aimed at defining Android-specific fault classes.

One of the first attempts at classifying Android faults is due to Hu and Neamtiu [27] that proposed 8 bug types by analyzing the faults they found in 10 open source Android apps. Their fault classification is based on bug report analysis, whereas we abstract Android fault classes by analyzing the causes of GUI failures observed by testing 68 real apps. Moreover, their fault categories are described at an high level of abstraction and are not supplemented by code-related information. Instead, we provide a more structured description of each class of fault, made of 7 characteristics that also contains the Android app component where the fault may be detected and a possible code fix.

Shan, Azim and Neamtiu [45] focused on a specific fault class due to the incorrect handling of the data that should be preserved when an app is resumed or restarted. They named KR errors the failures caused by these faults. These authors proposed a technique for finding KR errors and performed an experiment where they found 49 KR errors in 37 real Android apps. Most of these errors manifested themselves on the GUI, similarly to the GUI failures we dealt with in this work. But, unlike this work that distinguishes among different types of GUI failures, their paper generically classifies them as KR errors.

Banerjee *et al.* [78, 79] focused on another type of problem in Android apps i.e. abnormal energy consumption, called energy hotspots. These authors proposed an automated test generation framework aimed at detecting energy hotspots. Like this work, they also explored the causes of these failures in the Android app code and defined four fault classes (Resource Leak, Wakelock Bug, Vacuous Background Services, Immortality Bug), each one corresponding to a different energy hotspot category. Similarly to this work, they propose a structured description of each defect type made of Affected components, Defect pattern, Patch suggestion, and a Real-world example.

Deng *et al.* [80, 62] also dealt with Android bugs but with the different aim of defining novel operators to mutate the source code of Android apps. Part of their operators are designed on the basis of unique technical features of the Android framework. Another part is based on common faults in real apps obtained by investigating bug reports and code change history logs on Github repositories. This work instead introduces 6 classes of faults discovered by testing real Android apps rather than by mining bug tracking repositories. The majority of the considered bugs were not already present in issue trackers since we focused on problems often overlooked by developers and testers. The work of these authors shares one piece of common ground with this work, since they designed a specific operator to force testing of orientation changes. We focused on this specific issue and provided a comprehensive analysis that spans from GUI failures to code faults.

3.8 Conclusions and Future Work

In this Chapter, I addressed the problem of GUI failures in Android mobile applications that are caused by the screen orientation changes. I proposed a classification of these failures based on 2 attributes named scope and mode.

In order to investigate the impact of these failures in the context of Android, we performed 2 exploratory studies that involved both open source apps and apps distributed by the official Android Google Play market. The studies exploited amplification based black-box testing techniques for analyzing 78 apps. The results showed that more than 88% of the analyzed apps were affected by these failures and thus highlighted that this problem is widespread in the context of Android mobile apps. This study is the first one to point out the relevance of this issue in mobile apps context.

Almost all the failures detected by this study were novel and not already present in issue trackers. We made available the set of collected GUI failures as open-source; it provides the largest currently available dataset of this kind of failures and may be exploited to evaluate and compare the effectiveness of different testing techniques and tools.

The study also showed that some failure modes were more frequent than others and some GUI object types were more frequently involved. This suggests that developers should be aware and more careful about specific features of the Android framework. The management of some GUI object types may be critical and error-prone in Android app code because of deficiencies in Android framework and its documentation. However, I cannot exclude that we found more failures of certain types because the GUIs of the considered apps mostly used these objects. A controlled study would be necessary in order to verify this hypothesis.

The study also highlighted 3 types of failures that were more common than others among mobile apps and provided a relevant sample of failure instances of these types. We analyzed the source code of the apps affected by these failures and discovered 6 classes of common faults that cause them. These classes abstract common errors that should be avoided by developers in order to improve the app quality and to ensure better user experience.

In future work, we plan to exploit these Android-specific fault classes to develop new mutation operators for testing of Android apps and to define fault localization techniques focused on source code bugs that may cause the observed failures.

This work addressed GUI failures in the context of Android. However, the preliminary investigation presented in Section 3.2 shows that the addressed problem affects other mobile platforms. Thus, we plan to extend this work by considering other mobile operating systems, such as iOS and Windows10.

This work targeted GUI failures triggered by the orientation change event. In the next Chapter, I consider other mobile-specific events which may cause GUI failures.

Chapter 4

An Automated Black-Box Testing Approach for Android Activities

In this Chapter I propose ALARic, a fully automated black-box event based testing technique that explores an app under test for detecting issues tied to the Android Activity lifecycle. ALARic has been implemented in a tool. An experiment involving 15 real Android apps showed the effectiveness of ALARic in finding GUI failures and crashes tied to the Activity lifecycle. Moreover, ALARic proved to be more effective in detecting crashes than Monkey, the state-of-the practice automated Android testing tool.

4.1 Introduction

As introduced in Section 2.1.3, an Activity is a fundamental Android app component which represents a single GUI that allows the user to interact with the app. The Android Framework defines a peculiar lifecycle for Activity instances to guarantee a smooth user experience.

The official Android Developer Guide stresses the relevance of the Activity lifecycle feature and warns the developers of the threats it introduces in several sections; therefore it provides recommendations and guidelines to help programmers in the correct handling of this feature.

Despite this, several works in the literature have pointed out that mobile apps, including industrial-strength ones, suffer from issues that can be attributed to Activity lifecycle mishandling [4, 42, 43, 6, 28, 44, 45, 7, 46, 47, 48, 49, 50]. Zein *et al.* [3] performed a systematic mapping study of mobile application testing techniques involving 79 papers and emphasized the need for specific testing techniques targeting Activity lifecycle conformance.

Some solutions have been presented in the literature to address this problem. A part of them proposed testing techniques that rely on existing testing artifacts [43, 81] or GUI models [42] to automatically generate test cases able to properly exercise the Activity lifecycle. Another work [45] uses static analysis to detect bugs that may cause a corrupt state when an app is paused, stopped, or killed. Their solution can

also automatically generate test cases to reproduce bugs but it needs to modify the app code in order to verify the statically detected issues.

Another group of approaches leverages AGETs to find issues tied to the Activity lifecycle. These dynamic techniques mostly focus on finding a specific type of failure, such as crashes [6, 7] or resource leaks [46]. Only one of them [28] addressed GUI failures but their authors only considered the issues tied to the orientation change event, potentially neglecting the ones tied to other events that exercise the Activity lifecycle.

As pointed out by the exploratory studies presented in Chapter 3, GUI failures tied to the Activity lifecycle represent a widespread category of issues in Android apps and there is the need to define effective testing techniques to detect them.

To overcome these limitations, I propose ALARic (Activity Lifecycle Android Ripper), a fully automated black-box event-based dynamic testing technique that I realized with the support of my research group.

ALARic is able to detect both GUI failures and app crashes related to the lifecycle of the Activities of an app by systematically testing each Activity GUI state encountered during the automated app GUI exploration. To this aim, it leverages mobile-specific events able to exercise the Activity lifecycle and specifically designed testing oracles. This solution does not require any prior knowledge of the app under test, app modification or manual intervention.

The effectiveness of ALARic in detecting issues tied to the Activity lifecycle was demonstrated in an experiment involving 15 real Android apps. The experiment also showed that ALARic was more effective in detecting crashes tied to the Activity lifecycle than the state-of-the practice automated Android testing tool, *i.e.* Monkey, the most widely used tool of this category in industrial settings.

This work improves the literature on automated GUI testing with the following contributions:

- a novel automated GUI testing technique to detect GUI failures and crashes tied to the Android Activity lifecycle;
- an experiment involving real Android apps showing the validity of the proposed technique.

The remainder of the Chapter is structured as follows. Section 4.2 describes the background and Section 4.3 provides an overview of the proposed testing approach. Section 4.4 presents design and implementation details about the tool while Section 4.5 reports the experimental evaluation. Section 4.6 provides related work. Finally, Section 4.7 draws the conclusions and presents future work.

4.2 Background

4.2.1 Activity Lifecycle Loops

According to the Android Developer Guide, the Android Activity lifecycle presents 3 key loops that are named *Entire Loop*, *Visible Loop*, and *Foreground Loop*, respectively.

In the following, I briefly describe these loops. Moreover, for each key loop I provide a descriptive diagram in which the edges represent the callback methods that are invoked by the Android platform when an Activity transits between states during the execution of the loop:

- The *Entire Loop (EL)*, shown in Figure 4.1, consists in the Resumed-Paused-Stopped-Destroyed-Created-Started-Resumed sequence of states. There are different ways of exercising this loop that produce different behaviors of the Activity. It can be exercised by events that cause a configuration change, e.g. an orientation change of the screen, that destroys the Activity instance and then recreates it according to the new configuration[26]; in this case, the system is expected to retain the Activity instance state. Instead, if an Activity instance is in the foreground, and the user taps the Back button and then he returns to the Activity, this loop is still exercised but the Activity instance is destroyed and also removed from the back stack since there is no expectation of returning to the same instance of the Activity;

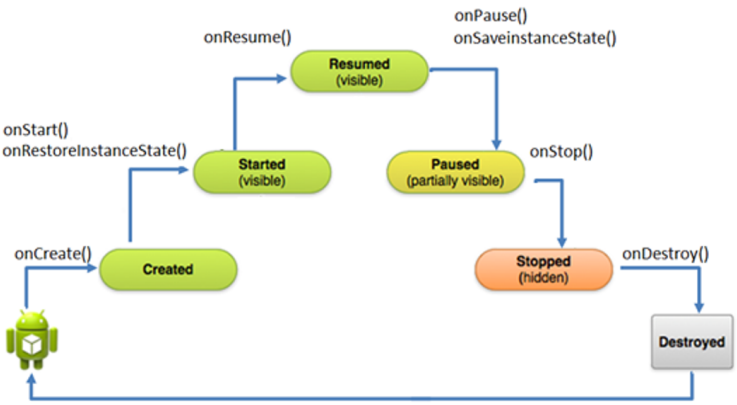


FIGURE 4.1: The Entire Loop

- The *Visible Loop (VL)*, shown in Figure 4.2, corresponds to the Resumed-Paused-Stopped-Started-Resumed sequence of states during which the Activity is hidden and then made visible again. There are several event sequences able to stop and restart an Activity, e.g. turning off and on the screen or putting the app in background and then in foreground again through the Overview or Home buttons;

[26] https://developer.android.com/guide/components/activities/state-changes.html

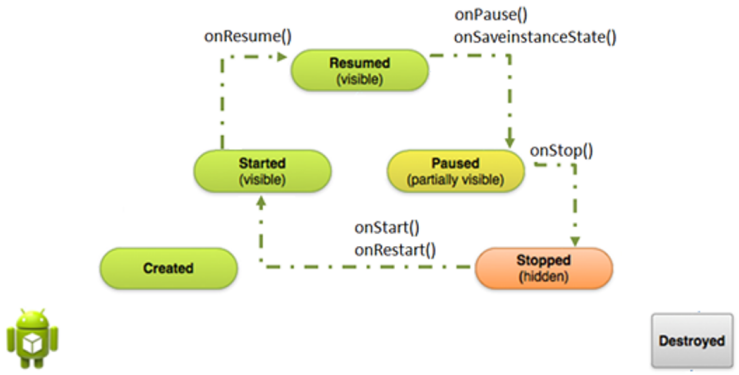


FIGURE 4.2: The Visible Loop

- The *Foreground Loop (FL)*, shown in Figure 4.3, involves the Resumed-Paused-Resumed state sequence. The transition Resumed-Paused can be triggered by opening non full-sized elements such as modal dialogs or semi-transparent activities that occupy the foreground while the Activity is still visible in background. To trigger the transition Paused-Resumed the user should discard this element. Therefore, this loop can be exercised by the event sequence that consists in opening and closing a non full-sized element.

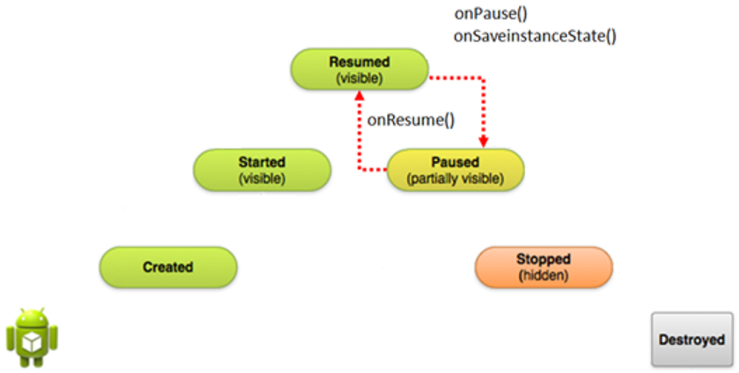
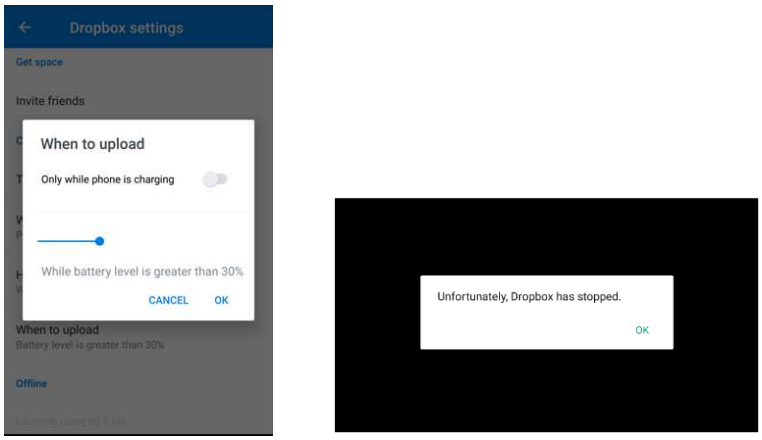


FIGURE 4.3: The Foreground Loop

4.2.2 Issues Tied to the Activity Lifecycle

Android App developers should correctly implement Activities, taking into account their lifecycle. This ensures the app works the way users expect and does not exhibit aberrant behaviors as it transitions through different lifecycle states at runtime. Good implementation of the lifecycle callbacks and the awareness of the Android Framework features can help the programmer to develop apps that behave as expected and prevent a number of issues. In this work, I will focus on Crashes and GUI failures.



(a) GUI state before exercising the Entire Loop

(b) GUI state after exercising the Entire Loop

FIGURE 4.4: Crash exposed by the Dropbox app

4.2.2.1 Crashes

A crash occurs when an app stops functioning properly and exits unexpectedly. When an app crashes, Android terminates its process and displays a dialog to let the user know that the app has stopped[27]. It is a very undesirable app behavior and, indeed, the most blatant one. Therefore, a number of testing tools has been proposed in the literature to expose Android crashes [2, 71, 7, 9]. This work will focus on crashes tied to the Activity lifecycle, i.e. crashes triggered by events that exercise the Activity lifecycle. I report an example of crash that we detected in Dropbox version 27.1.2[28], the Android client app offered by the popular file hosting service. If the user selects the third item of the *Camera Uploads* settings, a modal dialog appears (see Figure 4.4(a)). When the user changes the orientation of the device, the app suddenly stops working, as shown in Figure 4.4(b).

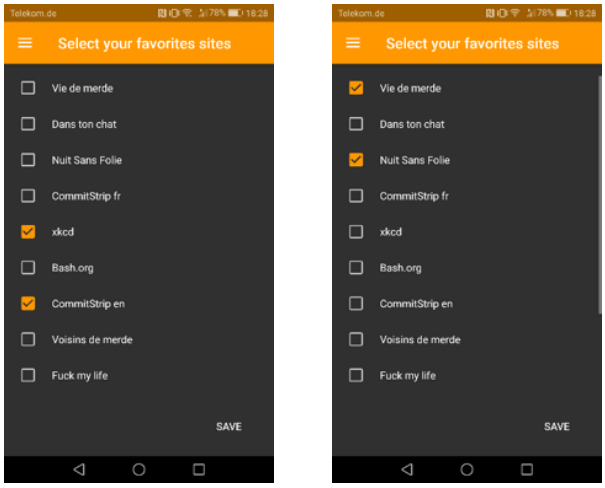
4.2.2.2 GUI Failures

GUI failures are a relevant class of failures that may disrupt the user experience and consist in the manifestation of an unexpected GUI state [52]. In particular, there may be a GUI failure tied with the Activity lifecycle when the GUI state before the Activity is stopped, paused or destroyed is different from the GUI state displayed after the user returns to the Activity [42, 43, 45, 82, 53].

The studies I presented in Chapter 3 pointed out that GUI failures triggered by exercising the Entire Loop, i.e. changing the orientation of the screen, are a widespread problem that affect both open source and industrial-strength apps. The work presented in this Chapter will focus instead on GUI failures triggered by exercising all the 3 Activity lifecycle key loops.

[27] https://developer.android.com/topic/performance/vitals/crash.html
[28] https://dropbox.zendesk.com, Dropbox Support request ID #5199918

As an example, I report a GUI failure triggered by exercising the Visible Loop that we found in version 1.1.2 of Anecdote, an Android application that displays quotes and that is available for free on the Google Play Store. The users can select from a list their favorite websites by checking the corresponding checkboxes (see Figure 4.5(a)). When the users put the app in background by tapping the Home button and then push it back in foreground, they naturally expect that the GUI state will remain the same. Instead, the app will exhibit an unexpected GUI state, i.e. the previously checked boxes are unchecked and different websites appear selected as shown in Figure 4.5(b).



(a) GUI state before exercising the Visible Loop

(b) GUI state after exercising the Visible Loop

FIGURE 4.5: GUI Failure exposed by the Anecdote app

4.2.3 Lifecycle Event Sequences

Android apps are EDS that can react to several types of events. Some specific event sequences are able to cause transitions between Activity states.

In particular, I will refer to sequences of events able to trigger one of the key loops of the Activity lifecycle as *Lifecycle Event Sequences*. Table 4.1 reports the Lifecycle Event Sequences we elicited from the Android Developer Guide, their description and the key loops they are able to trigger.

4.3 The ALARic Approach

In this section, I present the approach I proposed in collaboration with my research group to test the Activity lifecycle of Android apps, named ALARic.

ALARic implements a fully automated online testing technique since it explores the application under test (AUT) and at the same time detects aberrant behaviors tied to the Activity lifecycle, i.e. Crashes and GUI failures.

TABLE 4.1: Lifecycle Event Sequences

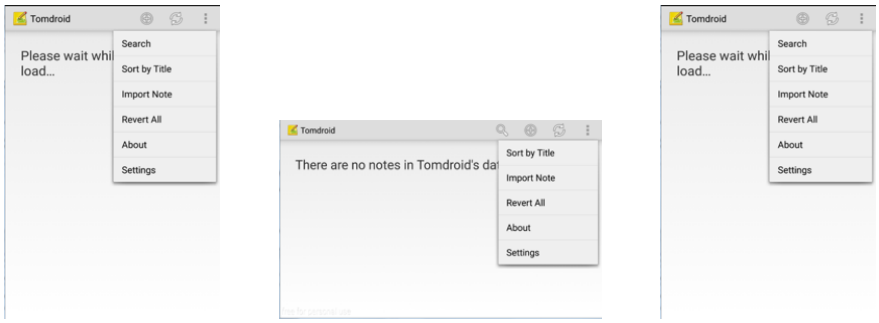
		Triggered Activity Lifecycle Loops		
Lifecycle Event Sequence	**Description**	**EL**	**VL**	**FL**
Orientation Change (OC)	Change the orientation of the device	✓		
Back and Reopen Activity (BRA)	The back button is pressed then the Activity is started again	✓		
Turn Off screen Turn On screen (TOTO)	The Screen is turned off and then it is turned on by pressing the power button		✓	
Background Foreground (BF)	The app is brought into the background opening the Task manager and then is brought back into foreground selecting it from the active tasks.		✓	
Receive phone call Close phone call (RC)	A phone call is received and then is hanged up		✓	
Semi-Transparent Activity Intent (STAI)	A Semi-transparent Activity is invoked by an intent and then it is closed pressing the back button			✓
Dialog Open Dialog Close (DODC)	A Dialog is shown and then it is closed by pressing the back button			✓

ALARic exercises the AUT through input event sequences, being Android apps event-driven software systems [22]. The exploration strategy adopted by ALARic sends random input events to the AUT and systematically executes an input event sequence able to exercise one of the 3 key Activity lifecycle loops each time a new GUI is encountered for the first time during the app exploration. After the Activity lifecycle loop is exercised, ALARic evaluates whether the app exposes any issue related to the Activity lifecycle. The absence of issues tied to the Activity lifecycle after the Activity Lifecycle is exercised is a necessary property of the target Android apps that can be seen as a kind of metamorphic relation [83]. This relation is exploited by ALARic to verify the correctness of the apps under test even when their requirements are not available.

To exercise the 3 key lifecycle loops, we leverage 3 Lifecycle Event Sequences, i.e., the *Double Orientation Change (DOC)*, the *Background Foreground (BF)* and the *Semi-Transparent Activity Intent (STAI)* event sequences. We chose these Lifecycle Event Sequences for 2 main reasons. The former reason is that each of these event sequences is able to exercise a different lifecycle loop. The latter reason is that the GUI state of the Activity should be retained after the execution of these event sequences [42] and, therefore, they are suitable for detecting GUI failures.

The *Double Orientation Change (DOC)* event sequence exercises twice the *EL loop* and consists in a sequence of 2 consecutive orientation change events. As in the studies presented in Chapter 3, we used the DOC event sequence to detect GUI failures. Figure 4.6 reports an example that explains why applying a single orientation change may not be sufficient to detect GUI failures. It shows a DOC event sequence triggered on an Activity of the Tomdroid app starting from the GUI state in Figure 4.6(a). After a single orientation change event, the Activity is rendered in landscape mode (see Figure 4.6(b)) and differs from the starting state since the Action Overflow Menu does not have the Search option anymore but presents a magnifier icon. These minor differences in GUI content are indeed acceptable because the app provides the

same functionality in both landscape and portrait orientations. After a second consecutive orientation change (see Figure 4.6(c)), the GUI content and layout is the same as before the first orientation change.



(a) (b) (c)

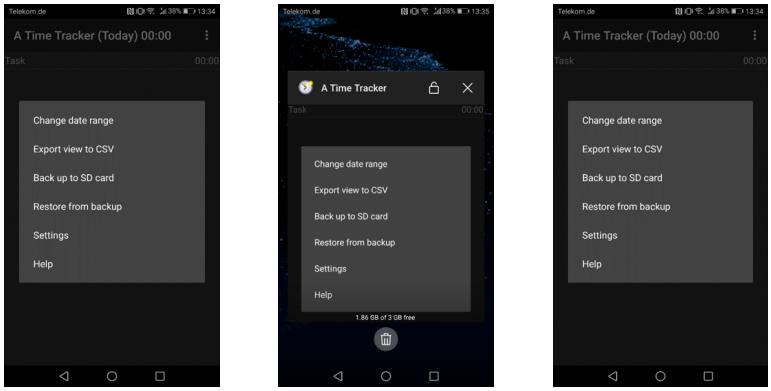
FIGURE 4.6: The Double Orientation Change Lifecycle Event Sequence

The *Background Foreground (BF)* sequence puts the app in background through the tap of the Home button and then pushes the app again in foreground. It exercises the *VL loop*. Figure 4.7 shows a BF event sequence triggered on an Activity of the A Time Tracker app.

As regards the *FL loop*, it is exercised by the *Semi-Transparent Activity Intent (STAI)* event sequence. It consists in starting a semi-transparent Activity that pauses the current foreground Activity and then returning to it by tapping the Back Button. Figure 4.8 shows a STAI event sequence triggered on an Activity of the Amaze app.

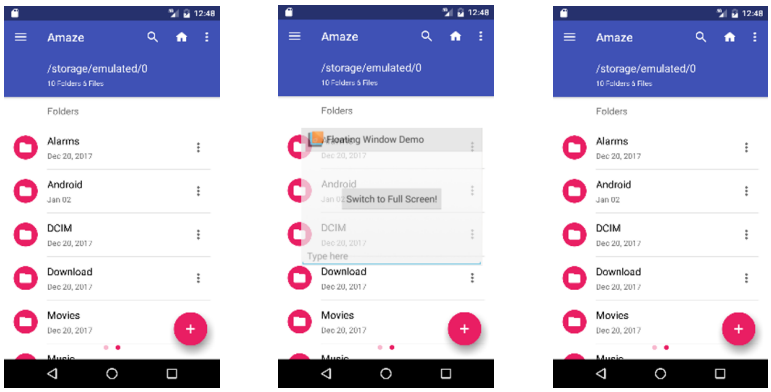
The ALARic approach is configurable and allows the tester to set up one type of Lifecycle Event Sequence to apply in order to exercise the Activity lifecycle in all the GUI states exposed by an Activity that are encountered during the app exploration.

Figure 4.9 shows a real example of how ALARic works. In this example, I use the DOC Lifecycle Event Sequence to test the Amaze app version 3.1.2 RC4. The snapshots represent the GUI states encountered during the automated GUI exploration.



(a) (b) (c)

FIGURE 4.7: The Background Foreground Lifecycle Event Sequence



(a) (b) (c)

FIGURE 4.8: The Semi-Transparent Activity Intent Lifecycle Event Sequence

The red edges represent Lifecycle Event Sequences, whereas the black edges are random planned events. At each exploration iteration, ALARic describes the current GUI state and verifies whether it has been explored before during the exploration. The GUI states encountered for the first time, i.e. A, C, E, H, L, are exercised by a DOC. Whereas, the GUI states already encountered, i.e. D, F, G, J, K, are exercised by random planned events. The tool compares the GUI states before and after the DOC event and checks whether they are equivalent. ALARic found 3 GUI failures in this exploration, i.e. after the 3th, 5th and 8th iteration. Moreover, the app crashed after the triggering of the 12th event. When a crash occurs, ALARic starts the app from scratch. The exploration terminates either after the triggering of a predefined number of events or after a given testing time.

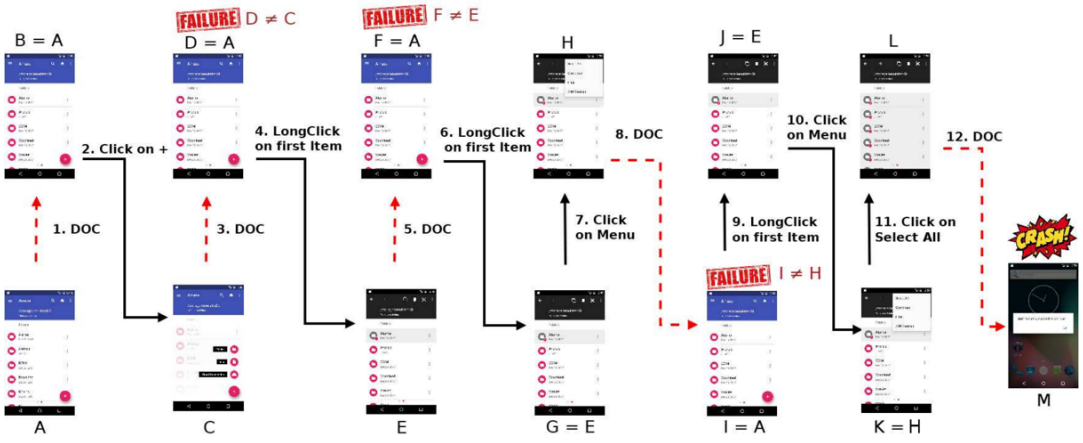


FIGURE 4.9: ALARic testing example

4.4 The ALARic Tool

The ALARic approach has been implemented in the *ALARic Tool*[29]. Fig. 4.10 shows the tool architecture that consists of 2 components: the *ALARic Engine* and the *Test Executor*.

The *ALARic Engine* component is responsible for implementing the business logic of the testing approach. It analyzes the GUI currently rendered by the AUT, plans the next input event sequence to be fired and checks the presence of failures. It does not interact directly with the AUT but delegates this task to the *Test Executor* component that is able to fire input event sequences on the AUT and to fetch the description of the current GUI in terms of its composing widgets and their attribute values.

The tool takes as input a *Configuration File* that is needed to set up the testing process. In this file, the user specifies the Lifecycle Event Sequence to be triggered and the termination condition. As for the termination condition, it is possible to set either a maximum execution time or the number of input event sequence to be fired. The *ALARic Engine* fetches the AUT by exploiting its *.apk* or its source code and then *installs* it on the *Test Executor*.

During the automatic app exploration, ALARic is able to save the descriptions of the encountered GUIs. At each exploration iteration, it compares the GUI currently rendered on the screen against the descriptions of the GUI states already encountered during the exploration. When the current GUI state does not match any stored GUI description, ALARic saves it and test it by applying the specified Lifecycle Event Sequence and evaluating the test oracle. In this way, the proposed tool tests only the GUI states encountered for the first time.

The tool produces a *Report File* about the detected crashes and GUI Failures. The report contains for each GUI failure: (1) the app name, (2) the Activity name where the failure was detected, (3) the sequence of events that led to the failure and (4) the executed Lifecycle Event Sequence type. Moreover, for the GUI Failures, it also contains the description and the screenshot of the GUI states before and after the application of the Lifecycle Event Sequence. As for the crashes, it contains the unhandled exception type and its stack trace.

4.4.1 ALARic Engine

The online testing process implemented by the *ALARic tool* is described by the UML Activity diagram shown in the *ALARic Engine* component of Figure 4.10. It extends the generic online testing algorithm presented in the framework proposed by Amalfitano et al. [25]. The steps belonging to the original algorithm are reported in white, whereas the ones introduced by the ALARic approach are colored in gray.

The *ALARic Engine* performs an iterative process of automatic GUI exploration where sequential steps are executed until a given termination condition is reached.

[29]The ALARic tool is available for download at the following link `https://goo.gl/ypTMVs`.

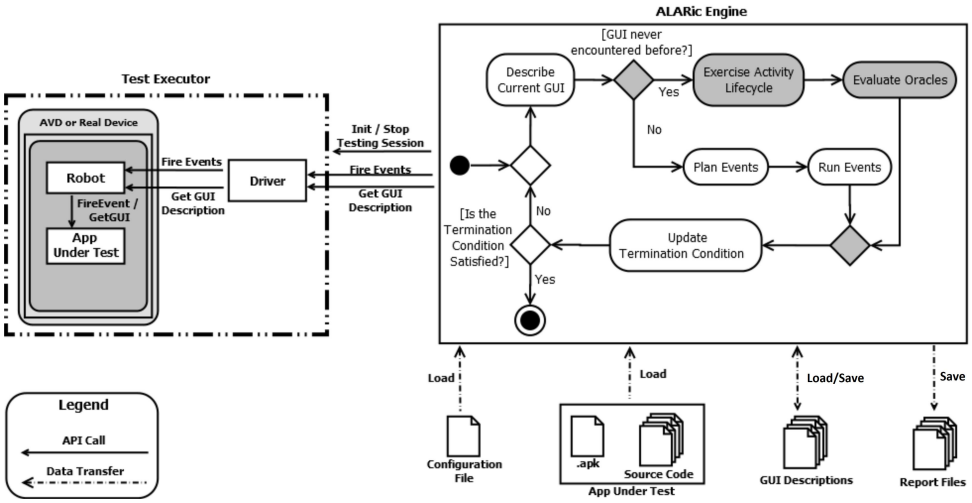


FIGURE 4.10: The ALARic tool architecture

In the *Describe Current GUI* step, a description of the current GUI state is inferred, according to a GUI description abstraction criterion. The description of a GUI state includes the $(attribute, value)$ pairs assumed by its components at runtime. The description of the current GUI state is compared with the ones previously encountered to evaluate whether it has ever been met before during the exploration. The *Exercise Activity Lifecycle* and *Evaluate Oracles* steps are executed when a GUI state is encountered for the first time. Otherwise, the *Plan Events* and *Run Events* steps are executed.

In the *Exercise Activity Lifecycle* step, one predefined Lifecycle Event Sequence is triggered. The *Evaluate Oracles* step allows the verification of oracles appositely crafted to detect the presence of specific types of Activity lifecycle issues. The current ALARic implementation is able to evaluate 2 oracles, i.e. GUI failures and crashes. As for GUI failures, similarly to [42, 43, 45, 82, 53], the tool is able to recognize failures that occur when the GUI states before and after the application of a Lifecycle Event Sequence are not equivalent. As for the crashes, ALARic checks whether an unhandled exception occurs after the execution of a Lifecycle Event Sequence.

The *Plan Events* step plans the input event sequences that will be fired on the current GUI according to a uniform random scheduling strategy. In the *Run Events* step, the planned input event sequence is actually executed.

Finally, in the *Update Termination Condition* step, it is evaluated whether a maximum testing time or a maximum number of fired input events, defined by the tester, is reached.

The *Engine* requires the *REST APIs* provided by the *Test Executor* component to carry out its activities. It calls the *Init Testing Session* and *Stop Testing Session* APIs at the begin and at the end of the testing process for installing and uninstalling the application under test, respectively. The *Fire Events* API is used for triggering events, whereas the *Get GUI Description* one is exploited for retrieving the description of the current GUI.

4.4.2 Test Executor

The *Test Executor* component is in charge of executing the testing activities of the *ALARic Engine* on the AUT. It is able to interact with both a physical device and a Android Virtual Device (AVD)[30]. It is made of 2 components, i.e. *Robot* and *Driver* that interact through Java socket technology.

The *Driver* component is in charge to decouple the business logic implemented in the ALARic Engine from the device where the AUT is installed. The *Robot* should run on the same device where the AUT is installed and interacts with it by firing events and describing the GUIs rendered at runtime. This component exploits the APIs provided by the Robotium library[31] and the Android Debug Bridge (ADB)[32].

4.5 Experimental Evaluation

In this section, I report the study I carried out with the collaboration of my research group to investigate the ability of the ALARic tool in detecting issues tied to the Activity lifecycle. We consider as tied to the the Activity lifecycle the issues that are exposed by *Lifecycle Event Sequences*. The study aimed at answering the following research questions:

RQ$_1$ How effective is the ALARic tool in detecting issues tied to the Activity lifecycle in real Android apps?

RQ$_2$ How does the effectiveness of the ALARic tool in detecting issues tied to the Activity lifecycle in real Android apps compare to the state-of-the-practice tool Monkey?

Some tools have been proposed in literature that exploit dynamic analysis and can find crashes tied to the Activity lifecycle [6, 7]. However, we were unable to compare the ALARic tool against them, since they are either no longer available, or are not supported anymore and are unable to target the latest Android OS and SDK versions. Therefore, we considered the Monkey[33] tool since it is regarded as the current state-of-practice for automated Android app testing [84, 9], being the most widely used tool of this category in industrial settings [39, 85].

4.5.1 Objects

As objects of the evaluation, we selected 15 apps that were distributed by the official Google app store whose source code was available in the F-Droid repository. In this way, we selected apps that were representative of the typical apps available to Android users. The availability of the source code allowed us to better analyze the

[30] https://developer.android.com/studio/run/emulator.html
[31] https://github.com/RobotiumTech/robotium
[32] https://developer.android.com/studio/command-line/adb.html
[33] https://developer.android.com/studio/test/monkey.html

detected failures. We chose F-Droid since it is a well-known repository of Free and Open Source Software (FOSS) applications for the Android platform that has been widely used in other studies on Android testing proposed in literature [2, 7, 9, 82, 84]. Table 4.2 reports for each selected app its name, the considered version, its size and the number of its Activities.

TABLE 4.2: Object Apps

	App Name	Version	Apk Size (kB)	#Activities
A1	A Time Tracker	0.21	115	5
A2	Port Knocker	1.0.9	2,200	6
A3	Who Has My Stuff?	1.0.27	104,3	4
A4	Agram	1.4.1	723	5
A5	Alarm Klock	1.9	640	5
A6	Padland	1.3	2,000	10
A7	Syncthing	0.9.1	19,300	12
A8	Anecdote	1.1.2	1,800	3
A9	Amaze File Manager	3.1.2 RC4	5,900	5
A10	Google Authenticator	2.21	708	5
A11	BeeCount	2.3.9	3,200	8
A12	FOSDEM companion	1.4.6	1,300	8
A13	Periodical	0.30	925	6
A14	Taskbar	3.0.2	1,600	23
A15	SpaRSS	1.11.8	1,400	8

4.5.2 Metrics

To evaluate the effectiveness of ALARic in detecting GUI failures, we considered the following metrics:

- $\#DGF_{DOC}$: number of distinct GUI Failures triggered by the DOC event sequence;
- $\#DGF_{BF}$: number of distinct GUI Failures triggered by the BF event sequence;
- $\#DGF_{STAI}$: number of distinct GUI Failures triggered by the STAI event sequence;
- $\#DGF_{Total}$: number of distinct GUI Failures tied to the Activity lifecycle triggered by either DOC, BF, or STAI.

Analogously, to evaluate the effectiveness of the tools in finding crashes, we considered the following metrics:

- $\#DC_{DOC}$: number of distinct Crashes triggered by the DOC event sequence;
- $\#DC_{BF}$: number of distinct Crashes triggered by the BF event sequence;
- $\#DC_{STAI}$: number of distinct Crashes triggered by the STAI event sequence;

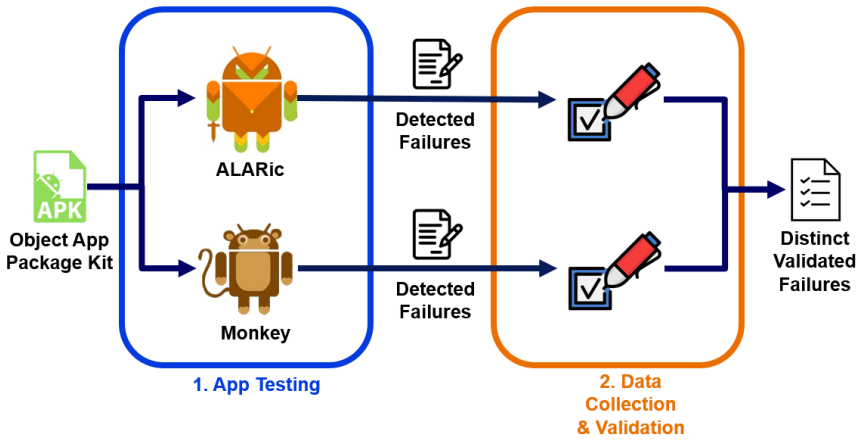


FIGURE 4.11: Overview of the Experimental Procedure

- $\#DC_{Total}$: number of distinct Crashes tied to the Activity lifecycle triggered by either DOC, BF, or STAI.

Since the same issue may be exposed multiple times during a testing process, we decided to count only the occurrences of distinct issues. We made these assumptions:

- GUI failures are distinct if they involved not equivalent start states and not equivalent end states given the notion of equivalence between GUI states introduced in Section 3.3.3;
- crashes are distinct if they do not refer to the same type of unhandled exception and do not have the same stack trace [6, 7].

4.5.3 Experimental Procedure

The experimental procedure we followed is shown in Figure 4.11. It was organized in two steps, namely (1) App Testing and (2) Data Collection & Validation.

4.5.3.1 App Testing

This step was conducted in two phases. In the former phase, all the objects were tested using the ALARic tool. We used 3 configurations of the tool. Each configuration allows ALARic to plan and execute only one of the Lifecycle Event Sequence types, *i.e.* DOC, BF, and STAI. We performed 3 runs for each configuration to mitigate the non determinism of the apps and of the random exploration techniques [2]. Each run lasted 1 hour. A total of 9 one-hour testing runs for each app were performed.

In the latter phase, we tested the object apps using the Monkey tool, in order to compare the tools effectiveness. Monkey is an automated testing tool for Android apps, belonging to the Android SDK. This tool adopts a random testing approach,

which sends a random stream of UI and system-level events to the app under test. We performed a testing process where 9 one-hour Monkey testing runs were executed for each app. We executed 9 Monkey runs as with ALARic in order to ensure a fair comparison among the tools. In this phase, we set the maximum verbosity level of the Monkey tool in order to produce a more accurate and rich output containing information about the seeded events and the detected crashes.

All the testing processes have been performed on the same testing infrastructure which consists of a desktop PC having an Intel(R) Core(TM) i7 4790@3.60GHz processor and 8 GB of RAM, running a standard Nexus 5 AVD with Android 6 (API 23). The host PC was equipped with the Ubuntu OS, version 16.04. All the runs were executed on AVDs created from scratch to assure that each run was executed in the same conditions.

4.5.3.2 Data Collection & Validation

At the end of the testing processes, all the reports produced by the considered tools were gathered. A team composed by me, another Ph.D. students and a Postdoctoral Researcher having knowledge on Software Debugging and Android Testing was asked to analyze the failures exposed by the tools and to validate them. To this aim, the team examined the reports produced by the considered tools to identify the distinct failures exposed by each tool. Unlike ALARic, Monkey does not offer a detailed description of each failure exposed at runtime [7]. To extract the crashes detected by Monkey from the output it generated, the team manually inspected these files to find the exception stack traces instances and the events that led to them. Then they exploited the information contained in the issue reports to reproduce all the distinct failures. To this aim they manually tried to reproduce the reported issues on a real LG G4 H815 device equipped with Android 6.0. In this way we are able to consider only real failures caused by an incorrect application logic and discard the ones caused by issues tied to the testing infrastructure and the virtual device. As regards the GUI failures, the team also assessed whether each failure was actually the manifestation of an incorrect GUI state rather than an intended behavior of the application, *e.g.* a timer that continues to count down or a news feed that adds new elements may cause the GUI state to be different after the execution of a Lifecycle Event Sequence. To guarantee that the issues were actually tied to the Activity lifecycle, the team performed a debugging activity to verify that the issues were a manifestation of faults that are exercised by executing the Activity lifecycle.

4.5.4 Results and Analysis

The validated distinct failures have been used to calculate the values of the metrics. Table 4.3 reports, for each app, the total number of GUI failures and crashes that have been found by ALARic and validated by the team, grouped by the Lifecycle Event Sequence type that triggered them. Table 4.4 shows for each app the number

of total crashes tied to Lifecycle Event Sequences detected by ALARic and Monkey, respectively. We did not compare the results regarding GUI failures since Monkey is not able to detect them.

TABLE 4.3: Experimental Results

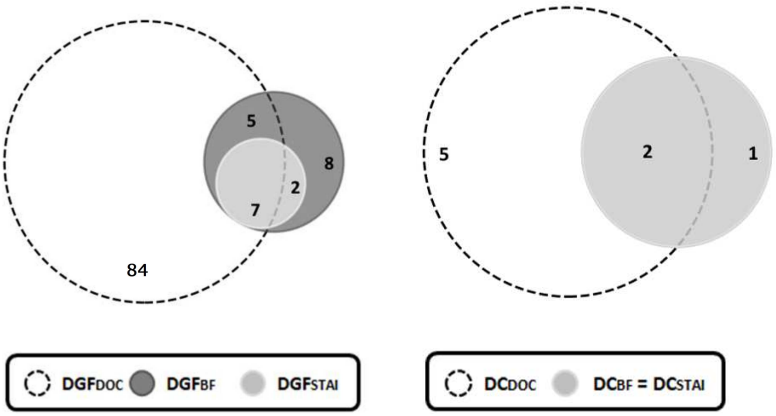
	GUI Failures				**Crashes**			
App	$\#DGF_{Total}$	$\#DGF_{DOC}$	$\#DGF_{BF}$	$\#DGF_{STAI}$	$\#DC_{Total}$	$\#DC_{DOC}$	$\#DC_{BF}$	$\#DC_{STAI}$
A1	12	9	5	1	0	0	0	0
A2	5	5	0	0	0	0	0	0
A3	5	4	3	0	0	0	0	0
A4	8	8	0	0	1	1	0	0
A5	4	3	2	1	0	0	0	0
A6	8	8	0	0	1	1	0	0
A7	7	7	0	0	1	1	0	0
A8	2	1	1	1	0	0	0	0
A9	17	17	3	3	2	2	2	2
A10	5	4	2	0	0	0	0	0
A11	8	6	4	3	1	0	1	1
A12	3	3	1	0	0	0	0	0
A13	4	3	1	0	0	0	0	0
A14	13	13	0	0	0	0	0	0
A15	5	5	0	0	2	2	0	0
Total	106	96	22	9	8	7	3	3

Overall, ALARic found 111 distinct GUI failures and 8 crashes. The team validated as true positives 106 GUI failures and all the crashes. All the apps exposed at least 2 GUI failures and 6 apps exhibited at least one crash. The DOC triggered the highest number (96) of GUI failures and it was able to expose GUI failures in all the considered apps. A total of 22 GUI failures tied to BF were found in 9 apps. STAI triggered 9 GUI failures in 5 apps. As concerns the crashes, the DOC triggered the highest number of crashes (7) in 5 apps. A total of 3 crashes related to the BF sequence were found in 2 apps, whereas STAI triggered 3 crashes in 2 apps.

We analyzed the relations among the sets of issues exposed by each of the 3 considered Lifecycle Event Sequences. As shown by the Venn Diagrams reported in Fig. 4.12, in some cases the same issue was exposed by more than one type of Lifecycle Event Sequence, whereas other issues were triggered by only one Lifecycle Event Sequence type.

As Figure 4.12(a) shows, 7 out of the 106 GUI failures detected by ALARic were found by all the three considered Lifecycle Event Sequences. Among the 96 GUI failures triggered by DOC, 84 were not found by BF and STAI. 8 GUI failures have been triggered only by BF. 5 GUI failures triggered by BF were also caused by DOC but not by STAI. 7 out of 9 GUI failures triggered by STAI, are also detected exploiting both the other 2 Lifecycle Event Sequences. The remaining 2 GUI failures triggered by STAI were also caused by BF but not by DOC.

Fig. 4.12(b) illustrates that only the 2 crashes exposed by $A9$ were triggered by all the 3 considered Lifecycle Event Sequences. 5 out of 8 crashes were triggered only by DOC. Instead, the crash exposed by $A11$ was triggered by BF and STAI but not by DOC.



(a) GUI Failures triggered by the Lifecycle Events

(b) Crashes triggered by the Lifecycle Events

FIGURE 4.12: Issues detected by ALARic

In conclusion, these results suggest that DOC is more likely to expose issues tied to the Activity lifecycle since it has been the most effective in revealing GUI failures and crashes in the experiment. However, BF doesn't have to be neglected since it has shown the capability to discover issues that the other Lifecycle Event Sequences missed. Also STAI that exercises the FL and has a limited impact on the Activity lifecycle, led to the detection of 9 GUI failures and 3 crashes.

On the basis of the obtained results, it is possible to answer the first research question RQ_1 and conclude that:

ALARic detected issues tied to the Activity lifecycle in all the analyzed apps. It exposed both GUI failures and crashes. Lifecycle Event Sequences that exercise diverse key lifecycle loops showed different capabilities in exposing app issues.

TABLE 4.4: Experimental Comparison

App	$\#DC_{ALARic}$	$\#DC_{Monkey}$
A1	0	0
A2	0	0
A3	0	0
A4	1	1
A5	0	0
A6	1	0
A7	1	0
A8	0	0
A9	2	0
A10	0	0
A11	1	0
A12	0	0
A13	0	0
A14	0	0
A15	2	1
Total	8	2

Regarding the comparison between ALARic and Monkey, the data in Table 4.4

shows that for 6 out of 7 apps ALARic was able to find more crashes tied to the Activity lifecycle than Monkey. In $A4$ both the tools exposed the same crash. Moreover, both the tools detected an additional crash in $A9$ that was not tied to the Activity lifecycle. To better understand this result we analyzed in detail the reports produced by Monkey. Monkey was able to seed events that exercise the Activity lifecycle, e.g. orientation changes, back button, but it applied them without a proper strategy, failing in discovering several issues tied to the Activity lifecycle that were found by ALARic, instead. On the basis of this data it is possible to answer to RQ_2 concluding that:

ALARic outperformed the state-of-the-practice tool Monkey in the ability to detect issues tied to the Activity lifecycle. In total, ALARic triggered more crashes than Monkey.

4.5.5 Lesson Learned

The experimental results showed that Lifecycle Event Sequences are able to exercise the Activity lifecycle and to expose failures. The debugging activity performed in the failure validation step showed that the faults causing the failures were mostly located outside the code that overrides the lifecycle callback methods.

As an example, the crash found in $A11$ occurs when the `onSaveInstanceState()` callback method of the `EditProjectActivity` is called, but its cause is located inside the `LinkEditWidget` class that defines a custom GUI object. The programmer indeed overrode the `onSaveInstanceState()` callback method to save at runtime the instance state information of the Activity, as introduced in Section 2.1.3, i.e. the state of the `LinkEditWidget` custom GUI object contained in the `EditProjectActivity` Activity. To this aim, the programmer correctly serialized the `LinkEditWidget` objects and properly implemented the `Serializable` interface in the class that defines the `LinkEditWidget` object. However, the user-defined `LinkEditWidget` contains `android.widget.Spinner` GUI components that do not implement the `Serializable` interface. Therefore a `java.io.NotSerializableException` is thrown at runtime when the lifecycle of the `EditProjectActivity` Activity is exercised.

Another example is related to a failure that regarded 57 out of the 106 GUI failures detected by ALARic. It involved `Dialog` objects disappearing from the GUI after the execution of a Lifecycle Event Sequence. This failure affected most of the considered apps since 12 out of 15 apps exposed it. A `Dialog` is a small window that does not fill the screen and is normally used for modal events that require users to take an action before they can proceed. In most cases, the fault causing these failures has been localized in objects calling directly the public `show` method offered by the `Dialog` or the `AlertDialog Builder` classes to display a `Dialog` on screen. This will correctly pop up the dialog on the screen but the dialog will disappear when the Activity is destroyed and recreated in its lifecycle. Instead, Android guidelines explicitly prescribe that the control of a dialog GUI object (deciding when to show, hide, dismiss it) should be managed by the `DialogFragment` class, which ensures a

correct handling of Lifecycle Event Sequences. This fault resulted to be a common cause of this kind of failures also in the study presented in Section 3.4 that focuses on GUI failures due to orientation changes.

This analysis taught me 2 lessons that could be useful for Android developers and testers. The former lesson is that the Android framework components should be correctly used since they may cause inconsistencies in the app behavior at runtime when Lifecycle Event Sequences occur. The latter is that they should look for faults that may affect the lifecycle of the Activities also outside the methods that override the lifecycle callbacks.

4.5.6 Threats to Validity

This section discusses the threats that could affect the validity of the results obtained in the study [67].

4.5.6.1 Internal Validity

I know that the observed failures might not be caused exclusively by Lifecycle Event Sequences, but also by alternative factors, such as the execution platform or the timing between consecutive events. To mitigate this threat, during the validation step every detected failure was manually reproduced on a real device to exclude that they were tied to the testing infrastructure. A controlled experiment involving different Android OS versions, types of device, and time intervals between events should be carried out to further investigate this aspect.

4.5.6.2 External validity

I am aware that the small sample of considered Android apps may affect the generalizability of the experimental results and we intend to confirm these findings in the future by performing a wider experimentation involving a larger number of apps.

4.6 Related Work

Activity lifecycle has been identified as a major source of issues for Android apps by different works in the literature. Therefore, testing techniques aimed at exposing those issues have been proposed.

Franke *et al.* [81] presented a unit testing approach for verifying the app lifecycle conformance. Their approach considers Activities as units to be tested. Lifecycle-dependent properties have to be manually extracted from functional requirement specification and the Activity lifecycle methods are used to test such properties exploiting assertions. Unlike the fully-automated black-box technique I proposed, their approach heavily relies on manual effort to extract requirements and to define assertion-based unit test cases and requires the availability of the app source code.

The work of Zaeem *et al.* [42] is based on the intuition that different mobile apps and platforms share a set of features referred to as User-Interaction Features and that there is a common sense of expectation of how an app should respond to these features. Among these features they considered also the triggering of the key lifecycle loops. They propose an automated model-driven test suite generation approach and implement it in QUANTUM, a framework that automatically generates a test suite to test the user-interaction features of a given app by leveraging app agnostic test oracles. Differently from ALARic, QUANTUM needs a prior knowledge of the app under test since it requires a user-generated app GUI model as input.

Adamsen *et al.* [43] proposed a tool named `THOR` that systematically amplifies test cases by injecting *neutral* event sequences that should not affect the functionality of the app under test and the output of the original test cases. They focus on event sequences that are usually neglected in traditional testing approaches, including the ones that exercise the key lifecycle loops. THOR leverages existing test cases. Instead, ALARic does not require existing testing artifacts.

Shan et al. [45] focused on a specific fault class due to the incorrect handling of the data that should be preserved when the key Activity loops are exercised. They named *KR errors* the failures caused by these faults. These authors proposed an automated static analysis technique for finding KR errors. They also designed a tool that generates a sequence of input events that lead to the app state where the KR error manifests. Unlike ALARic, their solution needs app modification to verify the failures tied to Activity lifecycle by tracking app fields and dumping GUI states in the Activity lifecycle callback methods.

G. Hu *et al.* [6] introduced `AppDoctor`, a testing system able to perform a quick dynamic analysis of the app under test that aims at revealing app crashes. ALARic is able to detect also GUI failures. Their proposed app analysis, called *approximate execution*, is faster than real execution since it exercises an app by invoking directly event handlers rather than actually performing the corresponding events. ALARic instead triggers real events because they represent better real user interactions. AppDoctor automatically tries to verify the detected bugs by reproducing them using real events since its approximation may introduce several false positives. Like this work, they pointed out the relevance of exercising the Activity lifecycle in mobile testing. Therefore they introduced approximations of lifecycle event sequences among the events supported by AppDoctor.

Moran *el al.* [7] designed Crashscope, a fully automated testing approach for discovering, reporting, and reproducing Android app crashes. They also propose a fully automated black-box testing approach but they focus only on a specific failure type, i.e. app crashes, whereas ALARic is able to find also GUI failures. They identify the double orientation change lifecycle event as a major source of crashes and thus their automated app exploration performs a double rotation each time they encounter a rotatable Activity. Whereas, our approach is able to perform 3 different types of lifecycle events able to cover the 3 key Activity lifecycle loops. Moreover,

the ALARic exploration strategy performs a lifecycle event each time it encounters a GUI state never encountered before during the exploration.

Jun *et al.* [46] proposed LeakDAF, a fully automated testing approach for detecting memory leaks. Like this work, they propose a testing technique targeting Android app components lifecycle conformance that exploits an automated app exploration technique and does not need app modification or manual interaction. They test the apps with 2 lifecycle events that exercise only the Entire Lifecycle loop. Whereas, our work studies the effect of events that exercise also the Visible Lifecycle and Foreground Lifecycle loops. They aim at detecting a specific memory leak type, i.e. the leakage of Activity and Fragment Android app components. Instead, we aim at proposing a testing technique able to detect different types of issues related to the lifecycle of Android app Activities.

IC. Morgado and ACR. Paiva [28] presented the iMPAcT tool, a testing technique based on the presence of recurring behavior in mobile apps, referred to as UI Patterns. Like this work, they propose an automated GUI testing technique targeting Android apps. When iMPAcT detects the occurrence of a UI Pattern during the app exploration, it tests the correct implementation of the pattern by applying the corresponding test strategy. They include in their catalog of UI Patterns a pattern that checks the occurrence of GUI failures due to a single orientation change event. Whereas, our work studies the effect of events that exercise also the VL and FL loops.

4.7 Conclusions and Future Work

In this Chapter, I presented ALARic, an Android automated testing technique that combines the traditional testing approaches based on dynamic app exploration with a strategy that fires mobile-specific events able to expose issues tied to peculiar Android platform features. I focused with my research group on the Android Activity lifecycle management and designed a technique that systematically exercises the lifecycle of app Activities, to detect GUI failures and crashes.

The technique has been implemented in a tool and validated in a study involving 15 real world apps that showed the ability of the tool to automatically detect issues tied to the Activity lifecycle. The study also showed that ALARic is more effective in detecting crashes than standard random tools, such as Monkey, and allowed me to learn some lessons useful for both Android app testers and developers.

As a future work, we plan to extend the ALARic tool by adding other Lifecycle Event Sequences in addition to the 3 already implemented. We intend to propose and implement a set of oracles able to detect other issues tied to the Activity lifecycle, such as memory leaks and threading issues. To better prove the effectiveness of ALARic, we plan to conduct a wider experimentation involving a larger set of Android apps and considering different configurations of the tool. Finally, we plan to extend this approach to test the lifecycle of other app components, such as services, fragments and content providers.

Chapter 5

Combining Automated GUI Exploration of Android apps with Capture and Replay through Machine Learning

In this Chapter I propose juGULAR, a hybrid GUI exploration technique that combines automated GUI exploration with capture and replay. It exploits the human involvement in the automated process to overcome the limitations introduced by classes of GUIs that prevent the exploration of relevant parts of applications if they are not exercised with particular and complex input event sequences. This approach is able to automatically detect these GUI classes during the app exploration by exploiting a Machine Learning approach and to effectively exercise them by leveraging input event sequences provided by the user. juGULAR has been implemented in a modular software architecture that targets the Android mobile platform. An experiment involving 14 real Android apps showed that the hybridization introduced by juGULAR improves the exploration capabilities at a reasonable manual intervention cost. Moreover, the experimental results also proved that juGULAR is able to outperform the state-of-the-practice tool Monkey.

5.1 Introduction

Automated GUI Exploration Techniques (AGETs) have been widely adopted in the context of mobile apps for supporting critical engineering tasks such as testing [2], reverse engineering [36], network traffic generation and analysis [37, 38], performance and energy consumption analysis [18].

Although these techniques provide a viable approach for automatically exercising mobile apps, they suffer from the intrinsic limitation of not being able to replicate human-like interaction behaviors. In fact, some app features need to be exercised by exploiting app-specific knowledge that only human users can provide. As a consequence, these techniques often fail in exploring relevant parts of the application that

can be reached only by firing complex input event sequences on specific GUIs and by choosing specific input values [39, 40].

In the following, I will refer as *Gate GUIs* to the GUIs that need to be solicited by specific user input event sequences to allow the exploration of parts of the app that cannot be reached otherwise. Moreover, I will refer to the action of providing the specific input event sequence needed to effectively exercise a Gate GUI as to the activity of *unlocking* the Gate GUI.

There may be several types of Gate GUIs in real apps, such as *Login* Gate GUIs in which the users need to enter their credentials in order to access to functionality offered by the app to authenticated users only, *Settings* Gate GUIs that require the users to correctly configure the settings of services they intend to use through the app, or *QR code* Gate GUIs that request the users to scan a valid QR code through the device camera to access to particular app features.

The challenges posed by Gate GUIs to the app automated exploration processes are well-known not only in the field of software testing, but also in that of app network traffic signature generation [37] where dynamic analysis is used by large network vendors (e.g. Palo Alto Networks, Dell, HP, Sophos, MobileIron) to trigger app networking activities.

Although most of the automated GUI exploration techniques proposed in the literature do not explicitly address the issues tied to Gate GUIs, some of them offer solutions for unlocking Gate GUIs. Part of these solutions leverages on predefined input event generation rules embedded in the technique [12, 7, 9]. These approaches may not be able to exercise Gate GUIs that need app-specific knowledge. Other solutions require programming skills to understand the app-specific GUI structure and/or configure the AGET to properly manage each distinct Gate GUI [71, 72, 86, 6, 38]. These approaches are indeed labor intensive and may not extend to different applications. Finally, there are solutions that exploit manual user intervention. They suffer from the drawback of requiring an extensive human involvement throughout the entire exploration since the users have to recognize the Gate GUI and intervene in the process to properly exercise it [8, 87].

To address the limitations of existing automated GUI exploration techniques, I realized in collaboration with my research group a novel approach named *juGULAR (Gate gui UnLocking for AndRoid)*. Unlike other approaches, it does not require programming skills, mobile framework knowledge, and app comprehension for unlocking Gate GUIs. juGULAR automatically detects the occurrence of a Gate GUI and exploits human intervention to unlock it at runtime. However, the human intervention to unlock a specific Gate GUI is limited only to the first time that the GUI is encountered. Once the human intervention to unlock a specific Gate GUI is recorded by juGULAR, it can be replayed when the same GUI is detected again during the exploration. The result is a hybrid exploration technique that combines automated GUI exploration with Capture and Replay [88].

A key aspect of our approach is its ability to automatically detect the occurrence

of a Gate GUI. This can be considered as a GUI classification problem that we decided to solve with Machine Learning (ML). We adopted ML techniques to train classifiers to recognize given classes of Gate GUIs and exploited these classifiers to automatically detect Gate GUIs during the exploration.

We implemented the juGULAR hybrid exploration approach in a modular software architecture targeting Android apps and validated it by performing an experiment involving 14 real Android apps. The experiment showed that combining Capture and Replay with automated exploration improves the effectiveness of the exploration. juGULAR covered more source code and Activities and generated more network traffic than the purely automated exploration, thanks to the automatic detection of two classes of Gate GUIs, i.e. Login and Network Settings. The additional time for the manual intervention required by juGULAR was reasonable, being on average lower than 3% of the entire exploration time for all the considered apps. Moreover, the experiment showed that juGULAR outperformed the state-of-the-practice tool Monkey in terms of exploration effectiveness.

This work improves the literature on automated GUI exploration with the following contributions:

- a novel Hybrid GUI Exploration Technique that combines Capture and Replay with automated exploration, named *juGULAR*;
- a Machine Learning approach for the automatic detection of Gate GUIs;
- an experiment involving real Android apps showing the validity of the proposed hybrid technique.

The remainder of the Chapter is organized as follows. Section 5.2 presents a motivating example. Section 5.3 illustrates the Machine Learning-based approach designed for obtaining the Gate GUI classifiers that are used to automatically detect Gate GUIs. Section 5.4 describes the juGULAR approach and how it has been implemented in a software platform. Section 5.5 presents the experiment evaluation and its results. Section 5.6 reports related work. Finally, Section 5.7 reports conclusions and future work.

5.2 Motivating Example

In this section, I present a motivating example to show how the exploration of two real Android apps improves when their Gate GUIs are unlocked. This work focuses on two classes of Gate GUIs: Login and Network Settings.

Login Gate GUIs offer the login feature to registered users. These GUIs require the users to provide the credentials they used to register themselves to the app provider in order to be authenticated. The login feature usually allows to gain access to app functionality restricted to registered users only. Only through the insertion of

valid and previously registered account credentials the exploration of the remaining parts of the application is allowed.

Network Settings Gate GUIs are GUIs exposing Settings features to configure network parameters, such as: URLs, server address, port numbers, channels. This feature is necessary to configure the app access to remote resources.

We selected two publicly available Android mobile apps that expose Gate GUIs, e.g., Twitter[34] and Transistor[35]. Twitter renders the Login Gate GUI shown in Figure 5.1(a), whereas Transistor exhibits the Network Settings Gate GUI reported in Figure 5.1(b).

The Twitter Login Gate GUI requires valid and registered account credentials, without which the access to the app features that are available only to authenticated users is restricted. A purely automated exploration approach could be able to generate syntactically valid login and password, but it cannot be able to automatically generate text strings actually corresponding to a valid Twitter user account. This information should be necessarily defined by a human tester. The Transistor Network Settings Gate GUI requires the user to specify a valid audio stream URL to be reproduced. This scenario is very difficult to be managed by a fully automated approach; even if it is able to automatically generate syntactically valid URLs, it may not be able to generate any correct URL actually corresponding to an audio stream.

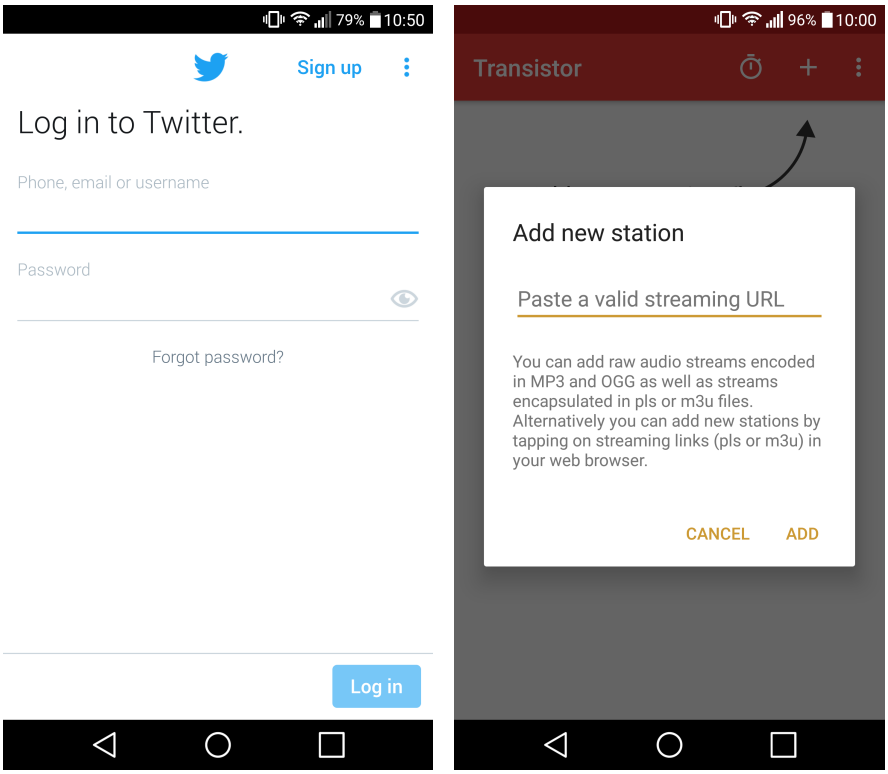
First, these apps were explored by the current state-of-the-practice AGET, Monkey, in its default configuration. Both explorations lasted one hour and were performed on an Android Virtual Device (AVD). We monitored the explorations of Monkey and noticed that it did not unlock the two Gate GUIs, being unable to provide correct credentials for the Twitter authentication, or a valid streaming URL to configure Transistor.

Then, the Gate GUIs were manually unlocked and Monkey was ran again on both apps. Also in this case, Monkey was executed in its default configuration for the duration of an hour. To unlock the Gate GUIs, we interacted manually with the AVD where the apps were installed to provide proper input event sequences able to unlock them.

At the end of each exploration, we evaluated the covered Activities and inferred the Dynamic Activity Transition Graph (DATG) model [30]. It is a graph whose nodes represent the explored Activities and the edges render the transitions triggered at runtime. We enriched this model by adding weights on each edge that indicate the number of times the transition has been traversed.

We inferred the DATG model from the analysis of the system message log dumped by the Android `Logcat`[36] tool. This did not require any app code instrumentation. We parsed the system message log to extract the sequence of the names of the Activities that were created during the app exploration. Activities having different names

[34] Version 6.40.0 - `https://play.google.com/store/apps/details?id=com.twitter.android`
[35] Version 2.2.0 - `https://f-droid.org/repository/browse/?fdid=org.y20k.transistor`
[36] `https://developer.android.com/studio/command-line/logcat.html`



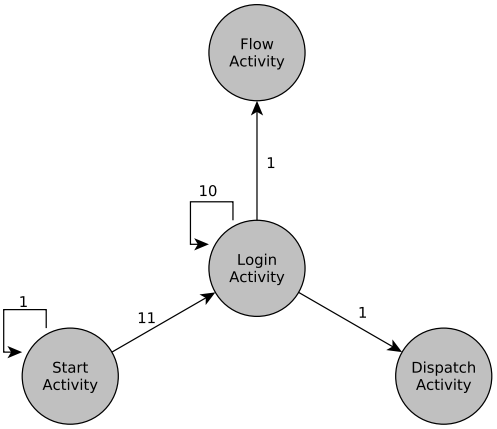
(a) Twitter - Login Gate GUI

(b) Transistor - Network Settings Gate GUI

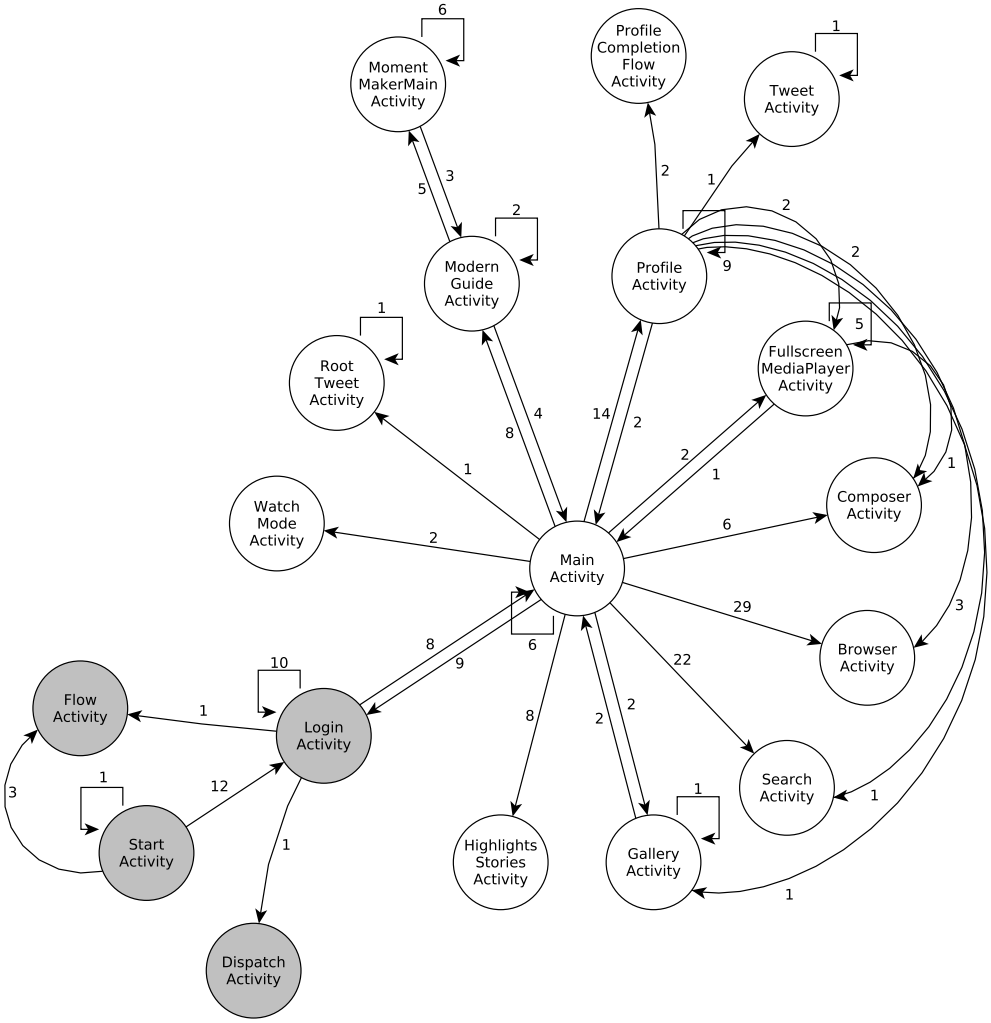
FIGURE 5.1: Gate GUIs exhibited by the considered Android apps

have been considered different nodes of the DATG. Each edge of the DATG links two consecutive Activities in the sequence. Android may not create an Activity every time it is encountered during the exploration but it could reuse a previously created instance of the same Activity. Therefore, we had to enable the on-device `"Don't keep activities"` developer option that destroys every Activity as soon as the user leaves it. In this way, we were sure that each explored Activity was created and thus logged. Of course, this option may change the app behavior, introducing additional invocations of Android framework callback methods. However, a similar behavior may be observed also when the app is exposed to other common events, such as the orientation change event [53]. Since this option was enabled in all the Monkey runs, I believe that the comparison is fair and the usage of the `"Don't keep activities"` option is acceptable for our purposes.

Figure 5.2 shows the DATGs inferred after the two explorations of Twitter. As it can be noticed from the DATG shown in Figure 5.2(a), Monkey was able to discover only 4 different app Activities in the first run, in which the Login Gate GUI was not unlocked. After the Gate GUI was unlocked, Monkey explored up to the 14 previously unreached Activities that are highlighted in white in Figure 5.2(b). These new Activities expose functionality exclusively available to authenticated users, such as showing the user timeline, posting a new tweet or sending a private message to a another Twitter user. We evaluated also the network traffic produced by the app. To



(a) DATG inferred without unlocking



(b) DATG inferred with unlocking

FIGURE 5.2: Twitter App: the DATGs inferred by Monkey explorations without (a) and with (b) Gate GUI unlocking

this aim, we counted the number of bytes transmitted over the network during the app explorations. To obtain this data, We used the `TCPdump`[37] command-line packet analyzer. The unlocking brought improvements also in network traffic generation. Without unlocking, Monkey generated around 1 MBytes of network traffic that increased up to 380 MBytes when valid login credentials were provided.

We were not able to measure the code coverage achieved during the Twitter app exploration, since this app provided compiled and obfuscated code.

Instead, the Transistor app source code is open. Therefore, besides the generated network traffic and the covered Activities, it was also possible to measure the source code coverage. To this aim, we had to preliminarily instrument the app source code by exploiting the `JaCoCo`[38] code coverage library.

As Figure 5.3(a) shows, the exploration of the Transistor app was limited to 2 Activities, when no valid URL was provided in the Main Activity. Instead, Monkey was able to reach a further Activity when a valid audio stream URL was provided, as shown in Figure 5.3(b). This additional Activity offered features for controlling and reproducing the audio stream located at the URL provided to unlock the Gate GUI. Without unlocking, Monkey was able to cover just 9.51% of app LOCs (Lines Of executable Code) and did not generate network traffic, whereas it executed the 58.25% of LOCs and transmitted more than 27 MBytes over the network when a valid audio stream URL was provided.

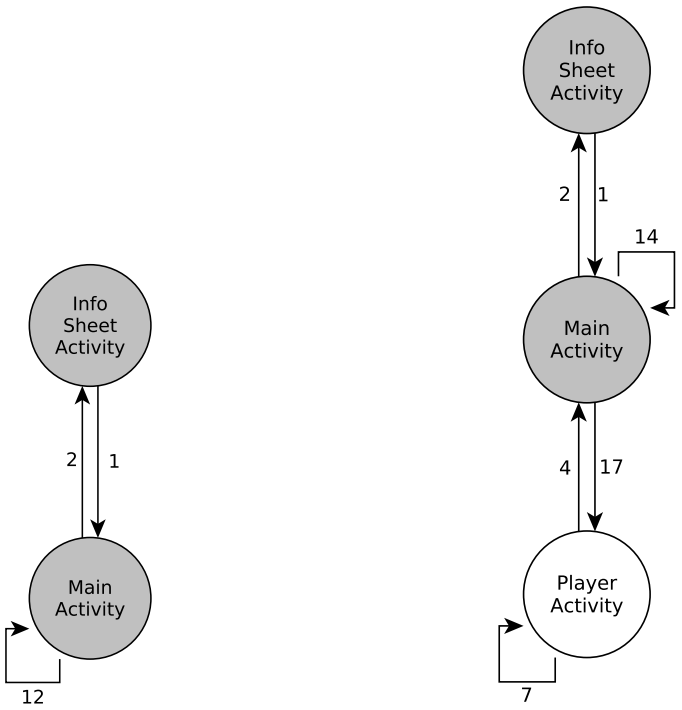
The same two apps were also explored by three state-of-the-art automated GUI exploration tools, i.e., Sapienz[39], AndroidRipper[40] and Dynodroid[41]. Each tool is representative of one the three AGET types reported in Section 5.6, respectively. We analyzed how they dealt with the considered Gate GUIs during the exploration.

Sapienz implements an AGET relying on predefined input generation rules. It did not unlock autonomously the two Gate GUIs, producing unsatisfactory results in terms of covered Activities, LOCs, and generated network traffic.

As for AndroidRipper and Dynodroid, it was possible to unlock the Gate GUIs and to obtain exploration improvements similar to the ones achieved by Monkey. However, these improvements required a considerable manual effort in both cases.

AndroidRipper provides configuration APIs that enable the tool to fire user-defined event sequences on GUI objects belonging to given app GUIs. We exploited these APIs to unlock the Gate GUIs of the considered apps but the tool configuration required a considerable manual effort. In fact, we had to analyze the properties of the specific Gate GUIs and extract the ones needed to detect them at runtime. Moreover, we had to identify the GUI objects to be exercised and define the corresponding events.

[37] http://www.tcpdump.org/
[38] http://www.eclemma.org/jacoco/
[39] https://github.com/Rhapsod/sapienz
[40] https://github.com/reverse-unina/AndroidRipper
[41] http://www.seas.upenn.edu/~mhnaik/dynodroid.html



(a) DATG inferred without Gate GUI unlocking

(b) DATG inferred with Gate GUI unlocking

FIGURE 5.3: Transistor App: the DATGs inferred by the Monkey explorations without (a) and with (b) Gate GUI unlocking

Dynodroid implements instead an AGET that can exploit manual intervention at runtime. Therefore, a human tester was actively involved for the entire duration of the explorations to supervise the Dynodroid execution and to manually unlock the Gate GUIs. He had to monitor the Activities that were encountered by Dynodroid and to promptly stop the exploration each time he recognized a Gate GUI. After the human stopped Dynodroid, he manually unlocked the GUI by providing a valid input event sequence and then restarted the automatic exploration.

These experiences exposed the limitations of currently available techniques for automated app exploration and motivated us to investigate novel and more effective solutions.

5.3 A Machine Learning-based approach for detecting Gate GUIs

To define juGULAR, it was necessary to preliminarily define an approach for automatically recognizing whether the GUIs that are encountered during the app exploration belong to a Gate GUI class. To solve this kind of classification problems,

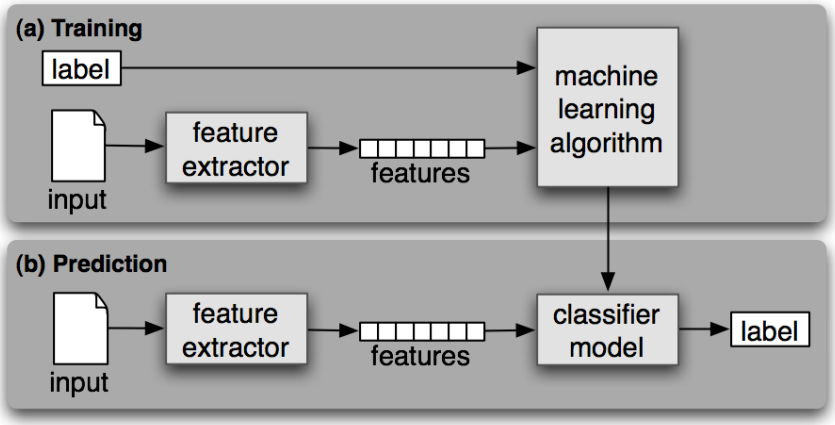


FIGURE 5.4: Supervised Classification framework. The upper part of the figure (a) represents the Training phase. The lower part of the figure (b) shows the Prediction phase.

rule-based or machine learning techniques are commonly employed in the literature [89].

Rule-based approaches exploit expert knowledge to make decisions. Rules are obtained from experts who encode their conditional beliefs into heuristics manually crafted for each specific class. These rules are usually elicited by error-prone and time-consuming processes that require a strong expertise about the considered domain [90]. Moreover, these rules work effectively only if all the possible situations under which decisions can be made are known ahead of time.

Instead, Machine Learning approaches aim to learn how to classify automatically through experience [91]. Therefore, they do not require expert involvement and are more effective at deriving general rules for classification problems, finding insights in data that may be underestimated by a human.

In this work, I with the collaboration of my research group defined a Machine Learning (ML) approach that trains a supervised classifier to determine the class a given GUI belongs to. Figure 5.4 shows the general framework used for the supervised classification[42]. According to this framework, the supervised classification foresees 2 main phases: Training and Prediction. During the Training phase, a feature extractor is used to convert each input item instance to an abstract representation. This representation consists in a *Feature Vector* that captures the basic information about each input that should be used to classify it. In this phase, each input item is provided with a label that identifies the class the item belongs to. Pairs of feature vectors and labels are fed into a machine learning algorithm to generate a classifier model. The trained classifier can be used to predict the class of unseen input item instances. During the Prediction phase, the same feature extractor is used to convert unseen inputs to feature vectors. These feature vectors are then fed into the classifier model, which generates predicted labels.

[42] http://www.nltk.org/book/ch06.html

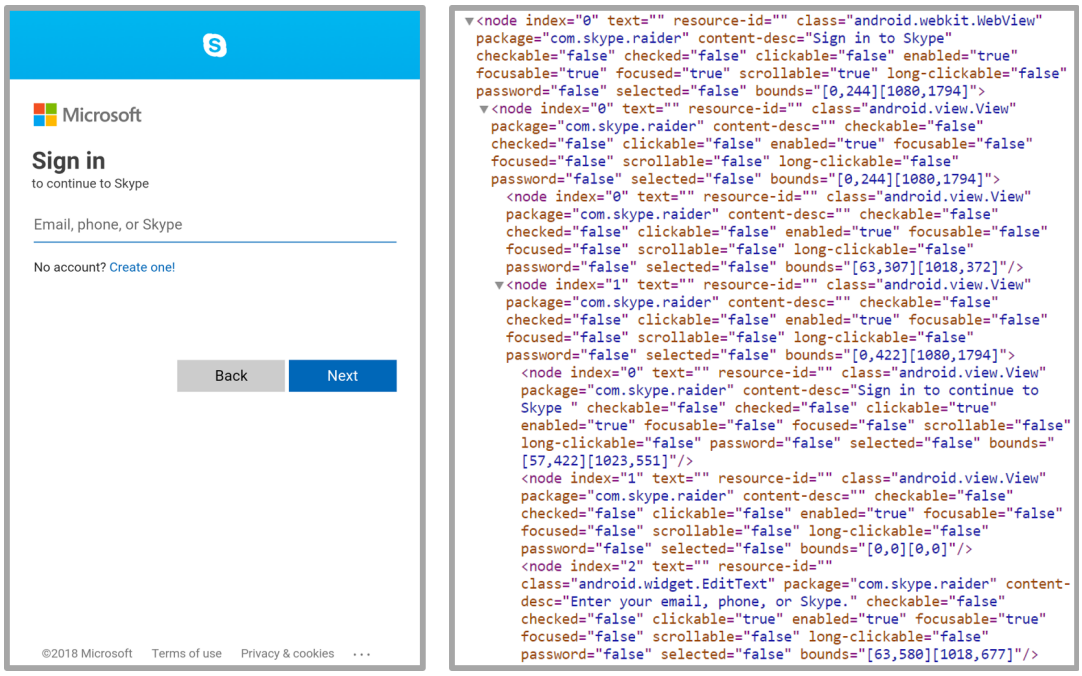


FIGURE 5.5: A GUI (left) and an excerpt of its XML Description (right)

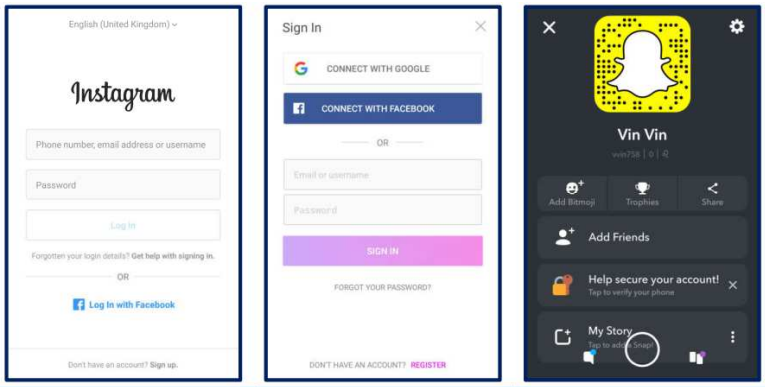
To obtain the feature vector associated to a GUI, our approach relies on a component-based GUI description model that abstracts the GUI in terms of its component objects and their properties [53, 59, 92]. In particular, this GUI description leverages the XML GUI representation provided by UI Automator[43]. Among all the component object properties, we consider the ones that contain textual information, *i.e.* id, text, hint, and content description. Figure 5.5 shows an example of a GUI and an excerpt of its XML description.

We assume that the descriptions of GUIs belonging to the same Gate GUI class are likely to share common textual information that we refer to as *keywords*. We select as features the presence or absence of such keywords in the values assumed by the considered properties.

In our approach, we did not want to arbitrarily predefine the keywords to be considered in the classification problem, but we wanted to empirically infer them for each considered Gate GUI class. For this purpose, we chose as keywords the most frequent terms among the GUIs belonging to the same Gate GUI class.

Figure 5.6 presents our intuition about how a GUI can be characterized by the presence of a set of keywords. The Figure reports 4 keywords that should characterize Login Gate GUI descriptions and shows whether they are present in 3 GUI descriptions belonging to different Android apps. The GUI instances in Figure 5.6(a) and in Figure 5.6(b) are actually Login screens and their descriptions present at least 3 out of the 4 distinctive keywords. Instead, the GUI instance in Figure 5.6(c) is not a Login screen and its description does not contain any of the considered keywords.

[43] https://developer.android.com/training/testing/ui-automator.html



Keyword	(a)	(b)	(c)
forgot	✓	✓	✗
login	✓	✗	✗
password	✓	✓	✗
facebook	✓	✓	✗

FIGURE 5.6: GUI Textual Information Content. The table in the lower part of the figure reports the presence (✓) or absence (✗) of the "forgot", "login", "password", "facebook" keywords in the description of the Instagram (a), PicsArt (b), and Snapchat (c) GUIs shown in the upper part of the figure.

In the following, I describe the process we designed to select the keywords and for training the classifiers. This process is depicted in Figure 5.7 and consists of 3 main activities: *Dataset Construction*, *Keyword Extraction* and *Classifier Training*. I implemented it with the collaboration of my research group by using the features provided by the Natural Language Toolkit 3.2.5[44] platform.

This process is based on Information Retrieval approaches that solve the problem of classifying documents into a set of known categories, given a set of documents along with the classes they belong to [91]. More specifically, we adopted semi-structured retrieval since we consider the XML representation of GUIs. This kind of approaches is well-known and is used to solve several classification problems, such as detection of spam pages, unwanted content, and sentiments, or email sorting [91] and app reviews' content classification [93]. To the best of my knowledge, we have been the first to use these techniques to solve the mobile app GUI classification problem and to improve the app automated exploration.

The process is general and it has been exploited for the 2 specific Gate GUI classes considered in this study: Login and Network Settings. The same process can be reused to build new classifiers for automatically detecting other GUI classes.

5.3.1 Dataset Construction

Since there was no existing base of knowledge to be used for our purposes, we built our own dataset consisting of GUI descriptions belonging to real Android apps and

[44] http://www.nltk.org/

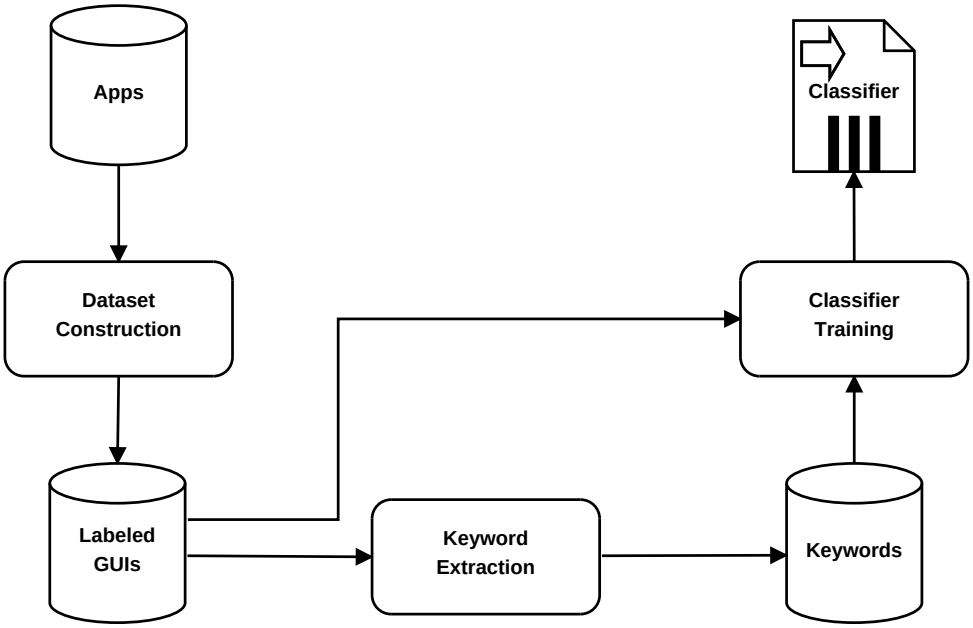


FIGURE 5.7: The Machine Learning based process for selecting the keywords and training Gate GUI classifiers. It is composed of three activities: Dataset Construction, Keyword Extraction and Classifier training.

labeled them according to our needs.

To this aim, we randomly picked 5,000 real Android apps that were distributed by the official Google Play store[45] and recruited 100 Computer Engineering M.Sc. students to obtain a set of labeled descriptions of GUIs belonging to these apps. Each student was asked to manually explore 50 of the selected apps and label their GUI interfaces by assigning them one of three possible labels: *Login Gate GUI*, *Network Settings Gate GUI*, *Other* (i.e. a GUI that can not be classified as one of the 2 considered Gate GUI classes). The students were provided with a *GUI Labeler* desktop application we developed to support the GUIs labeling task. The tool allows to select a label and assign it to the description of the GUI currently rendered on the device screen connected to the host PC. The tool was developed in Python and relied on the Android Debug Bridge (adb)[46]. The tool produces as output an image file of the captured screen in PNG format and the UI Automator GUI hierarchy in XML format with the chosen label. Figure 5.8 shows the interface of the GUI Labeler tool.

We provided each student with a device equipped with Android 6, that was reset to the factory settings to ensure that each capture was executed in the same conditions. Moreover, the system language was set to *English* in order to avoid inconsistencies among the captures. Each student was asked to complete the assigned task within a month and to spend at least 15 minutes and not more than 30 minutes for exploring each app.

[45] https://play.google.com/store/apps
[46] https://developer.android.com/studio/command-line/adb.html

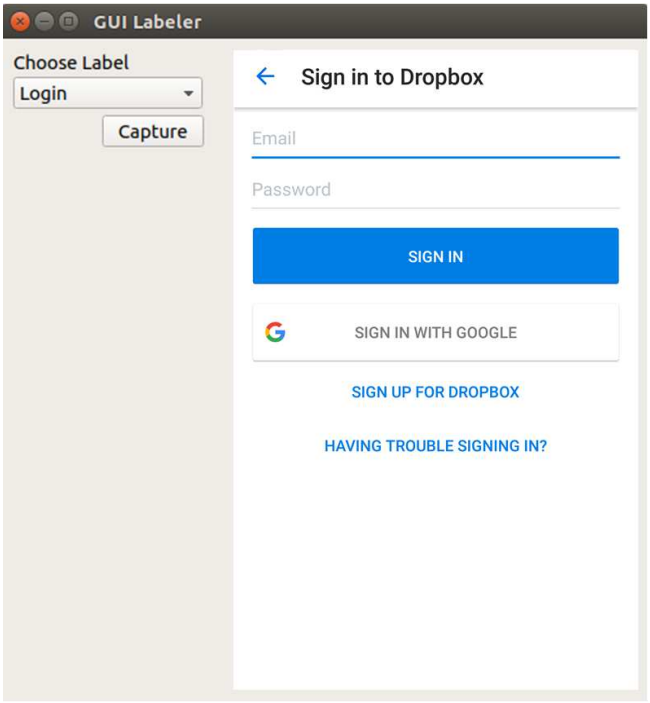


FIGURE 5.8: An example of GUI Labeler tool interface. In this case the user captured a Dropbox GUI and assigned it the "Login" label.

Upon the completion of the GUI Labeling task by all the recruited students, I with other 2 Ph.D. students and a Postdoctoral Researcher belonging to the research group and having knowledge of Android development, reviewed the labeled XML descriptions aided by the correspondent screen captures in order to validate them.

At the end of this step, we selected 400 XML descriptions for each of the 3 considered labels and stored the resulting 1200 descriptions in a repository of labeled GUIs.

5.3.2 Keyword Extraction

This activity allows to obtain a set of distinctive keywords for each Gate GUI class starting from GUI descriptions belonging to that class. Therefore, it has to be repeated for each considered Gate GUI class.

To this aim, for each Gate GUI class we partitioned the set of 400 XML descriptions labeled as belonging to the considered class in 2 subsets of 200 XML descriptions that I hereafter refer to as $G1$ and $G2$, respectively. The $G1$ subset was submitted to a keyword extraction process including linguistic preprocessing steps [91]. The $G2$ subset was instead exploited for training the classifier.

The keyword extraction process consisted of 5 steps:

1. *XML Nodes Extraction:* in this step, each XML GUI description belonging to $G1$ was filtered to obtain the XML nodes related to its Android `View` objects[47]. These objects represent the elements composing the GUI. We considered the

[47] https://developer.android.com/reference/android/view/View.html

values of the XML node attributes containing textual information, i.e *resource-id*, *hint*, *text*, *content-desc*. These values provided us a set of strings associated with each GUI description.

2. *Text Normalization:* in this step, special symbols and punctuation marks were removed from all the strings and each string was split into its constituent words. If a word was an identifier using the camel-case convention, it was split into the composing words (e.g., "processFile" is split into "process" and "File"). Finally, we converted each resulting word to lowercase.

3. *Stop words Removal:* in this step, we removed English *stop words* (like `and`, `a`, `to`, `do`, `of`) from the normalized strings. These stop words frequently appear in many GUI description and do not help much in differentiating one GUI description from another. We also removed terms specifically related to the Android SDK that are general and are not discriminating to identify Gate GUIs, e.g. `View`, `Toolbar`, `Button`.

4. *Stemming:* in this step, words were transformed to their root forms exploiting the Porter Stemming Algorithm[48]. For example, `localization`, `localized`, `localize`, and `locally` were all simplified to `local`.

5. *Term Frequency Evaluation:* in this step, the words obtained from the XML GUI descriptions belonging to $G1$ were gathered in a single set of terms, named T1. For each term of T1, the term frequency (*tf*) value was calculated producing a rank. The *tf* value of a term is equal to the number of occurrences of the term in the document or a corpus of documents [91]. In our study, we considered as corpus the set of terms $T1$. As an example, if a term *term1* occurs 40 times in $T1$, then its *tf* value will be 40. The terms having a *tf* greater or equal to a given threshold were selected and used to define a keyword set. We used more threshold values to define different keyword sets. We built 7 sets of keywords, using threshold values varying from 5 to 35, with a step of 5. Each set of terms obtained at the end of this process provided a candidate set of *keywords*.

In the following, I present a simple example to illustrate the Keyword Extraction process. In this example, I submit to the Keyword Extraction process the description of a GUI with a button having `"Click here if you have problems logging in!"` as textual label and `"login_troubleshooting_button"` as identifier.

In the XML Nodes Extraction step, the Button View object is identified and the values of its XML node attributes *resource-id*, *hint*, *text*, *content-desc* are returned. The output of this step is the following set of strings {Click, here, if, you, have, problems, logging, in!, loginTroubleshooting_button}.

In the Text Normalization step, the "!" and "_" special symbols are removed from the set. Moreover, the "loginTroubleshooting" string is split into "login" and "Troubleshooting" strings. Finally, all the words are converted to lowercase. As a result,

[48] https://tartarus.org/martin/PorterStemmer/

the following set of strings is obtained {click, here, if, you, have, problems, logging, in, login, troubleshooting, button}.

In the Stop words Removal step, the English *stop words* "here", "if", "you" and "in" along with the generic Android SDK term "button" are removed from the set. The output of this step is the following set of strings {click, problems, logging, login, troubleshooting}.

In the Stemming step, the words in the latter set are transformed in their root form, obtaining the final set of terms {click, problem, log, login, troubleshoot}.

This set of terms will be gathered with the ones obtained from the other XML GUI descriptions belonging to the considered corpus. The resulting set of terms will be submitted to the Term Frequency Evaluation step, in which the *tf* value is calculated for each term and compared against the considered threshold. The terms having a *tf* value greater than the threshold will finally provide the set of keywords.

5.3.3 GUI Classifier Training

For each considered Gate GUI class, a distinct Binary classifier [94, 95] had to be trained. Binary classification is a type of supervised learning in which a classifier is used to distinguish between a pair of classes. It is trained by using examples of objects belonging to both classes. We train a Login Gate GUI Binary classifier to predict whether a GUI belongs to the Login Gate GUI class or not. Moreover, we train a Network Settings Gate GUI Binary classifier to predict whether a GUI belongs to the Network Settings Gate GUI class or not. To train each classifier, we used 2 sets, the former consisting of 200 XML GUI descriptions labeled as belonging to each considered Gate GUI class (and different from the ones exploited in the Keyword Extraction step) I previously referred to as $G2$. The latter, that I hereafter refer to as $G3$, made of 200 XML GUI descriptions labeled as not belonging to the considered class.

Each labeled GUI description belonging to $G1 \cup G2$ was automatically processed by executing the XML Nodes Extraction, Text Normalization, Stop Words Removal, and Stemming steps. Then, it was associated to a *Feature Vector* of binary values in which the i_{th} element represents the presence (1) or absence (0) of the i_{th} keyword in the GUI description. This step was repeated 7 times, each one considering a different candidate set of features, thus obtaining 7 distinct classifiers.

We decided to use Naïve Bayesian (NB) classifiers as Binary classifiers. An NB classifier is a statistical classifier based on the Bayes'theorem that implements a simple, computationally efficient classification algorithm. NB classifiers are widely employed in several areas, including text classification, with comparable results to decision trees and artificial neural networks [96].

We trained and validated each classifier using a 10-*fold cross-validation process. In this process, an original dataset is randomly divided into 10 equal-sized

subsamples that are exploited for 10 validation steps. At each validation step, a single subsample is used for validation and the other nine subsamples are utilized for training.

Finally, we compared the accuracies obtained by the 7 classifiers and chose the one achieving the best F-measure value [91]. For both Gate GUI classes, the selected classifier was obtained using the set of keywords corresponding to the *tf* threshold of 20.

Table 5.1 reports for both the considered Gate GUI classes the average values of precision, recall and F-measure the selected classifiers obtained over the 10 validation steps.

TABLE 5.1: Performance in terms of average values of precision, recall and F-measure of the Login Gate GUI and Network Settings Gate GUI classifiers

	Login Gate GUI	**Network Settings Gate GUI**
Precision	0.814	0.751
Recall	0.807	0.900
F-measure	0.807	0.813

5.4 The proposed Hybrid GUI Exploration Technique

In this Section, I present our hybrid GUI exploration technique named juGULAR that targets Android mobile apps. Since Android apps are event-based software systems [22, 97], juGULAR explores the analyzed apps by automatically sending events to them. Our technique explores mobile apps regardless of whether they run completely on the Android device, or they belong to more complex and distributed systems. Indeed, juGULAR aims to explore the client-side Android app of such systems and can interact with their remote side by events that are fired on the UI of the Android client app.

Unlike other event-based exploration techniques reported in the literature [25, 92], juGULAR implements a novel approach that pragmatically combines fully automated GUI exploration with Capture and Replay, in order to enhance the app exploration and to minimize the human involvement. It is able to automatically detect Gate GUIs during the app exploration by exploiting classifiers that can be obtained through the Machine Learning approach introduced in Section 5.3. Moreover, it can unlock Gate GUIs by leveraging input event sequences provided by the user through a Capture and Replay technique.

The exploration implemented by juGULAR extends the automated GUI exploration algorithm presented in [25]. The workflow of juGULAR is described by the UML Activity diagram shown in Fig. 5.9. The Activity states describing the steps

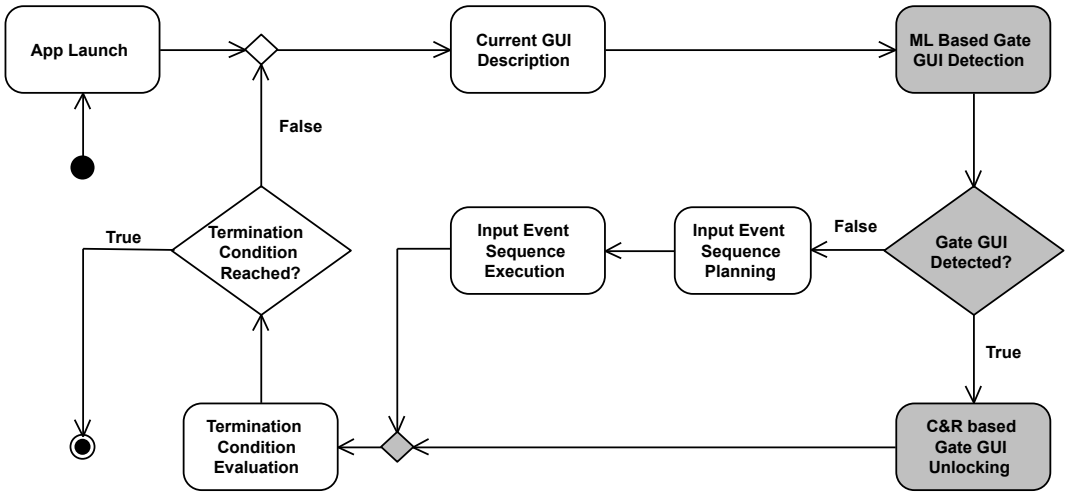


FIGURE 5.9: UML Activity Diagram describing the juGULAR workflow

of the original algorithm are reported in white, whereas the ones introduced by our approach are in gray.

Each app exploration is started by the *App Launch* step that installs and launches the app on an Android device. In the *Current GUI Description* step, a representation of the GUI state currently exposed by the app is inferred. It includes the $(attribute, value)$ pairs assumed by GUI components at runtime. The GUI description is analyzed in the *Gate GUI Detection* step to evaluate whether it is an instance of a Gate GUI.

If the current GUI is not a Gate GUI, the *Input Event Sequence Planning* and *Input Event Sequence Execution* steps are executed. In these steps, an event is chosen among all the ones triggerable on the current GUI and then it is executed.

juGULAR considers as triggerable the predefined sets of events that can be fired on GUI objects having the properties `clickable`, `enabled`, and `visible` set to `true` in the current GUI description. The value of the `type` attribute of the GUI object determines the set of possible events that can be triggered on it. As an example, events like `click` and `longclick` can be fired on `Button` and `ImageView` objects, whereas `selectItem` and `scroll` events can be sent to `ListView` objects.

Otherwise, if juGULAR detects a Gate GUI, the *Gate GUI Unlocking* step is executed. In this step, either an input event sequence will be captured for unlocking a Gate GUI or a recorded input event sequence will be replayed.

The *Termination Condition Evaluation* step evaluates whether the termination condition is met and the exploration can be stopped.

The UML Statechart diagram in Figure 5.10 provides an overview of how juGULAR combines automated app exploration with Capture and Replay.

In the *App Exploring* state, juGULAR iteratively fires input event sequences to the subject app according to a given input event generation strategy until it detects a Gate GUI, or a predefined termination condition is met. When juGULAR detects

a Gate GUI, it evaluates whether it has been previously encountered or not. To this aim, it compares the current GUI description with the ones of the Gate GUIs already encountered during the app exploration. Two GUI descriptions are considered as equivalent if they include the same set of objects and the same values of the objects' attributes [25].

If juGULAR detects a Gate GUI for the first time, it stores its GUI description and transits to the *Unlocking Input Event Sequence Capturing* state, where it captures an unlocking input event sequence that is manually provided by the user.

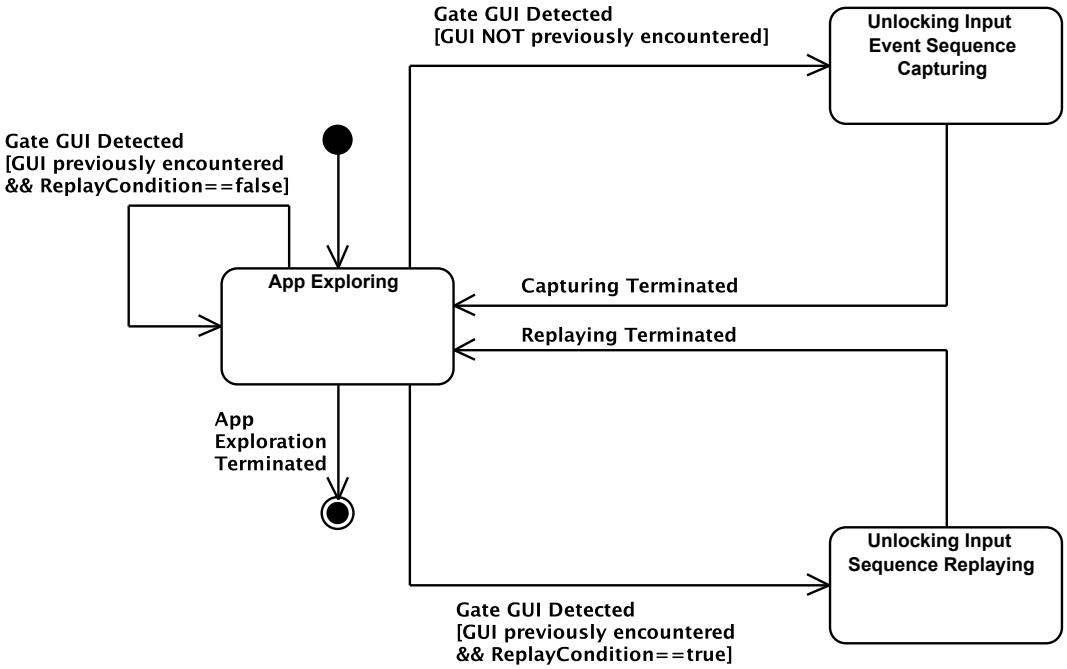


FIGURE 5.10: UML Statechart Diagram describing how juGULAR combines automated app exploration and C&R techniques

When the capturing ends, juGULAR returns to the *App Exploring* state. In this state, if juGULAR detects a Gate GUI that was previously encountered, it either will continue the automated app exploration, or will transit into the *Unlocking Input Event Sequence Replaying* state in which the corresponding input event sequence is replayed. The choice of the next state will depend on the value of a `ReplayCondition` random boolean variable that assumes the `true` value with a predefined probability p_{true}. The usage of this variable prevents the exploration process from being biased by the user's choice for unlocking a given Gate GUI.

At the end of the replaying, juGULAR returns to the *App Exploring* state.

It is worth pointing out that, in the *Replay* step, non-determinisms of the app may cause the app to expose a GUI that is different from the one exercised in the Capture, or to behave differently from the recorded behavior. In these cases, the recorded event sequence replay does not guarantee that the Gate GUI will be correctly unlocked. This is a known weakness of Capture and Replay approaches [40] and poses a limitation to juGULAR. In fact, juGULAR returns in the *App Exploring* state after

each Replay, regardless of whether the recorded event sequence has successfully unlocked a Gate GUI. In case of app non-deterministic behavior, there is the risk that juGULAR indefinitely encounters the same Gate GUI and tries to unlock it with the same recorded event sequence. The `ReplayCondition` random boolean variable mitigates this risk since it allows juGULAR to fire also event sequences different from the recorded Unlocking Input Event Sequence when it re-encounters a Gate GUI.

5.4.1 The juGULAR Platform

We implemented juGULAR in a software platform which targets Android mobile apps. An overview of the platform architecture is reported in Figure 5.11.

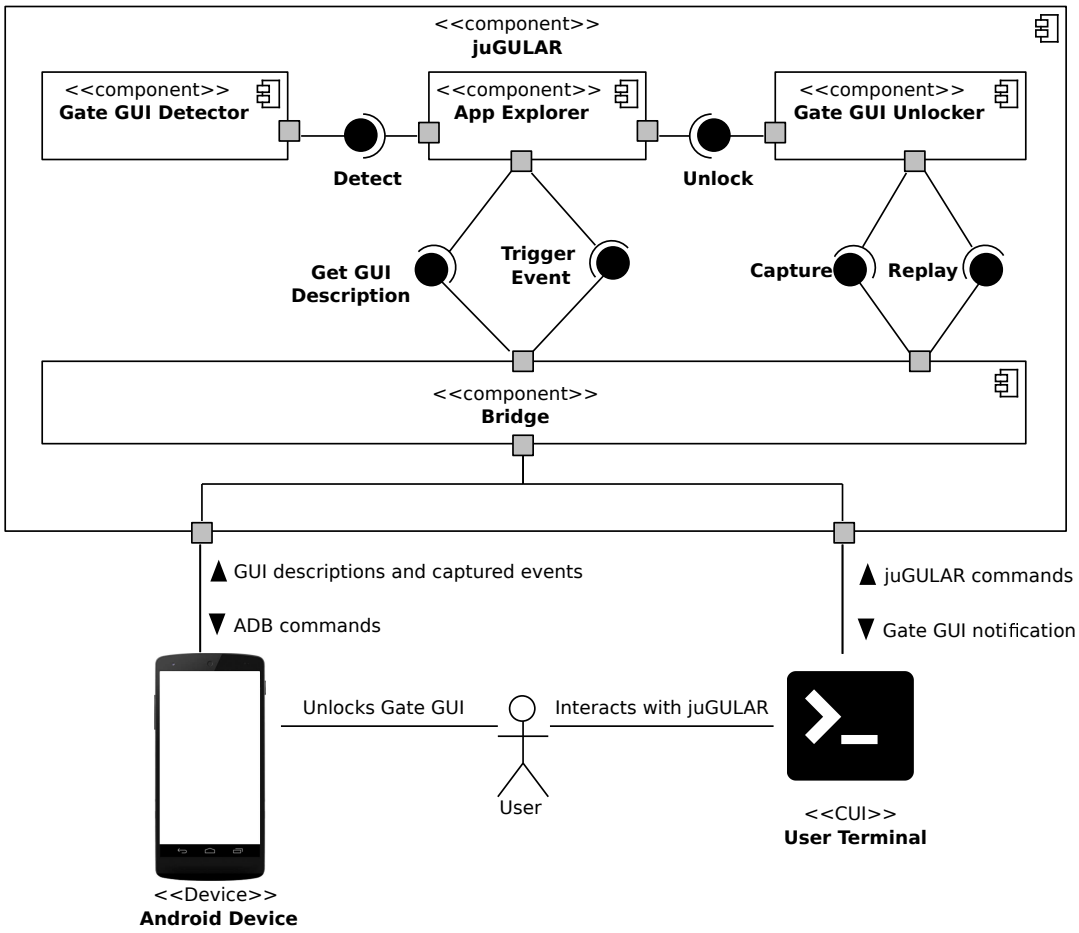


FIGURE 5.11: UML Component diagram describing the juGULAR platform architecture

The core of this architecture is the juGULAR component that embeds four inner components, namely *App Explorer*, *Gate GUI Detector*, *Gate GUI Unlocker*, and *Bridge*.

The *App Explorer* implements the app exploration logic. The *Gate GUI Detector* has the responsibility to automatically infer whether a GUI belongs to a Gate GUI class. The *Gate GUI Unlocker* offers the feature to unlock a Gate GUI. The *Bridge* allows the juGULAR components to interact both with a *User Terminal* and with an *Android Device* where the app being explored is installed and executed.

The *User Terminal* allows the users to launch juGULAR and to receive notifications when a Gate GUI is detected for the first time and should be unlocked. Thanks to this feature, the users do not have to monitor the app exploration waiting for a Gate GUI detection, but juGULAR notifies them when their intervention is needed for unlocking a Gate GUI. When the users have accomplished the capture activity, they resume the app exploration via the User Terminal.

The juGULAR components and the User Terminal are deployed on a host PC running either Windows or Linux operating system and equipped with the Android SDK[49]. The PC must be connected through the Android Debug Bridge (adb)[50] with an Android virtual device (avd)[51] hosted on the host machine, or a real Android device connected to the host machine via a USB connection.

Our platform can be used to explore an Android app for reaching different goals. Depending on the specific goal, additional tools can be exploited for capturing relevant information about the performed exploration. As an example, in the study that I present in Section 5.5, we aimed at evaluating the app coverage and the network traffic generated by the exploration. To reach this goal, we instrumented the app source code using the `jaCoCo` library[52] and ran tcpdump[53] on the host machine to get the network packets capture file in `pcap` format.

Additional implementation details about the platform components are reported in the following.

5.4.1.1 App Explorer Component

This component can be configured to explore an app using different exploration strategies, such as the Random or Active Learning ones [25]. The strategy determines the next event to be triggered on the app. The *App Explorer* uses the *Trigger Event* and *Get GUI Description* APIs offered by the *Bridge* component to send events to the app and to retrieve the description of the current GUI rendered by the device, respectively.

Moreover, it uses the *Detect* API offered by the *Gate GUI Detector* to assess whether the current GUI can be classified as a Gate GUI. When a Gate GUI is detected, the App Explorer uses the *Unlock* API provided by *Gate GUI Unlocker* component for unlocking it.

5.4.1.2 Gate GUI Detector Component

This component offers the *Detect* API that is exploited by the *AppExplorer* to assess whether a GUI can be classified as a Gate GUI. Its architecture is represented in Figure 5.12 by a UML Component diagram.

[49] Available for free download at `https://developer.android.com/studio/index.html`
[50] `https://developer.android.com/studio/command-line/adb.html`
[51] `https://developer.android.com/studio/run/managing-avds.html`
[52] `http://www.eclemma.org/jacoco/`
[53] `http://www.tcpdump.org/tcpdump_man.html`

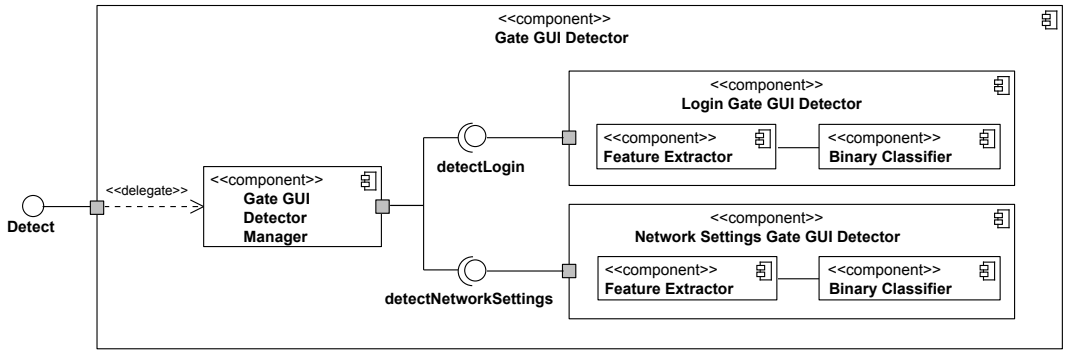


FIGURE 5.12: UML Component diagram describing the Gate GUI Detector architecture

The *Gate GUI Detector* comprises 3 components, i.e. the *Gate GUI Detector Manager*, the *Login Gate GUI Detector* and the *Network Settings Gate GUI Detector*. Each inner detector is able to detect a different Gate GUI class, i.e. Login and Network Settings.

The *Gate GUI Detector Manager* takes as input a GUI description in XML format, forwards it to the inner *Gate GUI Detector*s and gathers their outputs. According to these outputs, the *Gate GUI Detector Manager* returns a boolean value indicating whether a Login Gate GUI or a Network Settings Gate GUI have been detected.

The *Login Gate GUI Detector* and the *Network Settings Gate GUI Detector* exploit the Binary classifiers and the set of keywords obtained by the approach proposed in Section 5.3. Each of these components is in turn composed by a *Feature Extractor* and a *Classifier* component.

The *Feature Extractor* takes as input the GUI description and represents it as a feature vector. To this aim, it elaborates the GUI description through the preprocessing steps introduced in Section 5.3, i.e. XML Nodes Extraction, Text Normalization, Stop Words Removal, and Stemming. Then, it associates the GUI description to a vector of binary values, where the i_{th} element represents the presence (1) or absence (0) of the i_{th} keyword. Finally, it forwards the feature vector to the corresponding *Classifier*.

The *Gate GUI Detector* is a modular ensemble of Binary classifiers and thus it can be easily extended by introducing additional Gate GUI identifiers for other classes of Gate GUIs.

5.4.1.3 Gate GUI Unlocker Component

The *Gate GUI Unlocker* component provides the *Unlock* API that requires as input a GUI description in XML format. It stores in a local repository the descriptions of the Gate GUIs encountered during the app exploration. Moreover, it stores for each GUI description the sequence of user events that was recorded to unlock the corresponding GUI.

When the *Unlock* API is invoked, the component checks whether the input description matches with one of the descriptions stored in the repository.

If the *Unlocker* does not find any matching GUI description, it requires to capture a sequence of user events using the *Capture* API provided by the *Bridge*. Upon completion of the capturing, the input GUI description along with the captured sequence of user events will be stored in the repository.

Otherwise, the *Unlocker* either will invoke the *Replay* API to replay the related sequence of user events, or will return the control to the *App Explorer*, on the basis of the value assumed by the random boolean variable `ReplayCondition`.

The *Gate GUI Unlocker* can be configured by setting the p_{true} value the `ReplayCondition` relies on. The default value of p_{true} is 0.9.

5.4.1.4 Bridge Component

The *Bridge* component allows juGULAR to interact with the device where the app runs and with the User Command Prompt. It provides the *Trigger Event* and *Get GUI Description* APIs that are used by the *App Explorer* to send events to the app and to retrieve the XML description of the current GUI rendered by the device, respectively. These APIs are realized exploiting the UIAutomator framework[54].

The Bridge provides also the *Capture* and *Replay* APIs that are used by the *Gate GUI Unlocker* component. The *Capture* API sends a notification on the *User Terminal* to the user about the occurrence of a Gate GUI and records the event sequence the user performs to unlock it. The *Replay* API is used for replaying the recorded user event sequence that unlocks a specific Gate GUI.

Android encodes each user input event (e.g. Tap, Long Tap, Scroll, Hardware Button Press) into a large group of kernel-level events. I will refer to the former ones as *high-level user event*s to distinguish them from kernel-level events.

The *Capture* and *Replay* APIs have been developed exploiting the *getevent* and *sendevent* tools, respectively[55]. These tools, shipped within the Android SDK, are able to capture and replay the kernel-level event stream produced by the user interaction.

The *getevent* tool provides a live dump of kernel-level input events. The captured stream contains information about the input events, such as their timestamps, input device names, event types, and screen coordinates. The *sendevent* tool allows developers to send kernel-level events to the device.

As already reported in other works [88], the *sendevent* command does not allow to set the timing between consecutive sent events. Therefore, we had to develop an ad-hoc solution to replay the kernel-level events with a proper timing. This solution is intended both to avoid a too quick replay of high-level events and to faithfully replay them. To this aim, we implemented in the Bridge a pipeline that post-processes the input event stream captured by *getevent* and transforms it into an output stream

[54] https://developer.android.com/training/testing/ui-automator.html
[55] https://source.android.com/devices/input/getevent

suitable to be replayed by *sendevent*. The pipeline groups meaningful chunks of kernel-level events representing high-level events, inserts delays between them, and translates the stream in the format supported by *sendevent*.

The stream chunks are isolated using a set of clustering rules that are based on timestamps and event encoding patterns that Android exploits for transforming high-level user events into kernel-level ones. We inferred these patterns by analyzing streams of kernel-level events produced by triggering user events. The pipeline uses these rules to decompose each high-level event into a sequence of lower level events, e.g. Press, Release, Move. For clarity, I refer to these as *low-level events*. Each low-level event is in turn composed by a sequence of kernel-level events having the same timestamp and that is ended by a `SYN` code made of a sequence of zero values.

Figure 5.13 shows an example of a kernel-level event stream captured by *getevent* and how the clustering rules abstract from it 4 types of high-level events, i.e., Tap, LongTap, Scroll, and Key Home Press. The columns in the Figure report the timestamp between square brackets, the input device id followed by a colon, and an hexadecimal string representing the encoding of the kernel-level events. The leftmost brace groups consecutive kernel-level events into low-level events, whereas the rightmost brace groups low-level events into high-level user events.

The figure illustrates that a Tap event is composed by the sequence of Press and Release low-level events between which there is a delay less than 500 ms. A Long Tap is composed by a sequence of Press and Release between which there is a delay greater or equal to 500 ms. A Swipe is made by a sequence of a Press and a Release interspersed with one or more Move low-level events. A Key Home Press event is composed by a sequence of a Key Home Down and a Key Home Up low-level events.

Finally, the pipeline processes the captured stream composed by kernel-level events and produces an `unlocking sequence description` file. This file will be provided to the *Gate GUI Unlocker* component and it will be interpreted by the Bridge *Replay* API to unlock the corresponding Gate GUI by sending the kernel-level events with the proper timing.

Figure 5.14 reports the unlocking sequence description file corresponding to the stream shown in Figure 5.13. An unlocking sequence description file includes sequences of kernel-level events supported by *sendevent* along with specific commands, i.e. `#Start`, `#End`, `#SendEvents`, and `#Sleep`. The kernel-level events in this file are obtained from the corresponding ones in the captured event stream by removing the timestamps and the colons and by translating the hexadecimal values in decimal format. The `#Sleep` command is used to introduce delays between events. The Bridge adds delays of 1500 milliseconds between kernel-level event chunks, each one representing a single high-level event. This is intended to mitigate the risk of failures in the replay step due to a time not long enough to complete a requested UI task, e.g. UI updating, Web resource fetching. We found that a delay of 1500 milliseconds is long enough to complete mobile UI tasks in our benchmark apps. Moreover, a

delay of 600 milliseconds is introduced between the Press and the Release of Long Tap events to not mistake them for simple Tap events.

The event stream stored in the unlocking sequence description file allows the *Replay* API to execute the events on the same screen coordinates of the corresponding event stream acquired through the *Capture* API. We chose to stick to the same coordinates since the juGULAR architecture has the limitation that the same device is used both for capturing and replaying user event sequences. I am aware that if different devices want to be used in the Capture and Replay steps, our solution should be enhanced.

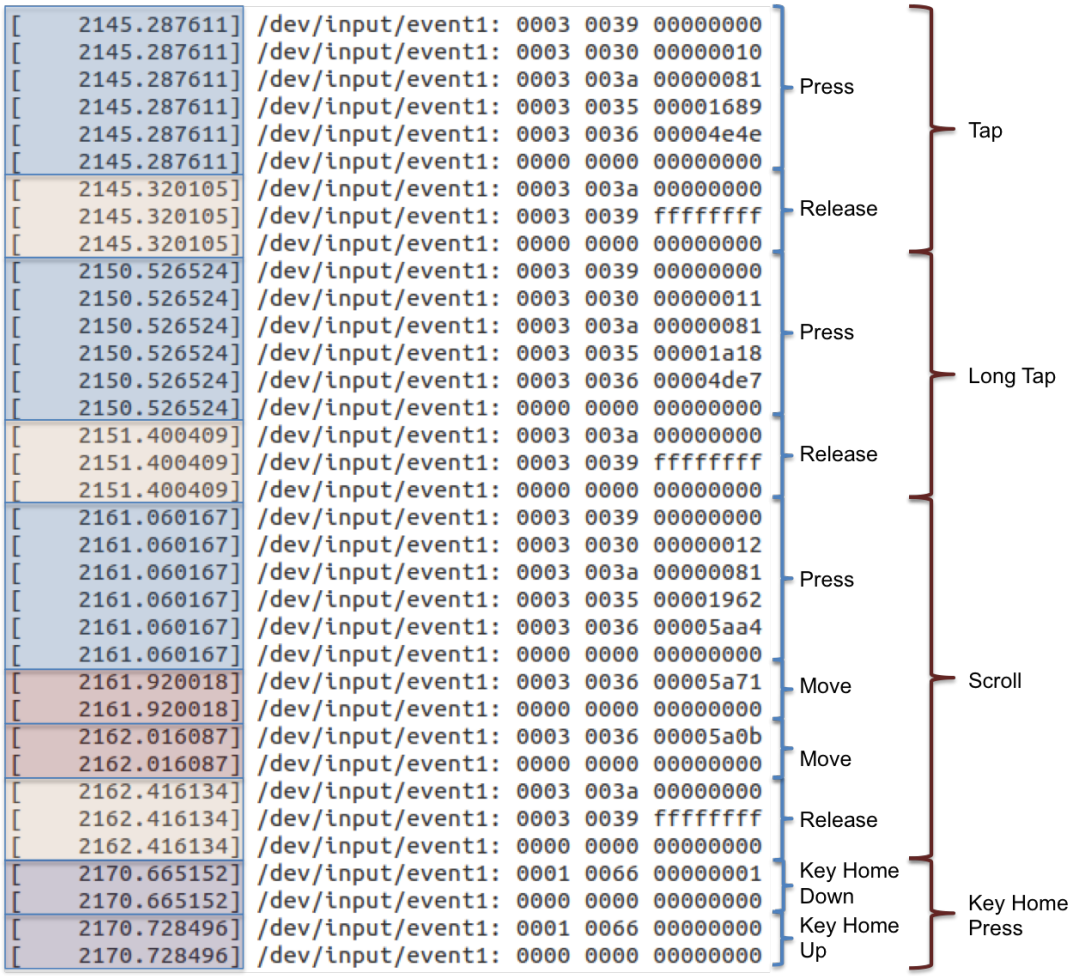


FIGURE 5.13: A sequence of kernel-level events captured by getevent. Each line reports a kernel-level event characterized by its timestamp, input device id, and the corresponding hexadecimal code. The leftmost brace groups consecutive kernel-level events into low-level events, whereas the rightmost brace groups low-level events into high-level user events.

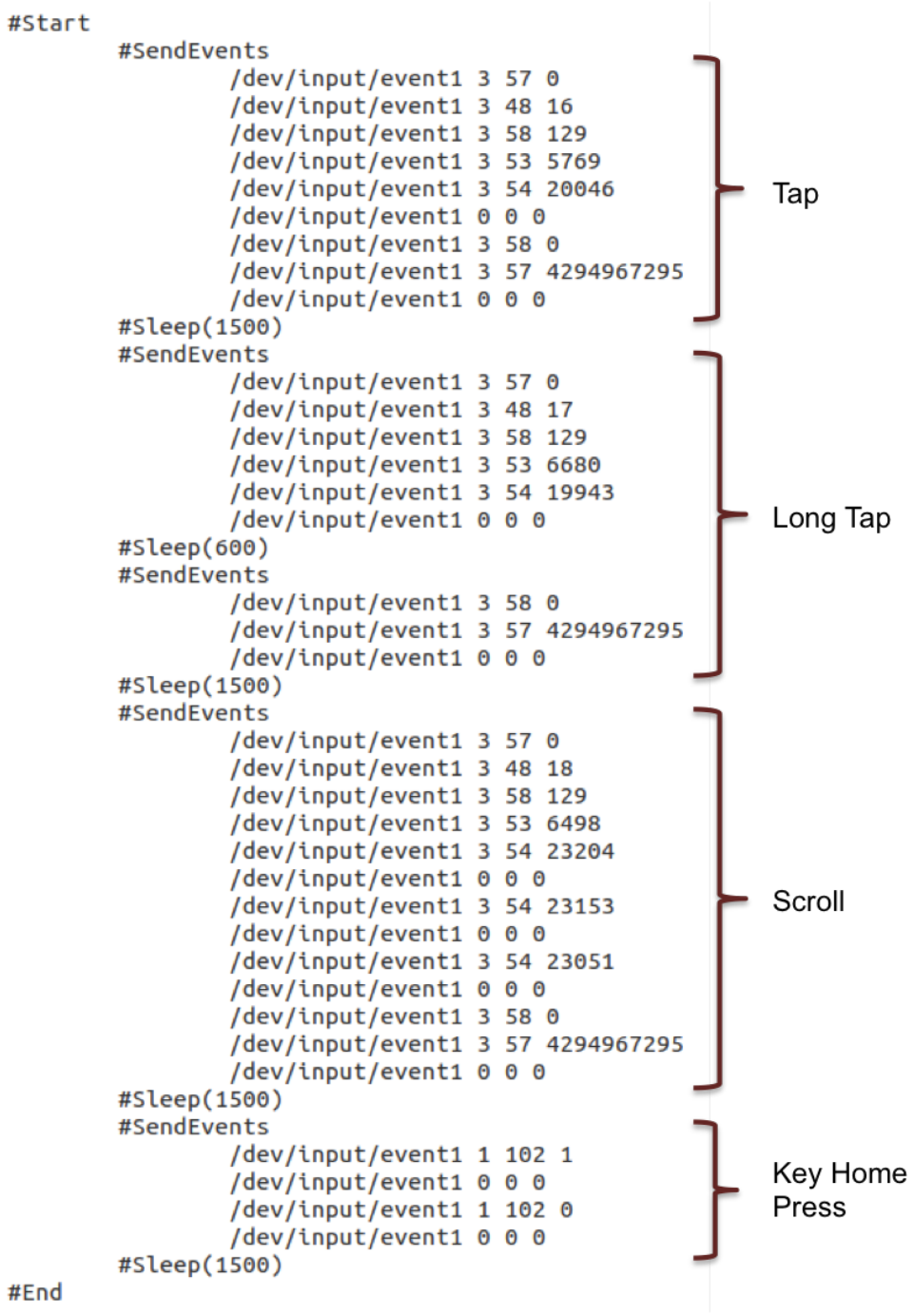


FIGURE 5.14: An Unlocking Sequence Description File. The sequence is contained within `#Start` and `#End` commands and includes five `#SendEvents` commands, followed each by a sequence of kernel-level events to be provided to the *sendevent* tool. The `#Sleep` commands are used to introduce delays between consecutive events.

5.5 Experiment

In this section, I describe the study I conducted with my research group to evaluate the performance of juGULAR. The exploration technique implemented by juGULAR can be exploited in different contexts and for reaching different objectives. Thus, in this study, we considered two usage scenarios of juGULAR: software testing and mobile app network traffic signatures generation.

Our goal was to understand how the hybridization proposed by juGULAR does impact the ability of fully automated GUI exploration techniques in analyzing apps and at what cost. Moreover, we were interested in evaluating how juGULAR compares with other state-of-the-practice AGETs. More precisely, the study aimed at answering the following three research questions:

RQ$_1$ How does the hybridization introduced by juGULAR affect the effectiveness of an automated exploration technique?

RQ$_2$ How does the manual intervention required by juGULAR affect the costs of the hybrid exploration approach?

RQ$_3$ How does the exploration effectiveness of juGULAR compare to the effectiveness of the AGET implemented by the state-of-the-practice Monkey tool?

5.5.1 Objects Selection

The presence of Gate GUIs in the object Android apps was a requirement for carrying out this study. Therefore, we needed to select apps that exposed at least one GUI belonging to the considered Gate GUI classes. To this aim, we chose a subset of apps from the official Google Play store whose GUIs belong to the dataset we built in the process described in Section 5.3 and that were not used in the Keyword Extraction and Classifier Training activities. In addition, since we wanted to evaluate also the code coverage reached due to the app exploration, we required that the selected apps belonged also to F-Droid.

Among the apps that satisfied these criteria, we randomly chose a sample made of 14 apps. Table 5.2 reports for each app its ID, the app name, the name of the Android app package, the considered app version, and a brief description of the app functionality. Table 5.3 instead shows for each app the app ID, the total number of Activities, the number of LOCs, the number of classes, the number of methods, and the presence of GUIs belonging to the considered Gate GUI classes.

We considered only the Java classes that contain the app code, i.e. we took into account neither the Java classes belonging to third party libraries nor the code written in native C/C++ used to develop the app.

As it emerges from the data shown in the tables, the selected apps are sufficiently diverse since they offer different functionality and have a variable size both in terms of Activity number and LOCs.

TABLE 5.2: Android apps involved in the study

App ID	App Name	Package Name	Version	App Description
A1	Flym News Reader	net.fred.feedex	1.9.0	Simple, modern and totally free RSS reader.
A2	Conversations	eu.siacs.conversations	1.19.5	Jabber/XMPP client for Android.
A3	DAVdroid	at.bitfire.davdroid	1.5.0.3-ose	Calendar synchronization app.
A4	Transistor Radio	org.y20k.transistor	2.2.0	App for listening to radio over internet.
A5	k9-Mail	com.fsck.k9	5.206	Email client supporting multiple accounts.
A6	mGit	com.manichord.mgit	1.5.0	Git client and text editor.
A7	Muspy	com.danielme.muspyforandroid	3.4.48	Client for Muspy.com.
A8	OpenRedmine	jp.redmine.redmineclient	3.20	Android Redmine client.
A9	OwnCloud	com.owncloud.android	2.3.0	Android client for private ownCloud Server.
A10	PortKnocker	com.xargsgrep.portknocker	1.0.11	App that pings a specific TCP/UDP port.
A11	LibreTorrent	org.proninyaroslav.libretorrent	1.4	Original Free torrent client.
A12	Connectbot	org.connectbot	1.9.2-oss	Powerful open-source Secure Shell (SSH) client.
A13	PodListen	com.einmalfel.podlisten	1.3.6	Free Podcast app.
A14	ServeStream	net.sourceforge.servestream	0.7.3	Open source HTTP streaming media player and media server browser.

App ID: unique identifier of the app.
App Name: name of the app.
Package Name: name of the application package.
Version: version of the app.
App Description: description of the main functionality offered by the app.

TABLE 5.3: Characteristics of the Android apps involved in the study

App ID	# App Activities	# App LOC	# App Classes	# App Methods	Presence of Gate GUIs	
					Login	Network Settings
A1	8	4,487	195	762		✗
A2	20	23,548	634	3,675	✗	
A3	10	4,498	284	850	✗	✗
A4	3	2,313	135	424		✗
A5	27	29,829	919	5,249	✗	✗
A6	10	4,394	232	921	✗	✗
A7	10	3,671	258	1,035	✗	
A8	16	9,638	495	2,716	✗	✗
A9	22	18,840	481	2,973	✗	✗
A10	5	1,272	97	321		✗
A11	9	8,436	247	1,436		✗
A12	12	7,256	236	1,198	✗	✗
A13	4	3,904	210	681		✗
A14	13	7,256	200	1,079		✗

App ID: unique identifier of the app.
 # App Activities: number of Activity classes of the app.
 # App LOC: overall number of executable Lines Of Code of the app.
 # App Classes: overall number of classes of the app.
#App Methods: overall number of methods exposed by the classes of the app.
Presence of Gate GUIs: the ✗ marker indicates the type of Gate GUI exposed by the app.

5.5.2 Subjects Selection

Since the juGULAR approach requires manual intervention of an end user to unlock the encountered Gate GUIs, we recruited 14 M.Sc. Software Engineering students. The selected subjects were involved in the study for providing the Unlocking Input Event Sequences, when needed, during the automated exploration of the object apps. They had a background on software testing and on network traffic analysis matured during their studies. They were selected by an interview. They had no prior in-depth knowledge about the selected apps nor a thorough knowledge about the underlying concepts of the Android Framework.

5.5.3 Metrics Definition

In this section, I describe the metrics we chose to answer the proposed research questions.

5.5.3.1 Effectiveness Metrics

Since in the study we focused on software testing and network signatures generation scenarios, we decided to evaluate the effectiveness of juGULAR as the ability to: (1) cover app Activities, (2) cover app Lines of Code (LOC), and (3) generate network traffic. To this aim we defined the following set of metrics:

- The *Covered Activities* percentage ($CA\%$) reports the percentage of the Activities covered during the exploration on the total number of App Activities; it can be measured according to the following formula:

$$CA\% = \frac{\#\,Covered\ Activities}{\#\,App\ Activities} * 100$$

- the *Covered Lines of Code* percentage ($CLOC\%$) defines the percentage of the app LOC exercised during the automated exploration on the total number of the App LOC. It is expressed by:

$$CLOC\% = \frac{\#\,Covered\ LOCs}{\#\,App\ LOCs} * 100$$

- the *Network Traffic Bytes* (NTB) metric responds to the need to evaluate the ability of the technique to trigger the generation of network traffic; it measures the number of bytes received or sent on the network by the app during the exploration and it is expressed by the following formula:

$$NTB = \#\,App\ Sent\ Bytes\ +\ \#\,App\ Received\ Bytes$$

5.5.3.2 Manual Intervention Cost Metric

To evaluate the cost of the manual intervention required by juGULAR, we considered for each app exploration process, the time spent in each capture activity, i.e. $CaptureTime_i$, and the Total Exploration Time of the technique. Therefore, we used the:

- *Manual Intervention Time Percentage* ($MIT\%$) that defines the percentage of time spent in the human interventions required for unlocking the encountered Gate GUIs on the total exploration time. It is expressed by:

$$MIT\% = \frac{\sum_i CaptureTime_i}{TotalExplorationTime} * 100$$

5.5.4 Experimental Procedure

The experimental procedure we designed for carrying out the study consisted of three sequential steps: *Training*, *Apps Exploration*, *Data Collection and Analysis*.

In the *Training* step, the researchers involved in the definition of juGULAR explained its approach to the selected subjects. The training was carried out through examples, where the researchers illustrated how the technique works, the type of Gate GUIs it is able to detect and the features it provides for unlocking them. In order to verify that all the subjects had correctly understood the approach and how to provide Unlocking Input Sequences when needed, in the last part of the Training, the subjects were asked to perform an exploration process on a sample Android app we appositely developed. The sample app exposed both a Login and a Network Settings Gate GUIs. Each subject was asked to provide Unlocking Input Sequences when needed. We did not instruct the subjects on the Unlocking Input Sequence to provide, they were free to insert the sequence they considered the most appropriate. At the end of this process, we analyzed the results of the exploration sessions. All the subjects were able to correctly adopt juGULAR and to provide the Unlocking Input Event Sequences. The entire Training step lasted 8 hours.

In the *Apps Exploration* step, we executed 3 different exploration processes. In the first process we used juGULAR, in the second one we exploited juGULAR with the Hybridization Disabled, and in the third we employed Monkey. JHD is an ad hoc juGULAR configuration that performs the automated app exploration without exploiting the hybrid features, i.e. detection and capture and replay. In the following, we name JHE the actual implementation of juGULAR. Since Monkey implements a random app exploration and we wanted to implement a fair comparison among the considered tools, we configured also JHE and JHD to explore the apps using a random exploration strategy. Regarding the process involving JHE, we decided to repeat each app exploration with different subjects, in order to mitigate the dependence of the exploration effectiveness on a specific subject's judgment. We divided the selected subjects in 2 groups made of 7 students, namely G_1 and G_2. We gave the

TABLE 5.4: The Android apps assigned to each group of students

GROUP	APP IDs
G_1	A3, A4, A7, A9, A11, A12, A13
G_2	A1, A2, A5, A6, A8, A10, A14

subjects of each group the task of exploring 7 of the object apps, that were randomly assigned to each group. Table 5.4 reports the object apps assigned to the groups. To carry out the exploration tasks with JHE, we provided each student with a PC equipped with the tool. The students had to launch the app explorations and to intervene in the process only when a Gate GUI was encountered for the first time. For each app, since the explorations were random, 3 runs lasting 60 minutes had to be executed by each subject. We configured the `ReplayCondition` to assume the `true` value with a probability $p_{true} = 0.8$ so that the recorded Unlocking Input Event Sequences were not the only ones to be executed when a Gate GUI was detected. At the end of the Exploration step, we obtained 21 exploration runs for each app. A researcher controlled that subject performed the experiment according to the instructions provided in the Training step. The experiment was conducted under "exam conditions", i.e., subjects were not allowed to communicate with others for not biasing the experimental findings. As to the processes involving JHD and Monkey, we launched 21 exploration runs lasting 60 minutes for each app. In this way we obtained the same number of explorations as JHE. All the explorations were carried out on desktop PCs having an Intel(R) Core(TM) i7 4790@3.60GHz processor and 8 GB of RAM, running a standard Nexus 5 Android Virtual Device (AVD)[56] with Android API 19; the host PC was equipped with the Ubuntu OS, version 16.04. Each experiment was executed on a newly created AVD.

In the *Data Collection and Analysis* step, we analyzed the reports produced by JHE, JHD, and Monkey to obtain the number of Activities and LOC covered during the explorations, as well as the generated network traffic bytes for the three exploration processes. As for the explorations with JHE, we also evaluated the capture times spent by each subject during the three exploration runs performed on each app.

5.5.5 Experimental Results

Table 5.5 reports the average effectiveness values we measured for the explorations carried out with JHE, JHD and Monkey, respectively. The average values have been obtained with respect to the 21 exploration runs for each app. The Table also reports the average values of all the metrics calculated considering all the selected apps.

The same results are shown by the histograms in Figure 5.15 that provide a graphical visualization and allow the comparison among the average values of CA%, CLOC% and NTB obtained with JHD, JHE and Monkey.

As emerged from the analysis of the reports produced by the JHE explorations, juGULAR successfully detected all the Gate GUIs we identified in the object apps.

[56] https://developer.android.com/studio/run/managing-avds.html

TABLE 5.5: Effectiveness results of the app explorations performed by JHD, JHE, and Monkey

App ID	JHD - juGULAR Hybridization Disabled			JHE - juGULAR Hybridization Enabled			Monkey		
	CA%	CLOC%	NTB	CA%	CLOC%	NTB	CA%	CLOC%	NTB
A1	50	20.25	11,128	75	49.19	3,170,055	20	24.49	0
A2	30	8.33	0	50	18.64	352,196,871	30	5.65	0
A3	40	10	2,256,788	60	28.63	5,450,397	40	12.09	2,055,974
A4	66.7	12.21	0	100	62.73	38,894,905	66.6	10.18	0
A5	7.4	4.22	0	25.93	37.14	93,987,389	7.4	4.9	0
A6	30	14.43	16,890	80	46.61	3,259,397	40	19.18	15,404
A7	10	15.41	33,722	50	49.09	6,295,654	10	18.63	30,462
A8	25	4.96	0	31.25	16.09	74,088	25	6.01	0
A9	4.5	8.11	854,033	22.73	23.90	6,376,599	4.5	9.43	787,737
A10	60	53.62	0	60	60.53	2,275	60	44.65	0
A11	12.63	22.22	2,834,611	44.4	39.4	670,369,445	33.3	25.83	4,947,046
A12	33.3	14.25	103,550	58.3	41.21	15,987,272	33.3	17.24	0
A13	91.6	41.8	6,758	100	43.34	241,584	75	38.4	9,978
A14	30.76	9.61	11,243,538	53.9	30.65	30,496,185	30.7	11.28	1,679,895
Avg	35.13	17.10	1,240,072	57.96	39.08	87,628,722	33.98	17.71	680,464

App ID: unique app identifier.
CA%: average percentage of covered Activities.
CLOC%: average percentage of covered executable lines of code.
NTB: average number of Bytes sent and received on the network by the app.

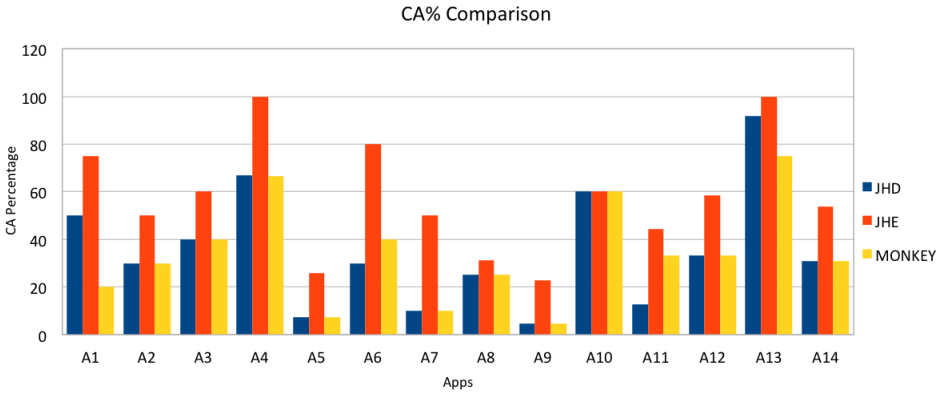
These data also showed that each subject spent different amounts of capture time to unlock each Gate GUI.

In order to answer RQ_1, we compared the average effectiveness values obtained using JHE and JHD.

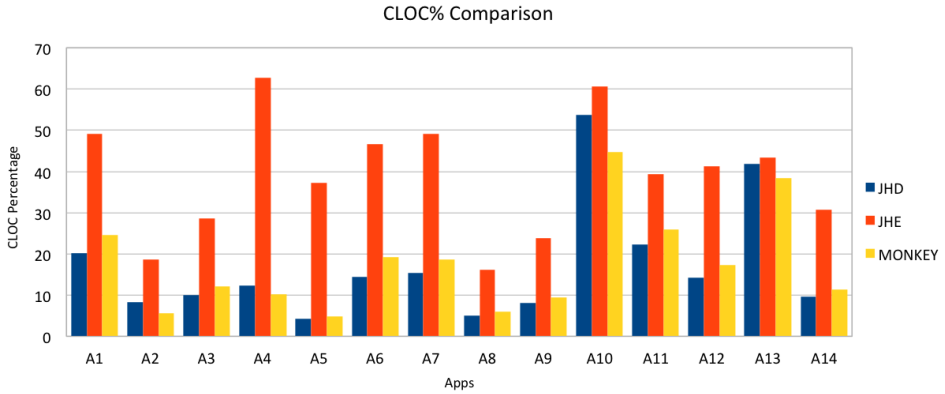
As regards the Activities exploration capability, the CA% data values reported in Table 5.5 show that JHE covered a greater percentage of Activities than JHD in 13 out of the 14 object apps. Only for $A10$ (PortKnocker), juGULAR achieved the same results covering 3 out of 5 Activities either with or without the hybridization. However, the 2 unexplored Activities of this app could not have been reached otherwise, since one of them is rendered only for older Android versions and the other one is accessible only from the app external widget. The average $CA\%$ increment with JHE was of 23%, while the minimum and maximum increment values were of 6.25% in $A8$ and up to 50% in $A6$, respectively.

In the case of $A8$ (OpenRedmine), the reduced increment in Activities coverage was essentially due to the choice of the input event sequences values provided by the subjects to unlock the Login Gate GUI. We observed that all the app features regarding the project management could not be exercised even after the unlocking. This happened since all the credentials they provided were associated to Redmine repository accounts having no associated projects. On the contrary, as to the $A6$ (mGit) app, we obtained a considerable Activity coverage increment with JHE, because at least one subject provided Login credentials associated with accounts having non-empty project repositories.

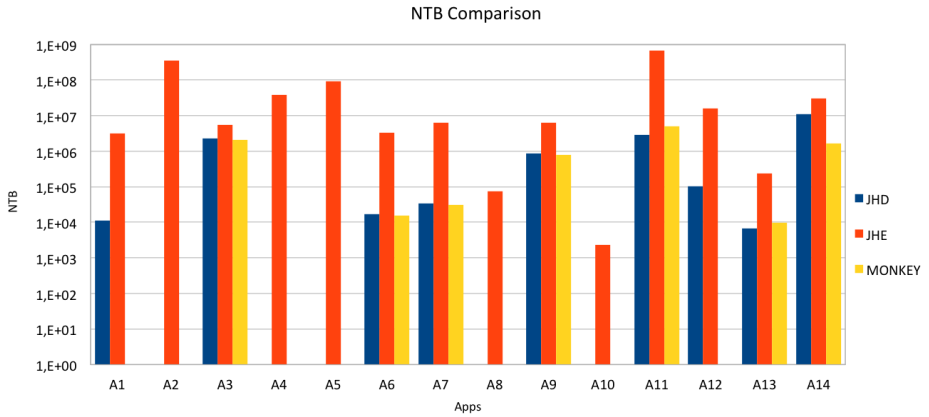
As for the Source Code exploration capability, the CLOC% data values reported



(a) Average CA% values obtained by JHD, JHE, and Monkey



(b) Average CLOC% values obtained by JHD, JHE, and Monkey



(c) Average NTB values obtained by JHD, JHE, and Monkey

FIGURE 5.15: Average effectiveness results of the explorations executed by JHD, JHE, and Monkey

in Table 5.5 show that JHE always covered a wider percentage of source code than JHD, with an increment of 22%, on average. The minimum increment of code coverage percentage was of 1.54%, and was observed in $A13$ (PodListen). The Network Settings Gate GUI exposed by the app was unlocked even without hybridization since PodListen allows the user to subscribe to a podcast not only by adding a podcast URL but also selecting a predefined podcast provided by the internal podcast database. The maximum increment of code coverage percentage was instead of 50.52%, and was observed in $A4$ (Transistor). This app exposed a Network Settings Gate GUI that should be unlocked to execute the code that implements the features for controlling and reproducing audio streams. The only way to unlock this Gate GUI was to provide a valid audio stream URL as had been done by the subjects.

Regarding the ability of generating network traffic, the hybridization allowed juGULAR to obtain impressive results. JHD was not able to produce any network traffic in 5 out of the 14 object apps. Considering all the apps, JHD was able to produce 1 MByte of network traffic, on average. Instead, JHE produced more traffic than JHD in all the object apps, with about 88 MBytes of generated traffic, on average.

As for $A10$, JHE had the minimum increment of Network Traffic Bytes over JHD of 2275 Bytes. This happened since the app exposed a Network Settings Gate GUI that required a valid and reachable IP address along with a valid Port number that were never provided without hybridization. However, even the amount of network traffic the app generated by unlocking this Gate GUI was still small since it consisted only in a few TCP or UDP network packets used to ping the specified ports.

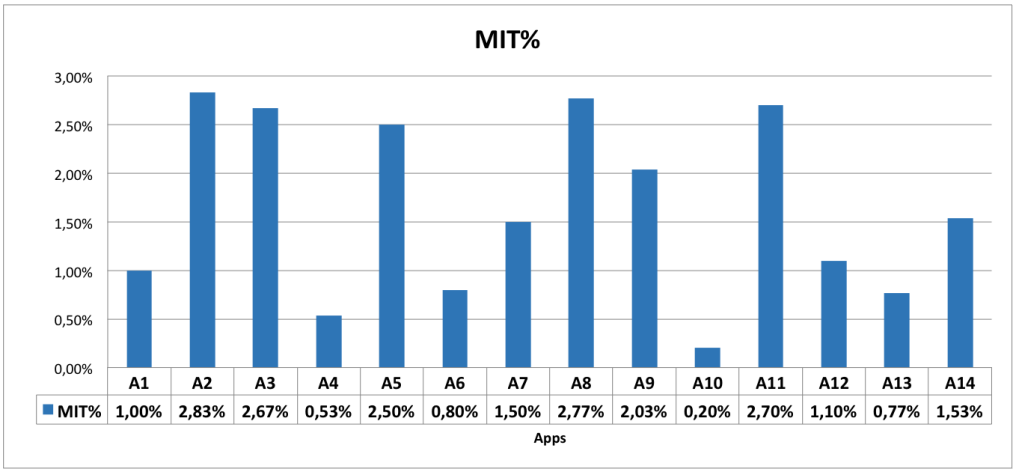
In the $A11$ app, i.e. LibreTorrent, it was measured the maximum NTB increment that consisted of over 667 MByte. This was obtained since the subjects unlocked the Network Settings Gate GUI providing valid Torrent URL that pointed to large files.

On the basis of these results it is possible to answer the first research questionRQ_1 and conclude that:

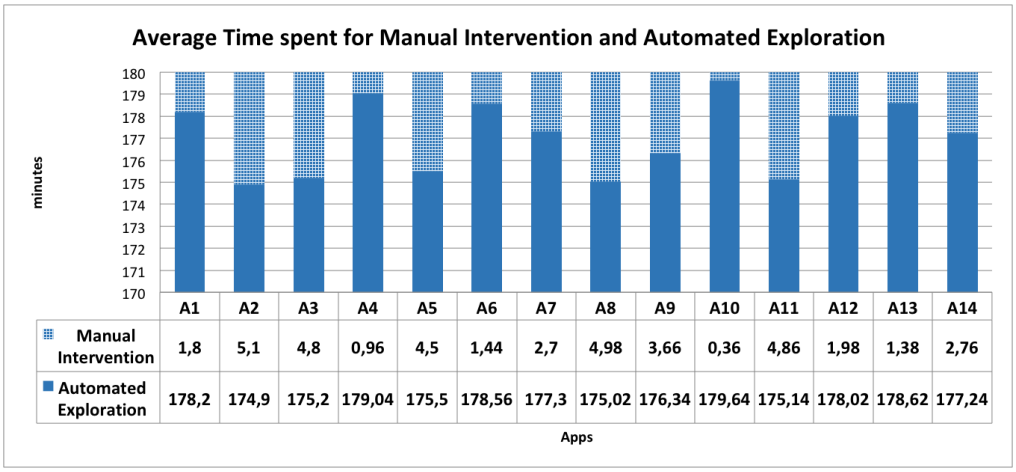
> *The hybridization introduced by juGULAR had a positive impact on the exploration effectiveness in both the considered scenarios. It allowed to obtain better results in terms of Covered Activities, Covered LOC and Generated Network Traffic.*

In order to address RQ_2, we considered the cost of the manual interventions required by the hybridization introduced by juGULAR. Figure 5.16(a) reports the average MIT% value for each app. The histogram and the table reported in Figure 5.16(b) show, for each app, the time spent for the manual intervention and for the automated exploration during the 180 minutes session time, averaged on all the subjects.

As Figure shows, the time for the manual intervention required by JHE was on average lower than 3% of the entire exploration time for all the considered apps. All the subjects were able to define the Unlocking Input Event Sequence in less than



(a) MIT%: Average percentage of time spent in the human interventions for unlocking the encountered Gate GUIs on the total exploration time, with the JHE approach in the 180 minutes sessions, averaged on all the subjects.



(b) Average Time (in minutes) spent for the Manual Intervention and for the Automated Exploration with the JHE approach in the 180 minutes sessions, averaged on all the subjects.

FIGURE 5.16: Costs of the manual interventions required by the hybridization introduced by juGULAR

about 5 minutes on average. For the *A*10 app, it took the subjects less than 40 seconds, on average, to define the Unlocking Input Event Sequence since the app exposed a simple form in which the subjects mostly inserted well-known IP addresses, e.g. the `localhost` or the Google DNS addresses.

As for *A*2 (Conversations), the subjects had to unlock a Login Gate GUI in which the user had to provide valid credentials of an account registered to an existing Jabber/XMPP service. Since most of the subjects did not own such an account, they spent time to create it before unlocking the Gate GUI.

According to these results, it is possible to answer the second research question RQ_2 concluding that:

> *The manual intervention required by juGULAR has a limited impact on the cost of the exploration technique, being always lower than 3%.*

In order to answer the RQ_3, we compared the effectiveness of JHE and Monkey. The data in Table 5.5 shows that JHE was always more effective than the Monkey tool. On average, JHE was able to cover 24% more Activities, 21% more LOCs and to generate 86 network traffic MBytes more than Monkey.

The reported data allow to answer the third research question RQ_3:

> *jUGULAR is more effective than the state-of-the-practice tool Monkey in exploring real Android apps.*

5.5.6 Study Conclusion

The experimental results showed that the hybridization of the automated exploration approach proposed by juGULAR produced in average a considerable improvement of the exploration effectiveness. They confirmed the usefulness of our approach that allows the user to provide knowledge at runtime rather than using pre-configured and generic input event sequences. This is consistent with other work that point out the complementarity between automated machine-generated tests and human tests [98, 99].

The fact that juGULAR was always able to outperform the other tools in terms of generated network traffic suggests that this approach may be especially useful in scenarios that leverage on realistic generated network traffic, such as ground-truth generation of mobile app traffic [100] or mobile app network traffic signatures generation [37].

5.5.7 Threats to validity

The following threats affect the validity of this experimental study [67].

5.5.7.1 Internal Validity

In this study, a possible threat to the internal validity could have been the assignment of the subjects to the objects. To mitigate the possible bias, the objects were randomly assigned to the subjects. A possible factor that could have influenced the outcome was the subject experience. To mitigate this threat, each app was explored multiple times with different subjects.

The outcome could be also influenced by the Gate GUI classes we considered. A further experimentation considering a wider sample of Gate GUI classes should be carried out to mitigate this threat.

Another possible threat to the internal validity is that the effectiveness improvements that we observed in the experiment were not actually due to the hybridization, but rather to other factors, such as the randomness of the explorations. We tried to mitigate this threat by executing 21 random explorations and involving 7 different subjects for each app and by performing the validation task of the Data Collection and Analysis step.

5.5.7.2 External Validity

I am aware that the choice of object apps is a possible threat to the external validity. The diversity of the selected apps can mitigate this threat. However, to extend the validity of our results, a wider sample of real Android apps including also industrial-strength apps from Google Play store should be considered.

Also choosing students as subjects of the study may have affected its external validity. However, they present characteristics that make them representative of possible future users of automated exploration techniques. To further mitigate this threat, case studies in real industrial settings should be carried out to assess the validity of the approach on the field.

In the study, we measured the manual intervention costs required by juGULAR by the MIT% metric. Since this metric depends on the total exploration time, the measured manual intervention costs are influenced by the choice of the app run length and our conclusions may not generalize beyond the considered experimental settings. We tried to mitigate this threat using in our experiments the value of the app run length that is adopted in state of the art works in Android app automated exploration. Indeed we set one hour as exploration time for each app run, following the experimental setup used in the previous thorough benchmark assessment study by Choudhary *et al.* [2] and in the experiment performed by Mao *et al.* [9].

I cannot claim that our results generalize to other Gate GUI classes. To further extend the validity of our study, an experiment involving a wider set of Gate GUI classifiers trained and integrated in the juGULAR platform should be carried out.

5.6 Related Work

5.6.1 Automated GUI Exploration Techniques for Android apps

A widely used automated GUI exploration tool for Android apps is Monkey, that is part of the Android SDK. This tool adopts a quite simple exploration approach by sending pseudo-random events to the app under test. It is mainly used for a quick and repeatable robustness testing of Android apps, revealing crashes, unhandled exceptions and Application Not Responding (ANR) errors. This tool is regarded as the current state-of-practice for automated Android app testing [84, 9], being the most widely used tool of this category in industrial settings [39, 101].

In recent years several smarter automated GUI exploration techniques for Android apps have been proposed in the literature, especially in the context of online testing [102]. Each technique adopts its own strategy to define input event sequences to explore the app behavior.

Amalfitano *et al.* [25] analyzed a set of 13 testing techniques implementing AGETs and abstracted, in a general framework, the characteristics of the different GUI exploration approaches.

Choudhary *et al.* [2] presented a comparative study of the main existing tool-supported test input generation techniques for Android, including 7 tools exploiting an AGET. They concluded that Monkey outperforms the considered tools; however, they highlighted that each tool shows perks that can be leveraged and combined in order to achieve significant overall improvements.

Zeng *et al.* [39] investigated the limitations of the Monkey tool in an industrial setting in which the apps can be far more complex than the open-source ones considered by Choudhary *et al.* [2]. One of the solutions they suggested to enhance the capabilities of Monkey consists in manually constructing and performing sequences of events based on the user knowledge when the app requires the user to login, provide valid address information or scan a valid QR code.

In the following, I describe the related work reporting their contribution and providing details about how and to what extent they dealt with exploration limitations related to the ones discussed in this Chapter. These contributions have been organized in 3 main groups on the basis of how they generate input event sequences that may be useful to interact with Gate GUIs.

5.6.2 AGETs that rely on predefined input event generation rules

A first group of AGETs leverages on predefined input event generation rules embedded in the technique such as textual input generation rules or rules to exercise specific GUI object types.

Karami *et al.* [12] proposed a software inspection framework for the identification of malicious apps. Like our approach, they exploit an AGET to send random sequences of GUI events to the app. It is able to generate significant input data for

text fields, by applying rules predefined in the tool based on the detected text field type. The text length can also be tuned by the user before the exploration. However, their automated input data generation strategy fails to unlock Gate GUIs that need app-specific knowledge. Instead, juGULAR leverages input event sequences provided by the user to unlock Gate GUIs.

A^{3}E [30] implements a model-based automated GUI exploration strategy for Android apps. Like our approach, it automatically detects Activities related to special responsibilities, such as login, using a rule-based classification approach. Instead, we adopt a Machine Learning approach since it does not require a strong expert involvement to define rules for each specific Gate GUI class. Their approach exercises these Activities with input events predefined in the tool. The authors have also raised the need for complex interactions to reproduce certain apps functionality. They have not addressed this limitation but planned to do it through Record and Replay as future work. Our approach successfully combines automated GUI exploration with Capture and Replay to exercise GUIs that require particular and complex input event sequences.

CrashScope [7] is a tool that explores Android apps using systematic input generation and exploiting several strategies with the aim of triggering crashes. It detects the type of text expected by an app field and automatically generates text input data to exercise it by applying rules predefined in the tool. Unlike our approach, these rules do not aim at exercising Gate GUIs to unlock them. Instead, CrashScope fills textual fields with expected and unexpected data inputs to trigger crashes due to input data not correctly handled in the code.

Sapienz [9] is a multi-objective search-based automated Android app exploratory testing approach; it is based on a preliminary exploration performed by an AGET. Like our approach, Sapienz adopts strategies to explore parts of the apps that can be reached only by exercising specific GUIs with particular and complex input event sequences. Its dynamic exploration technique exploits information retrieved by a static analysis of the app resources to fill the textual fields. Its authors addressed the need for complex interactions by using predefined patterns, referred to as *motif genes*, that capture testers' experience and allow to reach higher coverage when combined with atomic events. Our approach exploits neither static analysis nor predefined patterns to unlock Gate GUIs. Instead, we capture the human knowledge necessary to unlock a Gate GUI by recording input event sequences provided by the user at runtime.

The approaches belonging to this group may suffer from limitations in exercising Gate GUIs that need app-specific knowledge or contextual information that is available only at runtime and thus results hardly predictable before the app exploration.

5.6.3 Configurable AGETs that exploit input event sequences predefined by the user

The solutions belonging to this group also rely on input event sequences defined before the app exploration. But they allow the user himself to define ad-hoc rules in order to enhance the exploration by app-specific knowledge.

Amalfitano *et al.* [71], propose a configurable tool that implements both random and systematic GUI exploration strategies and has been exploited also for model-based testing [4]. Their work points out that the ability of the tool to cover the app source code and to discover faults depends on several factors including the timing between consecutive input events and input values provided to the GUI input fields. To this aim, the user can provide an ad-hoc and app-specific configuration of the tool before the exploration. Unlike AndroidRipper, our approach does not require human effort to preliminarily configure the exploration technique. Instead, juGULAR detects Gate GUIs during the app exploration by exploiting a Machine Learning approach, without any previous app-specific knowledge. Moreover, our approach unlock the Gate GUIs by using the input event sequences provided by the user during the exploration.

Choi *et al.* [72] designed an automated technique, named SwiftHand, that uses active learning to reconstruct a model of the app during testing. The AGET implemented by SwiftHand uses the learned app model in order to select at each iteration the next input event to be executed; it is chosen among the input events enabled at the current state. This technique can detect `EditText` GUI objects and fill them with significant input strings defined by the user before the exploration with the aim to improve the exploration. Also our approach aims at improving the app exploration. However, we do not need to predefine app-specific input and we are not limited to textual inputs. Instead, juGULAR exploits the input events provided by the user during the exploration. Moreover, we do not aim at detecting only `EditText` GUI objects, but we automatically detect GUIs that require to be exercised with particular input event sequences by exploiting a Machine Learning approach.

PUMA [86] is a programmable framework that can be exploited to dynamically analyze several app properties, such as correctness, performance and security. It provides a generic AGET that can be extensively configured to guide the app exploration. It can generate a textual input when it is needed according to a policy coded by the user. Moreover, the user can also specify app-specific events to be applied when the exploration reaches a *codepoint*, i.e. a precise point of the app binary. Also juGULAR uses app-specific events when the exploration reaches a certain state, *i.e.* a Gate GUI. However, our approach requires neither human effort to code ad-hoc policies nor the knowledge of the app binary to preliminarily configure the exploration technique. Instead, juGULAR detects Gate GUIs during the app exploration by exploiting a Machine Learning approach without any previous app-specific knowledge. Moreover, a juGULAR user does not have to code before the exploration the

app-specific events needed to exercise the detected Gate GUI. Instead, we leverage input event sequences provided by the user during the exploration.

Gang Hu *et al.* [6] proposed Appdoctor, a testing tool able to perform a quick exploration, called *approximate execution*. Their exploration strategy is faster than real execution since it exercises an app by invoking directly event handlers. Our technique instead triggers real events because they represent better real user interactions. Appdoctor presents a component for the generation of proper input for text fields, number pickers, lists and seekbars in order to improve code coverage. It detects the type of text expected by a text field and applies input data drawn from dictionaries predefined in the tool. Alternatively, it can exploit rules defined by the user before the exploration to generate the input for a specific GUI object. We also aim at improving the code coverage reached by the app exploration. Unlike Appdoctor, our approach does not need app-specific inputs defined by the user before the exploration. Instead, we exploit input event sequences provided by the manual user intervention during the exploration.

AndroGenerator [38] generates network traffic through automated exploration of Android apps. Its authors pointed out the limitations of their adopted technique since it is not able to provide right inputs to trigger the app code that generates network traffic. Therefore, their approach exploits also input event sequences defined by the user before the app exploration. Instead, in our approach input event sequences are provided during the exploration by the manual user intervention. Therefore, a juGULAR user does not need any previous app-specific knowledge.

The main limitation of these approaches is that they require programming skills to understand the app-specific GUI structure and/or configure the AGET to properly manage each distinct Gate GUI. These approaches are indeed human-intensive and may not extend to different applications.

5.6.4 AGETs exploiting manual user intervention

These AGETs combine automatically generated input event sequences with manual user intervention. In this way, they obtain human knowledge necessary to achieve a meaningful app exploration at runtime.

Dynodroid, proposed by Machiry *et al.* [8], is a system that generates relevant input sequences to Android apps. They clearly expressed the need to introduce human intelligence for exercising some app functionality that can not be exercised otherwise by an AGET. Like us, their technique allows the user to generate arbitrary events directly on the app UI. To this aim, a Dynodroid user must first stop manually the automated exploration. Instead, our approach can automatically stop the automated event generation when a Gate GUI is detected.

NetworkProfiler [87], is a tool that implements a technique for inferring fingerprints of Android apps from the traffic they generate. It allows to perform complex

input event sequences by fuzzing and replaying manual user traces captured before the exploration. Also juGULAR exploits Capture and Replay but, unlike NetworkProfiler, it captures manual user traces during the exploration.

Another work that combines automated GUI exploration with captured user event sequences through machine learning has been proposed by Ermuth and Pradel [40] in the field of Web apps testing. This work defines a macro-based test generation approach for client-side Web applications. It aims at augmenting automated test generation techniques with complex sequences of events that represent realistic user interactions. A *macro event* abstracts a single logical step that users commonly perform to interact with real apps. This approach exploits machine learning techniques to cluster multiple similar event sequences belonging to different recorded usage traces and to infer from them single macro events. Instead, in our work we use machine learning to achieve a different goal, i.e. to train a classifier to detect Gate GUIs by providing it GUIs belonging to real Android apps. Both approaches require recorded usage traces to augment the automated GUI exploration. However, juGULAR captures usage traces only when a Gate GUI is detected for the first time during the app exploration. Instead, the macro-based technique requires adequate sets of traces to be preliminary recorded for each analyzed app.

These approaches are promising since they obtain human knowledge directly from manual intervention without needing programming skills or ad-hoc tool configurations, but they still suffer from some limitations. In fact, a Dynodroid user has to be constantly involved in the automated exploration in order to recognize a Gate GUI and intervene to properly exercise it. The other approaches belonging to this group, instead, require adequate sets of manual traces that exploit app-specific knowledge and need to be recorded before the app exploration.

5.7 Conclusions and Future Work

Automated GUI exploration techniques are becoming widespread in mobile app development processes, due to their capability to execute time-consuming tasks. However, one of their critical issues is the limited capability of exploring the behavior of apps that require meaningful sequences of input events on specific GUIs, i.e. Gate GUIs, in order to exercise some of their functionality.

In this Chapter, I addressed this issue by proposing juGULAR, a hybrid automated GUI exploration technique I designed with my research group that pragmatically combines fully automated GUI exploration with Capture and Replay in order to improve the Android app exploration and minimize the human intervention. We leverage Machine Learning to train classifiers that are exploited by juGULAR to automatically detect the occurrence of Gate GUI instances during the exploration. In this work, we focused on 2 specific classes of Gate GUIs: *Login* and *Network Settings*.

Our technique has been implemented in a software platform and validated with an experiment involving 14 real Android apps. The experiment showed that the app

exploration can improve thanks to the hybridization in terms of Covered Activities, Covered LOC and Generated Network Traffic. The manual intervention required by the technique had a limited impact on the entire exploration costs. The experiment also showed that juGULAR was more effective in app exploration than the state-of-the practice automated Android GUI exploration tool.

I am aware that juGULAR may suffer from the limitation introduced by app non-determinisms. As future work, I and my research group intend to address the issues of non-deterministic Gate GUI, such as those exposed by Games or containing CAPTCHAs, by investigating effective solutions to handle them. We plan to extend juGULAR by considering more Gate GUI classes besides the ones we have dealt with.

Finally, we plan to extend the validity of our experimental results by carrying out an industrial case study involving real practitioners and a wider set of Android apps. In addition, we would like to consider further performance indicators, such as the diversity of generated network traffic. This aspect is critical for assessing how realistic such traffic is and it can be exploited in several areas, e.g. mobile app traffic ground-truth generation and network traffic signatures generation. To this aim, we intend to investigate suitable measurement approaches and metrics for evaluating such diversity, since it is still an open issue in the literature.

Chapter 6

Conclusions and Future Work

Mobile apps are today an essential component in the everyday life of billions of people. These apps should guarantee a high level of quality to meet the users expectations and thus be successful. Automation tools help to ease the burden of quality engineering activities on mobile developers. In particular, automated GUI exploration techniques are widely adopted by researchers and practitioners in the context of mobile apps for supporting critical engineering tasks such as reverse engineering, testing, and network traffic signature generation.

The Software Engineering community has been devoting a great effort to propose methodologies, approaches and tools for automatically exploring mobile apps, but there are challenges and issues still open. Therefore there is the need for new solutions that can be actually applied to improve the existing techniques and to adapt them to the specific characteristics of the mobile platforms. In this work I focus on Android, since it is today the world's most popular mobile operating system.

In this dissertation, I stressed the relevance of the issues exposed by Android apps when the lifecycle of their Activity components is exercised through mobile-specific events. In particular, I investigated the problem of GUI failures due to orientation change events and proposed a framework for detecting and classifying them. I explored the impact of such failures on both open-source and industrial-strength apps showing that more than 88% of the considered apps are affected by GUI failures, some classes of GUI failures are more common than others, and some GUI objects are more frequently involved. Almost all the failures detected by these studies were novel since they were not already reported in issue trackers. The set of collected GUI failures is available as open source and provides the largest currently available dataset of this kind of failures. It may be exploited by future work to evaluate and compare the effectiveness of different testing techniques and tools. I analyzed the source code of the apps affected by these failures and point out six classes of common faults that should be avoided by developers to improve the app quality. These Android-specific fault classes could be exploited in future work to develop new mutation operators for testing of Android apps and to define fault localization techniques focused on source code bugs that may cause the observed failures.

As another contribution of this work, I proposed ALARic, a fully-automated testing technique that adopts a GUI exploration strategy that systematically exercises

the lifecycle of app Activities to expose GUI failures and crashes. As future work, I intend to propose and implement a set of oracles able to detect other issues tied to the Activity lifecycle besides GUI failures and crashes, such as memory leaks and threading issues. Moreover, I plan to extend the ALARic approach to test the lifecycle of other app components, such as services, fragments and content providers.

Finally, I addressed the limitations introduced by classes of GUIs that may prevent the exploration of relevant parts of applications if they are not exercised with app-specific and complex input event sequences that only human knowledge can provide. In this dissertation, I referred to these GUIs as Gate GUIs.

I proposed juGULAR, a novel hybrid automated GUI exploration technique that pragmatically combines automated GUI exploration with Capture and Replay in order to effectively exercise Gate GUIs and minimize the human intervention. juGULAR can automatically detect the occurrences of Gate GUI instances during the exploration by exploiting a machine learning approach and exercise them by leveraging input event sequences provided by the user. In this work, I focused on two specific classes of Gate GUIs, i.e. Login and Network Settings, but I plan to extend juGULAR by considering additional Gate GUI classes. I am aware that juGULAR may suffer from the limitation introduced by app non-determinisms. As future work, I intend to address the issues of non-deterministic GUIs, such as those exposed by Games or containing CAPTCHAs, by investigating effective solutions to handle them.

The effectiveness of all the proposed solutions has been demonstrated through experimental evaluations performed on real mobile apps. To further extend the generalizability of the results, I plan to conduct a wider experimentation involving a larger set of Android apps and considering different Android platform versions and configurations of the tools.

The contributions described in this thesis target the Android Operating System, but can be extended to other software platforms. Although the process of testing with the specifics of the Activity lifecycle implemented in ALARic may seem rather specific to the Android environment and thus quite narrow, it can be seen as an instance of metamorphic testing that aims to alleviate the test oracle problem that is general in software testing. The app lifecycle management is a critical feature also in other mobile platforms. Therefore, I plan to extend the analysis and the solution presented in this work by considering other mobile operating systems, such as iOS and Windows 10 Mobile. Moreover, the combination of automated GUI exploration with capture and replay through machine learning implemented in juGULAR could be adapted to other GUI-based software platforms in addition to mobile ones, e.g. Web applications.

For all these reasons, I expect this thesis will provide a solid foundation for future research in the Android context and beyond. I also believe the proposed solutions can effectively aid researchers and developers with their mobile app analysis tasks.

Bibliography

[1] D. Amalfitano, A. R. Fasolino, P. Tramontana, and B. Robbins. "Testing Android Mobile Applications: Challenges, Strategies, and Approaches". In: *Advances in Computers* 89 (2013), pp. 1–52. DOI: 10.1016/B978-0-12-408094-2.00001-1.

[2] S. R. Choudhary, A. Gorla, and A. Orso. "Automated Test Input Generation for Android: Are We There Yet? (E)". In: *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. 2015, pp. 429–440. DOI: 10.1109/ASE.2015.89.

[3] S. Zein, N. Salleh, and J. Grundy. "A Systematic Mapping Study of Mobile Application Testing Techniques". In: *J. Syst. Softw.* 117.C (July 2016), pp. 334–356. ISSN: 0164-1212. DOI: 10.1016/j.jss.2016.03.065.

[4] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon. "MobiGUITAR: Automated Model-Based Testing of Mobile Apps". In: *IEEE Software* 32.5 (Sept. 2015), pp. 53–59. ISSN: 0740-7459. DOI: 10.1109/MS.2014.55.

[5] H. Zhu, X. Ye, X. Zhang, and K. Shen. "A Context-Aware Approach for Dynamic GUI Testing of Android Applications". In: *2015 IEEE 39th Annual Computer Software and Applications Conference*. Vol. 2. July 2015, pp. 248–253. DOI: 10.1109/COMPSAC.2015.77.

[6] G. Hu, X. Yuan, Y. Tang, and J. Yang. "Efficiently, Effectively Detecting Mobile App Bugs with AppDoctor". In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys '14. Amsterdam, The Netherlands: ACM, 2014, 18:1–18:15. ISBN: 978-1-4503-2704-6. DOI: 10.1145/2592798.2592813.

[7] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk. "Automatically Discovering, Reporting and Reproducing Android Application Crashes". In: *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. Apr. 2016, pp. 33–44. DOI: 10.1109/ICST.2016.34.

[8] A. Machiry, R. Tahiliani, and M. Naik. "Dynodroid: An Input Generation System for Android Apps". In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2013. Saint Petersburg, Russia: ACM, 2013, pp. 224–234. ISBN: 978-1-4503-2237-9. DOI: 10.1145/2491411.2491450.

[9] K. Mao, M. Harman, and Y. Jia. "Sapienz: Multi-objective Automated Testing for Android Applications". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ISSTA 2016. Saarbrücken, Germany: ACM, 2016, pp. 94–105. ISBN: 978-1-4503-4390-9. DOI: 10.1145/2931037.2931054.

[10] G. Canfora, F. Mercaldo, C. A. Visaggio, M. D'Angelo, A. Furno, and C. Manganelli. "A Case Study of Automating User Experience-Oriented Performance Testing on Smartphones". In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. Mar. 2013, pp. 66–69. DOI: 10.1109/ICST.2013.16.

[11] S. Mostafa, Xiaoyin X. Wang, and T. Xie. "PerfRanker: Prioritization of Performance Regression Tests for Collection-intensive Software". In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2017. Santa Barbara, CA, USA: ACM, 2017, pp. 23–34. ISBN: 978-1-4503-5076-1. DOI: 10.1145/3092703.3092725.

[12] M. Karami, M. Elsabagh, P. Najafiborazjani, and A. Stavrou. "Behavioral Analysis of Android Applications Using Automated Instrumentation". In: *2013 IEEE Seventh International Conference on Software Security and Reliability Companion*. June 2013, pp. 182–187. DOI: 10.1109/SERE-C.2013.35.

[13] S. N. Dutia, T. H. Oh, and Y. H. Oh. "Developing Automated Input Generator for Android Mobile Device to Evaluate Malware Behavior". In: *Proceedings of the 4th Annual ACM Conference on Research in Information Technology*. RIIT '15. Chicago, Illinois, USA: ACM, 2015, pp. 43–43. ISBN: 978-1-4503-3836-3. DOI: 10.1145/2808062.2808065.

[14] A. Avancini and M. Ceccato. "Security testing of the communication among Android applications". In: *Automation of Software Test (AST), 2013 8th International Workshop on*. San Francisco, California: IEEE Press, May 2013, pp. 57–63. ISBN: 978-1-4673-6161-3. DOI: 10.1109/IWAST.2013.6595792.

[15] G. Canfora, F. Mercaldo, and C. A. Visaggio. "An HMM and structural entropy based detector for Android malware: An empirical study". In: *Computers & Security* 61 (2016), pp. 1–18. DOI: 10.1016/j.cose.2016.04.009.

[16] S. Yang, D. Yan, and A. Rountev. "Testing for poor responsiveness in android applications". In: *Engineering of Mobile-Enabled Systems (MOBS), 2013 1st International Workshop on the*. May 2013, pp. 1–6. DOI: 10.1109/MOBS.2013.6614215.

[17] M. M. Eler, J. M. Rojas, Y. Ge, and G. Fraser. "Automated Accessibility Testing of Mobile Apps". In: *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. Apr. 2018, pp. 116–126. DOI: 10.1109/ICST.2018.00021.

[18] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury. "Detecting Energy Bugs and Hotspots in Mobile Apps". In: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: ACM, 2014, pp. 588–598. ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2635871.

[19] M. Linares-Vásquez, G. Bavota, C. E. Bernal Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk. "Optimizing Energy Consumption of GUIs in Android Apps: A Multi-objective Approach". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: ACM, 2015, pp. 143–154. ISBN: 978-1-4503-3675-8. DOI: 10.1145/2786805.2786847.

[20] R. Jabbarvand and S. Malek. "muDroid: An Energy-aware Mutation Testing Framework for Android". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: ACM, 2017, pp. 208–219. ISBN: 978-1-4503-5105-8. DOI: 10.1145/3106237.3106244.

[21] K. M. Chandy. "Event-driven applications: Costs, benefits and design approaches". In: *Gartner Application Integration and Web Services Summit* (2006).

[22] F. Belli, M. Beyazit, and A. Memon. "Testing is an Event-Centric Activity". In: *Software Security and Reliability Companion (SERE-C), 2012 IEEE Sixth International Conference on*. June 2012, pp. 198–206. DOI: 10.1109/SERE-C.2012.24.

[23] B. N. Nguyen and A. M. Memon. "An Observe-Model-Exercise* Paradigm to Test Event-Driven Systems with Undetermined Input Spaces". In: *IEEE Transactions on Software Engineering* 40.3 (Mar. 2014), pp. 216–234. ISSN: 0098-5589. DOI: 10.1109/TSE.2014.2300857.

[24] D. Amalfitano, N. Amatucci, A. R. Fasolino, and P. Tramontana. "A Conceptual Framework for the Comparison of Fully Automated GUI Testing Techniques". In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. Nov. 2015, pp. 50–57. DOI: 10.1109/ASEW.2015.19.

[25] D. Amalfitano, N. Amatucci, A. M. Memon, P. Tramontana, and A. R. Fasolino. "A general framework for comparing automatic testing techniques of Android mobile apps". In: *Journal of Systems and Software* 125 (2017), pp. 322–343. DOI: 10.1016/j.jss.2016.12.017.

[26] Z. Liu, X. Gao, and X. Long. "Adaptive random testing of mobile application". In: *2010 2nd International Conference on Computer Engineering and Technology*. Vol. 2. Apr. 2010, pp. V2–297–V2–301. DOI: 10.1109/ICCET.2010.5485442.

[27] C. Hu and I. Neamtiu. "Automating GUI Testing for Android Applications". In: *Proceedings of the 6th International Workshop on Automation of Software Test*. AST '11. Waikiki, HI, USA: ACM, 2011, pp. 77–83. ISBN: 978-1-4503-0592-1. DOI: 10.1145/1982595.1982612.

[28] I. C. Morgado and A. C. R. Paiva. "The iMPAcT Tool: Testing UI Patterns on Mobile Applications". In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Nov. 2015, pp. 876–881. DOI: 10.1109/ASE.2015.96.

[29] Y. Zhauniarovich, A. Philippov, O. Gadyatskaya, B. Crispo, and F. Massacci. "Towards Black Box Testing of Android Apps". In: *2015 10th International Conference on Availability, Reliability and Security*. Aug. 2015, pp. 501–510. DOI: 10.1109/ARES.2015.70.

[30] T. Azim and I. Neamtiu. "Targeted and Depth-first Exploration for Systematic Testing of Android Apps". In: *SIGPLAN Not.* 48.10 (Oct. 2013), pp. 641–660. ISSN: 0362-1340. DOI: 10.1145/2544173.2509549.

[31] W. Yang, M. R. Prasad, and T. Xie. "A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications". In: *Fundamental Approaches to Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 250–265. ISBN: 978-3-642-37057-1. DOI: 10.1007/978-3-642-37057-1_19.

[32] P. Wang, B. Liang, W. You, J. Li, and W. Shi. "Automatic Android GUI Traversal with High Coverage". In: *2014 Fourth International Conference on Communication Systems and Network Technologies* (Apr. 2014), pp. 1161–1166. DOI: 10.1109/CSNT.2014.236.

[33] P. Maiya, A. Kanade, and R. Majumdar. "Race Detection for Android Applications". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '14. Edinburgh, United Kingdom: ACM, 2014, pp. 316–325. ISBN: 978-1-4503-2784-8. DOI: 10.1145/2594291.2594311.

[34] H. Wen, C. Lin, T. Hsieh, and C. Yang. "PATS: A Parallel GUI Testing Framework for Android Applications". In: *2015 IEEE 39th Annual Computer Software and Applications Conference*. Vol. 2. June 2015, pp. 210–215. DOI: 10.1109/COMPSAC.2015.80.

[35] D. Amalfitano, V. Riccio, N. Amatucci, V. De Simone, and A. R. Fasolino. "Combining Automated GUI Exploration of Android apps with Capture and Replay through Machine Learning". In: *Information and Software Technology* 105.1 (2019). ISSN: 0950-5849. DOI: 10.1016/j.infsof.2018.08.007.

[36] A. Memon, I. Banerjee, B. N. Nguyen, and B. Robbins. "The first decade of GUI ripping: Extensions, applications, and broader impacts". In: *2013 20th Working Conference on Reverse Engineering (WCRE)*. Oct. 2013, pp. 11–20. DOI: 10.1109/WCRE.2013.6671275.

[37] Y. Chen, W. You, Y. Lee, K. Chen, X. Wang, and W. Zou. "Mass Discovery of Android Traffic Imprints Through Instantiated Partial Execution". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: ACM, 2017, pp. 815–828. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134009.

[38] X. Su, D. Zhang, W. Li, and X. Wang. "AndroGenerator: An automated and configurable android app network traffic generation system". In: *Security and Communication Networks* 8.18 (2015). sec.1341, pp. 4273–4288. ISSN: 1939-0122. DOI: 10.1002/sec.1341.

[39] X. Zeng et al. "Automated Test Input Generation for Android: Are We Really There Yet in an Industrial Case?" In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. Seattle, WA, USA: ACM, 2016, pp. 987–992. ISBN: 978-1-4503-4218-6. DOI: 10.1145/2950290.2983958.

[40] M. Ermuth and M. Pradel. "Monkey See, Monkey Do: Effective Generation of GUI Tests with Inferred Macro Events". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ISSTA 2016. Saarbrucken, Germany: ACM, 2016, pp. 82–93. ISBN: 978-1-4503-4390-9. DOI: 10.1145/2931037.2931053.

[41] H. Muccini, A. Di Francesco, and P. Esposito. "Software testing of mobile applications: Challenges and future research directions". In: *Automation of Software Test (AST), 2012 7th International Workshop on*. Zurich, Switzerland: IEEE, 2012, pp. 29–35. ISBN: 978-1-4673-1821-1. DOI: 10.1109/IWAST.2012.6228987.

[42] R. N. Zaeem, M. R. Prasad, and S. Khurshid. "Automated Generation of Oracles for Testing User-Interaction Features of Mobile Apps". In: *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. Mar. 2014, pp. 183–192. DOI: 10.1109/ICST.2014.31.

[43] C. Q. Adamsen, G. Mezzetti, and A. Møller. "Systematic Execution of Android Test Suites in Adverse Conditions". In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. Baltimore, MD, USA: ACM, 2015, pp. 83–93. ISBN: 978-1-4503-3620-8. DOI: 10.1145/2771783.2771786.

[44] I. C. Morgado and A. C. R. Paiva. "Impact of Execution Modes on Finding Android Failures". In: *Procedia Computer Science* 83 (2016). The 7th International Conference on Ambient Systems, Networks and Technologies (ANT 2016) / The 6th International Conference on Sustainable Energy Information Technology (SEIT-2016) / Affiliated Workshops, pp. 284 –291. ISSN: 1877-0509. DOI: 10.1016/j.procs.2016.04.127.

[45] Z. Shan, T. Azim, and I. Neamtiu. "Finding Resume and Restart Errors in Android Applications". In: *SIGPLAN Not.* 51.10 (Oct. 2016), pp. 864–880. ISSN: 0362-1340. DOI: 10.1145/2983990.2984011.

[46] M. Jun, L. Sheng, Y. Shengtao, T. Xianping, and L. Jian. "LeakDAF: An Automated Tool for Detecting Leaked Activities and Fragments of Android Applications". In: *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 1. July 2017, pp. 23–32. DOI: 10.1109/COMPSAC.2017.161.

[47] Simone Graziussi. *Lifecycle and Event-Based Testing for Android Applications*. School Of Industrial Engineering and Information, Politecnico di Milano, 2016. URL: https://www.politesi.polimi.it/handle/10589/123981.

[48] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, and G. Pu. "Efficiently Manifesting Asynchronous Programming Errors in Android Apps". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE 2018. Montpellier, France: ACM, 2018, pp. 486–497. ISBN: 978-1-4503-5937-5. DOI: 10.1145/3238147.3238170.

[49] L. Fan et al. "Large-scale Analysis of Framework-specific Exceptions in Android Apps". In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE '18. Gothenburg, Sweden: ACM, 2018, pp. 408–419. ISBN: 978-1-4503-5638-1. DOI: 10.1145/3180155.3180222.

[50] U. Farooq and Z. Zhao. "RuntimeDroid: Restarting-Free Runtime Change Handling for Android Apps". In: *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. MobiSys '18. Munich, Germany: ACM, 2018, pp. 110–122. ISBN: 978-1-4503-5720-3. DOI: 10.1145/3210240.3210327.

[51] A. Issa, J. Sillito, and V. Garousi. "Visual testing of Graphical User Interfaces: An exploratory study towards systematic definitions and approaches". In: *2012 14th IEEE International Symposium on Web Systems Evolution (WSE)*. Sept. 2012, pp. 11–15. DOI: 10.1109/WSE.2012.6320526.

[52] V. Lelli, A. Blouin, and B. Baudry. "Classifying and Qualifying GUI Defects". In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. Apr. 2015, pp. 1–10. DOI: 10.1109/ICST.2015.7102582.

[53] D. Amalfitano, V. Riccio, A. C. R. Paiva, and A. R. Fasolino. "Why does the orientation change mess up my Android application? From GUI failures to code faults". In: *Softw. Test., Verif. Reliab.* 28.1 (2018). DOI: 10.1002/stvr.1654.

[54] V. Riccio, D. Amalfitano, and A. R. Fasolino. "Is This the Lifecycle We Really Want? An Automated Black-Box Testing Approach for Android Activities". In: *ECOOP/ISSTA Workshops, 2018, Amsterdam, the Netherlands, July 19, 2018*. New York, NY, USA: ACM, 2018.

[55] A. Memon, I. Banerjee, and A. Nagarajan. "GUI ripping: reverse engineering of graphical user interfaces for testing". In: *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings.* Nov. 2003, pp. 260–269. DOI: 10.1109/WCRE.2003.1287256.

[56] A. C. R. Paiva, J. C. P. Faria, and P. M. C. Mendes. "Reverse Engineered Formal Models for GUI Testing". In: *Formal Methods for Industrial Critical Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 218–233. ISBN: 978-3-540-79707-4. DOI: 10.1007/978-3-540-79707-4_16.

[57] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Mauro Santoro. "Automatic testing of GUI-based applications". In: *Software Testing, Verification and Reliability* 24.5 (), pp. 341–366. DOI: 10.1002/stvr.1538.

[58] A. Mesbah, A. van Deursen, and S. Lenselink. "Crawling Ajax-Based Web Applications Through Dynamic Analysis of User Interface State Changes". In: *ACM Trans. Web* 6.1 (Mar. 2012), 3:1–3:30. ISSN: 1559-1131. DOI: 10.1145/2109205.2109208.

[59] B. N. Nguyen, B. Robbins, I. Banerjee, and A. M. Memon. "GUITAR: An Innovative Tool for Automated Testing of GUI-driven Software". In: *Automated Software Engg.* 21.1 (Mar. 2014), pp. 65–105. ISSN: 0928-8910. DOI: 10.1007/s10515-013-0128-9.

[60] Y. M. Baek and D. H. Bae. "Automated model-based Android GUI testing using multi-level GUI comparison criteria". In: *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Sept. 2016, pp. 238–249. DOI: 10.1145/2970276.2970313.

[61] D. Amalfitano, N. Amatucci, A. R. Fasolino, P. Tramontana, E. Kowalczyk, and A. M. Memon. "Exploiting the Saturation Effect in Automatic Random Testing of Android Applications". In: *2nd ACM International Conference on Mobile Software Engineering and Systems, MOBILESoft 2015, Florence, Italy, May 16-17, 2015*. 2015, pp. 33–43. DOI: 10.1109/MobileSoft.2015.11.

[62] L. Deng, J. Offutt, P. Ammann, and N. Mirzaei. "Mutation operators for testing Android apps". In: *Information & Software Technology* 81 (2017), pp. 154–168. DOI: 10.1016/j.infsof.2016.04.012.

[63] "IEEE Standard Classification for Software Anomalies". In: *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)* (Jan. 2010), pp. 1–23. DOI: 10.1109/IEEESTD.2010.5399061.

[64] K. Holl and F. Elberzhager. "Mobile Application Quality Assurance: Reading Scenarios as Inspection and Testing Support". In: *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)* (2016), pp. 245–249. ISSN: 2376-9505. DOI: 10.1109/SEAA.2016.11.

[65] Z. Pingyu and S. G. Elbaum. "Amplifying Tests to Validate Exception Handling Code: An Extended Study in the Mobile Application Domain". In: *ACM Trans. Softw. Eng. Methodol.* 23.4 (2014), 32:1–32:28. DOI: 10.1145/2652483.

[66] D. A. Norman. *The Design of Everyday Things*. New York, NY, USA: Basic Books, Inc., 2002. ISBN: 9780465067107.

[67] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, and B. Regnell. *Experimentation in Software Engineering*. Springer, 2012. ISBN: 978-3-642-29043-5. DOI: 10.1007/978-3-642-29044-2.

[68] R. K. Yin. *Case Study Research: Design and Methods*. Applied Social Research Methods. SAGE Publications, 2009. ISBN: 9781412960991.

[69] S. Anand, M. Naik, M. J. Harrold, and H. Yang. "Automated Concolic Testing of Smartphone Apps". In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. FSE '12. Cary, North Carolina: ACM, 2012, 59:1–59:11. ISBN: 978-1-4503-1614-9. DOI: 10.1145/2393596.2393666.

[70] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood. "Testing Android Apps Through Symbolic Execution". In: *SIGSOFT Softw. Eng. Notes* 37.6 (Nov. 2012), pp. 1–5. ISSN: 0163-5948. DOI: 10.1145/2382756.2382798.

[71] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. "Using GUI Ripping for Automated Testing of Android Applications". In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ASE 2012. Essen, Germany: ACM, 2012, pp. 258–261. ISBN: 978-1-4503-1204-2. DOI: 10.1145/2351676.2351717.

[72] W. Choi, G. Necula, and K. Sen. "Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning". In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '13. Indianapolis, Indiana, USA: ACM, 2013, pp. 623–640. ISBN: 978-1-4503-2374-1. DOI: 10.1145/2509136.2509552.

[73] M. Cunha, A. C. R. Paiva, H. S. Ferreira, and R. Abreu. "PETTool: A pattern-based GUI testing tool". In: *Software Technology and Engineering (ICSTE), 2010 2nd International Conference on*. Vol. 1. San Juan, PR: IEEE, 2010, pp. V1–202 –VI–206. ISBN: 9781424486663. DOI: 10.1109/ICSTE.2010.5608882.

[74] R. M. L. M. Moreira, A. C. R. Paiva, and A. Memon. "A pattern-based approach for GUI modeling and testing". In: *IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013, Pasadena, CA, USA, November 4-7, 2013*. 2013, pp. 288–297. DOI: 10.1109/ISSRE.2013.6698881.

[75] M. Nabuco and A. C. R. Paiva. "Model-Based Test Case Generation for Web Applications". In: *14th International Conference on Computational Science and Applications (ICCSA 2014)*. 2014.

[76] C. S. Jensen, M. R. Prasad, and A. Møller. "Automated Testing with Targeted Event Sequence Generation". In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ISSTA 2013. Lugano, Switzerland: ACM, 2013, pp. 67–77. ISBN: 978-1-4503-2159-4. DOI: 10.1145/2483760.2483777.

[77] C. D. Nguyen, A. Marchetto, and P. Tonella. "Combining Model-based and Combinatorial Testing for Effective Test Case Generation". In: *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ISSTA 2012. Minneapolis, MN, USA: ACM, 2012, pp. 100–110. ISBN: 978-1-4503-1454-1. DOI: 10.1145/2338965.2336765.

[78] A. Banerjee, G. Hai-Feng, and A. Roychoudhury. "Debugging energy-efficiency related field failures in mobile apps". In: *Proceedings of the International Conference on Mobile Software Engineering and Systems, MOBILESoft '16, Austin, Texas, USA, May 14-22, 2016*. 2016, pp. 127–138. DOI: 10.1145/2897073.2897085.

[79] A. Banerjee, L. K. Chong, C. Ballabriga, and A. Roychoudhury. "EnergyPatch: Repairing Resource Leaks to Improve Energy-Efficiency of Android Apps". In: *IEEE Transactions on Software Engineering* 44.5 (May 2018), pp. 470–490. ISSN: 0098-5589. DOI: 10.1109/TSE.2017.2689012.

[80] L. Deng, N. Mirzaei, P. Ammann, and J. Offutt. "Towards mutation analysis of Android apps". In: *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. Apr. 2015, pp. 1–10. DOI: 10.1109/ICSTW.2015.7107450.

[81] D. Franke, S. Kowalewski, C. Weise, and N. Prakobkosol. "Testing Conformance of Life Cycle Dependent Properties of Mobile Applications". In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. Apr. 2012, pp. 241–250. DOI: 10.1109/ICST.2012.104.

[82] A. Méndez-Porras, G. Méndez-Marín, A. Tablada-Rojas, M. Nieto Hidalgo, J. M. García-Chamizo, M. Jenkins, and A. Martínez. "A distributed bug analyzer based on user-interaction features for mobile apps". In: *Journal of Ambient Intelligence and Humanized Computing* 8.4 (Aug. 2017), pp. 579–591. ISSN: 1868-5145. DOI: 10.1007/s12652-016-0435-7.

[83] TY. Chen, SC. Cheung, and SM. Yiu. *Metamorphic testing: a new approach for generating next test cases*. Tech. rep. HKUST-CS98-01, Department of Computer Science, Hong Kong, 1998.

[84] R. Mahmood, N. Mirzaei, and S. Malek. "EvoDroid: Segmented Evolutionary Testing of Android Apps". In: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: ACM, 2014, pp. 599–609. ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2635896.

[85] H. Zheng et al. "Automated Test Input Generation for Android: Towards Getting There in an Industrial Case". In: *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*. ICSE-SEIP '17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 253–262. ISBN: 978-1-5386-2717-4. DOI: 10.1109/ICSE-SEIP.2017.32.

[86] S. Hao, B. Liu, S. Nath, W. G. J. Halfond, and R. Govindan. "PUMA: Programmable UI-automation for Large-scale Dynamic Analysis of Mobile Apps". In: *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*. MobiSys '14. Bretton Woods, New Hampshire, USA: ACM, 2014, pp. 204–217. ISBN: 978-1-4503-2793-0. DOI: 10.1145/2594368.2594390.

[87] S. Dai, A. Tongaonkar, X. Wang, A. Nucci, and D. Song. "NetworkProfiler: Towards automatic fingerprinting of Android apps". In: *2013 Proceedings IEEE INFOCOM*. Apr. 2013, pp. 809–817. DOI: 10.1109/INFCOM.2013.6566868.

[88] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. "RERAN: Timing- and Touch-sensitive Record and Replay for Android". In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. San Francisco, CA, USA: IEEE Press, May 2013, pp. 72–81. ISBN: 978-1-4673-3076-3. DOI: 10.1109/ICSE.2013.6606553.

[89] K. P. Seng, L. Ang, and C. S. Ooi. "A Combined Rule-Based amp;amp; Machine Learning Audio-Visual Emotion Recognition Approach". In: *IEEE Transactions on Affective Computing* 9.1 (Jan. 2018), pp. 3–13. ISSN: 1949-3045. DOI: 10.1109/TAFFC.2016.2588488.

[90] G. A. di Lucca, A. R. Fasolino, and P. Tramontana. "Web Pages Classification using Concept Analysis". In: *2007 IEEE International Conference on Software Maintenance*. Oct. 2007, pp. 385–394. DOI: 10.1109/ICSM.2007.4362651.

[91] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. New York, NY, USA: Cambridge University Press, 2008. ISBN: 0521865719, 9780521865715.

[92] I. Banerjee, B. N. Nguyen, V. Garousi, and A. M. Memon. "Graphical user interface (GUI) testing: Systematic mapping and repository". In: *Information and Software Technology* 55.10 (2013), pp. 1679 –1694. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2013.03.004.

[93] S. Panichella, A. Di Sorbo, E. Guzman, C. A. Visaggio, G. Canfora, and H. C. Gall. "ARdoc: App Reviews Development Oriented Classifier". In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. Seattle, WA, USA: ACM, 2016, pp. 1023–1027. ISBN: 978-1-4503-4218-6. DOI: 10.1145/2950290.2983938.

[94] M. Galar, A. Alberto, E. Barrenechea, H. Bustince, and F. Herrera. "An overview of ensemble methods for binary classifiers in multi-class problems: Experimental study on one-vs-one and one-vs-all schemes". In: *Pattern Recognition* 44.8 (2011), pp. 1761 –1776. ISSN: 0031-3203. DOI: 10.1016/j.patcog.2011.01.017.

[95] N. Garcia-Pedrajas and D. Ortiz-Boyer. "An empirical study of binary classifier fusion methods for multiclass classification". In: *Information Fusion* 12.2 (2011), pp. 111 –130. ISSN: 1566-2535. DOI: 10.1016/j.inffus.2010.06.010.

[96] J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques*. 3rd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 0123814790, 9780123814791.

[97] D. Adamo, D. Nurmuradov, S. Piparia, and R. Bryce. "Combinatorial-based event sequence testing of Android applications". In: *Information and Software Technology* 99 (2018), pp. 98 –117. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2018.03.007.

[98] K. Mao, M. Harman, and Y. Jia. "Crowd Intelligence Enhances Automated Mobile Testing". In: *Proceedings of the 2017 32th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Oct. 2017, pp. 16–26. DOI: 10.1109/ASE.2017.8115614.

[99] M. Linares-Vásquez, M. White, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk. "Mining Android App Usages for Generating Actionable GUI-Based Execution Scenarios". In: *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. Apr. 2015, pp. 111–122. DOI: 10.1109/MSR.2015.18.

[100] V. F. Taylor, R. Spolaor, M. Conti, and I. Martinovic. "AppScanner: Automatic Fingerprinting of Smartphone Apps from Encrypted Network Traffic". In: *2016 IEEE European Symposium on Security and Privacy (EuroS P)*. Mar. 2016, pp. 439–454. DOI: 10.1109/EuroSP.2016.40.

[101] P. Liu, X. Zhang, M. Pistoia, Y. Zheng, M. Marques, and L. Zeng. "Automatic Text Input Generation for Mobile Testing". In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. May 2017, pp. 643–653. DOI: 10.1109/ICSE.2017.65.

[102] M. Utting, A. Pretschner, and B. Legeard. "A Taxonomy of Model-based Testing Approaches". In: *Softw. Test. Verif. Reliab.* 22.5 (Aug. 2012), pp. 297–312. ISSN: 0960-0833. DOI: 10.1002/stvr.456.