


RESEARCH ARTICLE

Why does the orientation change mess up my Android application? From GUI failures to code faults

Domenico Amalfitano¹ | Vincenzo Riccio¹ | Ana C. R. Paiva² | Anna Rita Fasolino¹ 

¹Department of Electrical Engineering and Information Technologies, University of Naples Federico II, Via Claudio 21 Naples, Italy

²Faculty of Engineering of the University of Porto, Porto, Portugal

Correspondence

Anna Rita Fasolino, Department of Electrical Engineering and Information Technologies, University of Naples Federico II, Via Claudio 21, Naples, Italy.

Email: anna.fasolino@unina.it

Summary

This paper investigates the failures exposed in mobile apps by the mobile-specific event of changing the screen orientation. We focus on GUI failures resulting in unexpected GUI states that should be avoided to improve the apps quality and to ensure better user experience. We propose a classification framework that distinguishes 3 main classes of GUI failures due to orientation changes and exploit it in 2 studies that investigate the impact of such failures in Android apps. The studies involved both open-source and apps from Google Play 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 us to identify 6 classes of common faults causing specific GUI failures.

KEYWORDS

Android bugs, Android testing, GUI failures, GUI testing, mobile testing, orientation change

1 | INTRODUCTION

Over the last decade, the number of users of mobile technology and smartphones has increased considerably. The total number of smartphone users worldwide is forecast to surpass 2.5 billion in 2019 [1].

This causes a constant demand for new software applications running on these devices (*apps*). As of the month of June 2016, both Android and iOS users had the opportunity to choose from among more than 2 million apps [2].

Mobile technology has radically changed the lifestyle of billions of people around the world. We use mobile apps for several hours every day, entrust them our sensitive data, and perform a large variety of activities through them, including critical tasks.

The demand for app quality has grown together with their spread. Apps users require them to be reliable, robust, efficient, secure, usable, etc. As a consequence, software developers should give proper consideration to the quality of their applications by adopting suitable quality assurance techniques, such as testing.

In the last decade, the research community has devoted great interest to the mobile app testing field. Several testing approaches have been proposed to assess different quality aspects of mobile applications [3], such as functionality [4], performance [5], security [6, 7], responsiveness [8], and energy consumption [9].

Since mobile apps are event-driven systems, many proposed techniques solicit them by means of sequences of events [10]. However, because of the peculiarities of the mobile devices, these apps should be tested with appositely crafted approaches [11]. As an example, testing processes should devote particular attention to exercise the apps through mobile-specific events, such as sending an application to the background and resuming it, receiving a call, changing the state of the network connections, or changing the orientation of the device.

Among these types of events, the orientation change deserves special attention. It is a peculiar event in mobile platforms that causes the switch of the running app between portrait and landscape layout configurations. Android guidelines recommend that, when the orientation change event occurs, the application 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

*<https://developer.android.com/guide/components/activities/state-changes.html>

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, 12, 13, 14, 15].

The GUI failures are a relevant class of failures that may disrupt the user experience. They consist in the manifestation of an unexpected GUI state, according to Lelli et al [16]. If an Android app does not correctly handle an orientation change event, it is likely that this event results in different types of GUI failures. As an example, 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 errors 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 paper, we propose a framework for classifying GUI failures and exploit it in 2 different exploratory studies that aimed at investigating their diffusion, the key characteristics, and possible faults causing them. The former study addressed the context of open-source Android apps, while the latter one considered very popular Android apps from Google Play Store. In both studies, the apps were tested and a considerable number of GUI failures due to the orientation change was detected. These failures were validated and classified and made available in public-shared documents. In the former study, we also analyzed the source code of a subset of applications exposing most frequent types of failure and were able to discover 6 classes of common faults causing them made by Android programmers. The results of both studies are presented in the paper.

Our paper contributes the Android community in several ways. It may help in the definition of a fault model specific to Android apps 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 [17]. 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 our studies may be exploited to evaluate and compare the effectiveness of different testing techniques and tools.

The remainder of the paper is structured as follows. Section 2 presents some examples of real GUI failures that motivated us to explore this issue.

Section 3 illustrates the framework we defined for characterizing GUI failures due to orientation changes.

Section 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 5 reports a second exploratory study that aimed at finding and classifying DOC GUI failure in top Android apps from Google Play Store. Section 6 discusses the threats that could affect the validity of the exploratory studies.

Section 7 provides related work. Section 8 finally draws the conclusions and presents future work.

2 | MOTIVATING EXAMPLES

In this section, we present 4 examples of GUI failures due to screen orientation changes in mobile apps. We found these failures by manually testing 3 different real mobile applications. We solicited the apps by using the double orientation change (DOC) event that consists in a sequence of 2



FIGURE 1 Example of Windows 10 GUI failure

consecutive orientation change events. To detect a GUI failure, we compared the GUI before and after this event. We used the DOC event because we observed that applying 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 [18], 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, we have a GUI failure [12, 15].

The first example of GUI failure occurs in the digital note-taking application OneNote running on Windows 10 Mobile OS. Figure 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

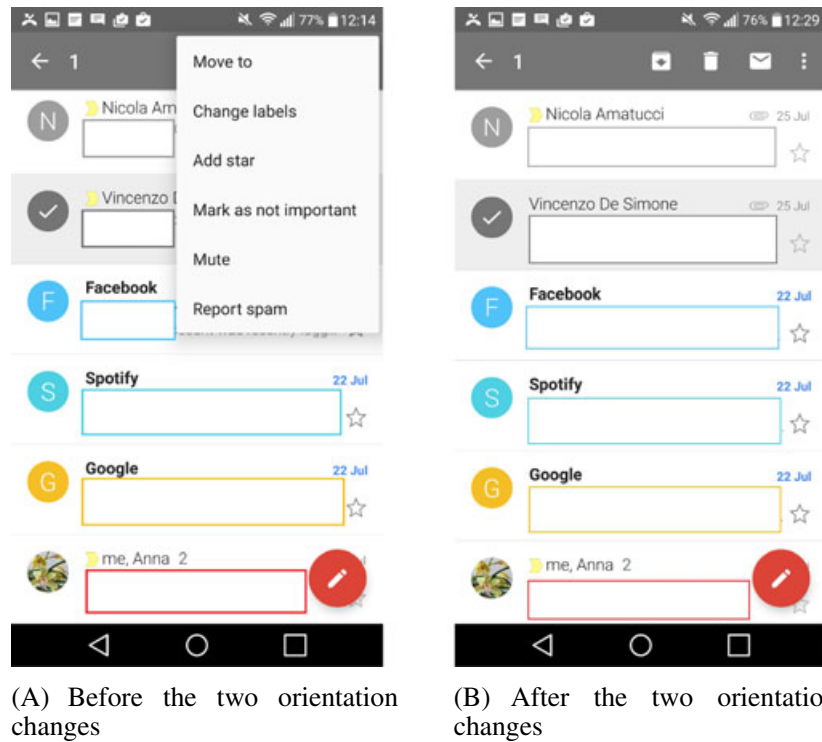


FIGURE 2 Android Gmail GUI failure

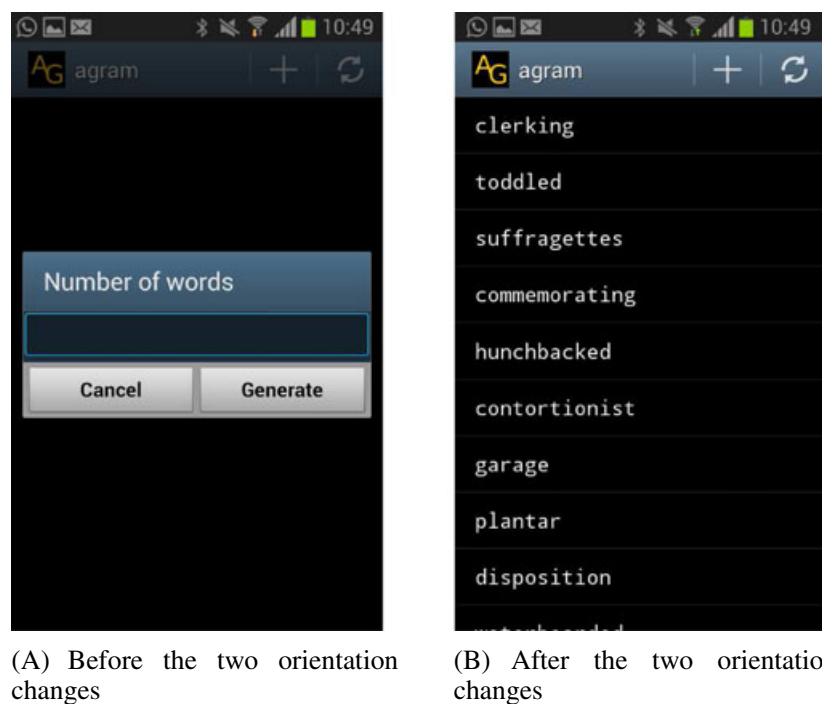
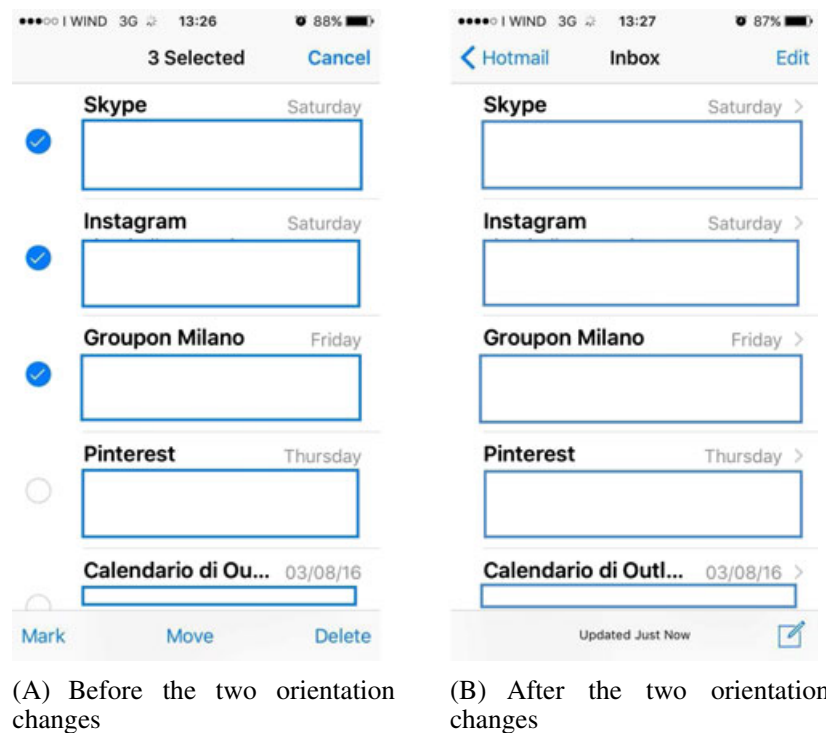


FIGURE 3 GUI fault exposed by Android Agram



(A) Before the two orientation changes

(B) After the two orientation changes

FIGURE 4 Example of iOS GUI failure

performed on the selected document such as syncing it, as shown in Figure 1A. After the DOC, the contextual menu disappears as reported in Figure 1B. 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 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 2A. After a DOC of the device, the menu disappears from the user interface, as shown in Figure 2B.

These failures also occur in lesser known apps such as Agram, an Android application that displays anagrams in English. Figure 3 shows the failure appearing in Agram version 1.4.1 that is in execution on a device equipped with Android 6.0. If the user chooses to create random words, a Dialog appears prompting the number of words he wants to generate (see Figure 3A). 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 3B.

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 4. The user can select one or more incoming messages he wants to manage, as reported in Figure 4A. After the DOC of the device, the application does not preserve the mail selection made by the user, as shown in Figure 4B. As a consequence, the UI widgets allowing to handle the selected mails (ie, Cancel, Mark, Move, and Delete) disappear and different ones appear on the user interface.

As these examples show, the observed failures concerned apps from all the major mobile platforms, ie, Android, iOS, and Windows. They affected even best selling applications and applications usually bundled in mobile devices as preinstalled software. Even if such failures are not so serious as crashes of the applications, their effects may have a negative impact on the user experience and contribute to nonpositive user ratings. These examples suggested us that this problem may be relevant and widespread and worth to be further investigated.

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 [16]. When we test an app by 2 consecutive orientation changes (or 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 previous examples showed, GUI failures may appear in different ways: They 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. The *mode* attribute indicates how the failure manifested itself in the GUI end state.

More precisely, the *scope* attribute can assume the values of one of the types of GUI objects used for implementing the GUI of the considered app in the considered mobile platform P . As an example, the types of GUI objects used for implementing GUIs in the Android platform include Button, ContextMenu, Dialog, and TextView. More in general, we say $S(P)$ 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 [19] and other GUI failure classification models proposed in the literature [20], this attribute can assume 1 of the 3 values of *Extra*, *Missing*, and *Wrong*. These values are defined as follows:

- **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 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.

Figures 2 and 3 show 2 examples of DOC GUI failures observed in the Android context. Figure 2 provides another example of Missing mode GUI failure involving an ActionOverflowMenu object. The 2 GUIs differ for one single object.

Figure 3 presents a more complex case where the same DOC event triggered 2 distinct failures, ie, 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 in the former GUI.

Figure 4 provides an example of 3 DOC GUI failures observed in the iOS context. According with our classification framework, these failures are characterized by the couple (Wrong, UITableView), since the *selected* property of the items in the UITableView changes state after the DOC.

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

3.1 | GUI Objects

The 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, and background color, which vary with the type of the considered object. Each 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 operating platform.

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

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

If needed, we use the dot notation for referring to the properties' value of a given object, ie, the notation $o.p_i$ indicates the value v_j assumed by the property p_i of the object o .

Among the set of properties of an object, we need to focus on a subset that is critical for identifying it and defining its layout, ie, 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. We define P^* as the set of *fundamental object properties*.

Definition 4. $P^* \triangleq \{type, position, size\} \subset P$

3.2 | GUI state and state transition

We formally describe the GUI state as the set of GUI objects that it contains. In mobile applications, like in any other event based software system, single events or event sequences may cause state transitions. We define the transition between 2 states, due to the triggering of one or more events, as a function that associates 2 states.

The DOC is a sequence of 2 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. $S \triangleq \{o_1, o_2, \dots, o_n\}$.

Definition 6. Given an Application Under Test (AUT), the set of Application States of AUT (AS_{AUT}) is defined by all the GUI States the AUT can render to the user: $AS_{AUT} \triangleq \{S_1, S_2, \dots, S_n\}$.

Definition 7. An event e is a function that associates 2 GUI states of an AUT. $e : AS_{AUT} \rightarrow AS_{AUT}$.

Definition 8. An event sequence es is a predefined ordered sequence $es = \langle e_1, \dots, e_n \rangle$ of n events, $n \geq 1$, that must be sequentially triggered starting from an initial GUI state of the AUT for reaching a final state of the AUT. Formally, an event sequence is a function that associates 2 GUI states of an AUT. $es : AS_{AUT} \rightarrow AS_{AUT}$.

Definition 9. The DOC is an event sequence consisting in 2 consecutive *orientationChange* events. $DOC \triangleq \langle orientationChange, orientationChange \rangle$.

3.3 | Equivalence and similarity between GUI objects

The formal definition of DOC GUI failure of different types will rely on the equivalence and similarity relations between GUI states. These relations in turn depend on the equivalence and similarity relations between the GUI component objects.

We assume that 2 objects are *similar* iff their type, position, and size properties assume exactly the same values. On the other side, 2 objects are *equivalent* iff all their properties assume exactly the same values.

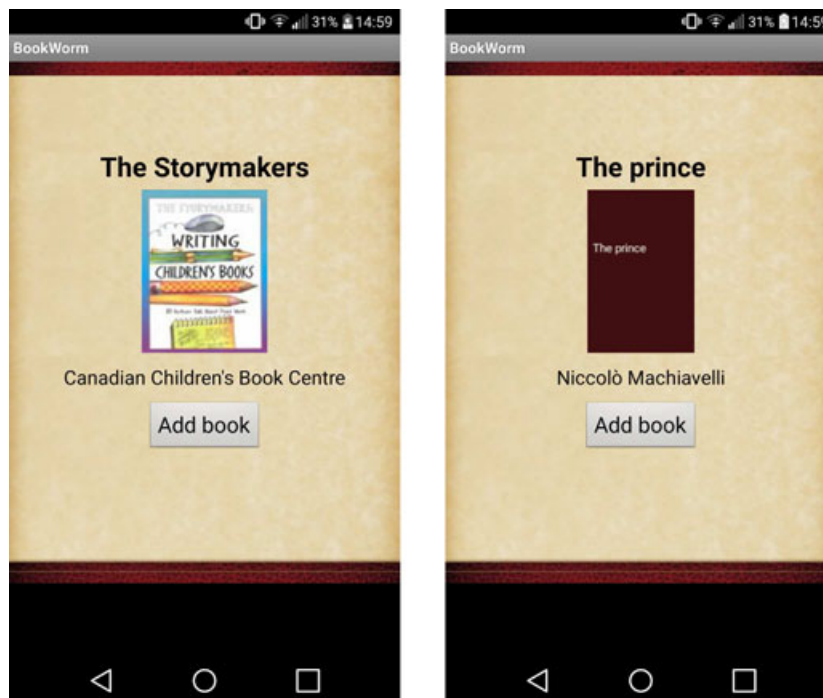
We say that 2 GUI states are *equivalent* if for each object of the former GUI state, there is exactly one equivalent object in the latter state, and vice-versa, for each object of the latter GUI state, there is exactly one similar object in the former state.

Figure 5A and 5B reports 2 GUIs of the Bookworm app that are not equivalent since they include 3 not equivalent objects.

They have 2 similar TextView objects since they assume the same values the 3 fundamental properties, but these objects differ for their textual values. Analogously, the ImageView object differs for the source value it assumes in the 2 graphical user interfaces. The Button object instead is equivalent among the 2 GUIs.

Definition 10. Two GUI objects o_i and o_j are similar when only their fundamental properties values coincide, while they may differ as to other properties.

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



(A) GUI one

(B) GUI two

FIGURE 5 Example of 2 similar GUIs that are not equivalent

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.

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



(A) Start GUI



(B) End GUI showing Extra failure



(C) End GUI showing Missing failure



(D) End GUI showing Wrong failure

FIGURE 6 Example of 2 similar GUIs that are not equivalent

Definition 12. Two GUI states S_i and S_j are equivalent

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

3.4 | DOC GUI Failures

We exploit the definitions presented so far to formalize the different types of DOC GUI failures. Given a GUI start state S and the GUI end state $\text{doc}(S)$ reached after a *doc* event, we have a DOC GUI failure if and only if S and $\text{doc}(S)$ are not equivalent.

Definition 13. Given a GUI state $S \in AS_{AUT}$, the DOC causes a DOC GUI failure if $S \not\cong \text{DOC}(S)$.

Now, we leverage the definitions given in this section 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 event is defined as follows:

$$\exists o_j \in \text{DOC}(S) : o_j \sim o_i, \forall o_i \in S \Rightarrow \exists \text{DOC GUI Failure} : (\text{DOC GUI Failure}).\text{type} = (\text{Extra}, o_j.\text{type}).$$

Comparing the GUI state in Figure 6A and the one in Figure 6B obtained after a DOC event, we see that there is a Dialog object appearing in the end state that is not present in the start state; thus, we have 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 event is defined as follows:

$$\exists o_j \in S : o_j \sim o_i, \forall o_i \in \text{DOC}(S) \Rightarrow \exists \text{DOC GUI Failure} : (\text{DOC GUI Failure}).\text{type} = (\text{Missing}, o_j.\text{type}).$$

Comparing the GUI state in Figure 6A and the one in Figure 6C obtained after a DOC event, we see that there are 2 Button objects in the start state that are no longer present in the end state; thus, we have 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 event is defined as follows:

$$\exists o_j \in S : \exists o(i) \in \text{DOC}(S), o(j) \sim o(i), o(j) \not\cong o(i) \Rightarrow \exists \text{DOC GUI Failure} : (\text{DOC GUI Failure}).\text{type} = (\text{Wrong}, o_j.\text{type}).$$

As an example, considering the GUI state in Figure 6A and the one in Figure 6D obtained after a DOC event, we see that there are 2 EditText objects in the start state that are still contained in the end state but have a different text value; thus, we have 2 DOC GUI failures having Wrong mode and EditText scope.

4 | EXPLORATORY STUDY 1

Our motivating examples showed that DOC GUI failures affect mobile apps irrespective of their mobile platform. We decided to investigate such failures in the context of Android apps, because of the great success of this platform[†] and to the large availability of apps in markets and open-source repositories.

We conducted our first study with the aim of exploring and classifying DOC GUI failures occurring in real Android apps. In this study, we considered open-source apps, which gave us 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 with respect to their mode and scope.

G3 - to find possible common faults causing DOC GUI failures.

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

[†]<https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems>

TABLE 1 Dataset 1 construction

Criteria	No. of Apps
Apps in F-Droid	2030
Provide an issue tracker	1807
Updated in the last 12 months	762
Allow Orientation Change	685
Dataset	68

TABLE 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 Klocker	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	Siletric	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
Lyricaly	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

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,[‡] a well-known repository of Free and Open-Source Software (FOSS) applications for the Android platform. F-Droid offers direct access to app 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 [21, 22, 23] and contains a growing number of applications belonging to different categories.

[‡]<https://f-droid.org/>

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, to be able to contact them and receive their feedback about the DOC GUI failures detected in the study. We needed indeed their opinion to discard the apparent DOC GUI failures that were instead a manifestation of an expected GUI behavior, ie, a feature of the analyzed application.

To this aim, we used the 3 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, we 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 screen portrait and landscape layouts.

Table 1 shows how we constructed our dataset. When our study was performed, the F-Droid repository contained 2030 apps, but only 1807 of them provided an issue tracker. Among these 1807 apps, 762 were updated in the last 12 months. Of the 762 apps, 685 allowed orientation changes. Finally, our dataset was built by randomly selecting the 10% of the 685 that met our criteria. Table 2 lists name, version, and category of the 68 apps in our dataset that covers 14 of the 17 categories provided by F-Droid.

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 cases in the domain of Android mobile applications has been used by Zhang and Elbaum [24] and by Adamsen et al [12].

Listing 1 Example of amplified test case for the A Time Tracker app

```
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.sendKeys(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))
...
}

```

To obtain the initial set of test cases, we involved 12 master students at the University of Naples. The students had been trained about automatic testing of GUI-based applications by techniques of Capture & Replay. Each student was asked to record a number of test cases sufficient to cover the features provided by 4 or 5 applications of the dataset. The students had one semester to accomplish their tasks, and each application was tested by a single student. The students exploited the Robotium Recorder tool⁵ for recording GUI test cases.

We automatically amplified the test cases recorded by the students by injecting after each user event a snippet of Robotium code that fires a DOC and checks the presence of the 3 DOC GUI failure modes through appropriate assertions. These assertions are based on the definitions given in Section 3.4. The code reported in the Listing 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 2 descriptions (CHECK ASSERTIONS).

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.

4.3 | DOC GUI failure validation

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

At first, one PhD student and one research fellow analyzed the collected GUI failures to remove the *duplicate* ones and obtain a list of unique DOC GUI failures.

We made the assumption that when 2 GUI failures are characterized by the same scope and mode, have equivalent start states and equivalent end states, and affect the same app, they are very likely to be considered as duplicate GUI failures. We could have reported all the detected GUI failures to developers, giving them the responsibility to decide whether or not 2 or more failures were actually duplicate GUI failures. However, we preferred this way of counting to not annoy the developers by flooding them with multiple similar (if not identical) requests.

Then we had to assess whether each failure was actually the manifestation of an incorrect GUI state rather than an intended behavior of the application. To this aim, we chose to consult 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 DOC GUI failure we had observed and a sequence of events leading to it. Once we received the developer answers, we analyzed them 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 reports the data we collected at the end of the validation activity. For each application, we reported the number of the unique DOC GUI failures we found, the number of accepted issues (giving the number of validated failures), the number of the issues that were not accepted by the developers (ie, the number of false positives), and the number of issues for which we did not receive any answer from the developers. As data show, we detected a total of 439 unique GUI failures 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. Sixty-one issues were not accepted by the developers as failures, whereas we did not receive any answer for 80 issues that remained as pending. Among the 298 accepted failures, only 7 were already known to their developers.

⁵<https://robotium.com/products/robotium-recorder>

TABLE 3 DOC GUI failures found in open-source applications

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
Lyricaly	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

continues

TABLE 3 Continued

App	DOC GUI failures	Accepted issues	Not accepted	Not answered
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

We analyzed the answers given by developers who refused the issues. In many cases, we observed that 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, we decided not to count them as accepted failures to avoid that the experimental results were biased by our personal and possibly misleading interpretations of the developers' answers. As a result, the data reported in Table 3 can be considered as a lower bound.

4.4 | DOC GUI failure classification

In this step, we characterized each validated GUI failure 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 4 reports the results of this classification.

As the table shows, we obtained 13 classes of Missing GUI failures that involved 13 different types of GUI objects. We 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, only 5 times over 298.

If we consider the types of GUI objects involved in failures, we can observe that there are GUI object types more involved in failures than other ones, ie, 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[†]; for each reported GUI failure, we provide a link to the issue opened in the app bug tracker.

4.5 | Common faults identification

In the previous step, we observed that 3 classes of DOC GUI failures, ie, (Missing, Dialog), (Wrong ListView), and (Wrong, ScrollView) occurred multiple times and affected more than 10 applications.

[†]<https://docs.google.com/spreadsheets/d/1k8lbnDKH9K-9kmTGI9Wnc-FPIrOOEJ2dCRd8Uqu9JGk/edit?usp=sharing>

TABLE 4 Classification of the DOC GUI failures found on the open source applications

Failure mode	Failure scope	No. of occurrences	No. of 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
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
	Scroll View	2	1
	Dialog	1	1

- **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 consisted 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 consisted 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 is placed into the list.

Since we had a significant sample of failures of these 3 types, we decided to investigate them to assess whether the occurrences of the same types of failure had similar causes.

We analyzed the source code of the Android apps exposing these failures and looked for the faults causing them. In our analysis, we considered 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, and 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.

[#] <https://developer.android.com/reference/android/app/DialogFragment.html>

Listing 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 BackupDialogFragment 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(); } }

```

At the end of our analysis, we actually detected 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, we were able to describe each class of fault using a template made of 7 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, we report the descriptions of the 6 classes of faults.

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 objects¹; 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 screen²;

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 developer can implement a class that extends `DialogFragment` and create the desired dialog in its `onCreateDialog()` callback method. He creates an instance of this class and calls `show()` on that object. The dialog appears, but it will not disappear when the activity is destroyed and recreated due to orientation changes; **Example:** We found an example of this fault 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 2 shows the relevant code. We 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.

¹ <https://developer.android.com/guide/topics/ui/dialogs.html>

² <https://developer.android.com/reference/android/app/DialogFragment.html>

Listing 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();
+   }
}

```

Listing 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"
    tools:context=".activities.EventDetailsActivity">

```

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 shows an example of this fault; we detected it in the SettingsActivity class of the app StageFever. When the user click on the Font Size of Notes options in the app settings, a dialog pops up. But it disappears on orientation changes. We 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.

4.5.3 | Missing Id in XML layout

Id: MIXL;

Associated failure: Wrong ScrollView;

Location: ScrollView element in layout XML files;

Background: 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,^{‡‡} 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, eg, 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;

^{††}<https://developer.android.com/guide/components/activities/activity-lifecycle.html>

^{‡‡}<https://developer.android.com/studio/index.html>

Listing 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"
  />

```

Example: The xml code in Listing 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 issue⁵⁵ adding an id to the scrollview.

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, ie, `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 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.

4.5.5 | List adapter not set in onCreate method

Id: LANSCLM;

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, eg, the position of the scrollable list is not preserved;

A possible fix: To call the adapter setter inside the `onCreate()` method;

Example: We found an example of this fault 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 losing the effect of the user interaction. As we see in Listing 6, the `showData` method that sets the list adapter is called by the overridden `onResume` method. To fix this issue, we simply moved the call to `showData` in the `onResume` method.

4.5.6 | List filled through background thread

Id: LFTBT;

Associated failure: Wrong `ListView`;

⁵⁵<https://github.com/cbeyls/fosdem-companion-android/commit/b2e50f8e4dea7739f776373f1c3669ce70c2deb5>

TABLE 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 6 Relationships matrix 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

Listing 6 Example of a LANSCM Fault and a fix in the app BeeCount

```

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

```

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^{¶¶};

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, ie, setting the `ListView` adapter, without having to manipulate threads and/or handlers.^{##} 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^{||};

Example: Listing 7 shows an example of this fault; we detected it 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, eg, scroll position, on orientation changes.

4.5.7 | Common faults summary

Table 5 summarizes the 6 classes of common faults we found and reports the fault acronym and the name we gave to the fault.

In Table 6, we report the relation among the 3 specific types of failures we considered in the fault study and the 6 classes of faults we inferred.

^{¶¶} <https://developer.android.com/guide/components/processes-and-threads.html>

^{##} <https://developer.android.com/reference/android/os/AsyncTask.html>

^{||} <https://developer.android.com/guide/components/loaders.html>

Each element of the table reports the number of apps where the failure type occurred because of 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 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, we observed that the FTA fault originated 2 different failures type in different apps.

4.6 | Study conclusion

At the end of the exploratory study, we obtained several interesting results that allowed us to reach the 3 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. Indeed, we verified that more than the 86% of the analyzed app sample exposed at least one DOC GUI failure. This datum suggests that the likelihood for an app user to encounter these failures is very high, at least when he uses open-source applications.

We cannot exclude that other testing techniques, different from the one used in our 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 showed us that most of the failures belonged to the category of Missing objects and Wrong objects, with 64% and 34% of apps affected by them, respectively.

Listing 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);
-   }
-   ...
-   @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.  
+ }
```

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 we found. Lastly, we discovered that 3 types of failure, ie, (Missing, Dialog), (Wrong, ListView), and (Wrong, ScrollView), were more widespread among the apps than the other ones, since each one of them affected more than 10 applications. This result provided us a subsets of failures worth to be investigated deeper.

Concerning the goal G3, our results showed that a subset of observed types of failures were due to the same classes of faults that occurred in several different applications. Thanks to our analysis, we could conclude that these classes of faults can be considered specific of Android apps, rather than isolated programming errors.

5 | EXPLORATORY STUDY 2

Our first exploratory study provided us evidence about the widespread diffusion of DOC GUI failures of different types in real Android apps coming from the open-source world. Since we did not want to limit the validity of our conclusions solely to the context of open-source apps, which are usually less mature than the ones available through the official app market, we decided to perform a second study. In this further study, we extended our analysis to the context of industrial-strength apps selected from the Google Play Android app market, with the aim of reaching the same 2 goals G1 and G2 of the previous study. The study does not have the G3 goal of the former study, because of our impossibility of accessing the source code of the non-open-source considered apps.

In this study, we 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.

5.1 | Objects selection

We picked 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 to select the most popular apps allowing the orientation change of the display and having the best quality perceived by the users. We randomly selected 10 apps that met the proposed inclusion criteria. Table 7 lists name, version, number of installs, and average rating of each selected app. These data are related to the period in which we performed our study.

5.2 | Apps testing

In this step, we tested all the applications by exploiting the same amplification technique used in the former study for testing the open source apps. To obtain the initial set of test cases, we selected 5 master students not involved in the previous study. We asked each student to record 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 to fire a DOC and then to check the presence of the 3 DOC GUI failure modes through appropriate assertions after each recorded event, according to the presented in Section 4. The amplified test cases were launched on a real LG G4 H815 device with Android 6.0. We collected the GUI failures automatically detected by these test cases.

TABLE 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.release	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

TABLE 8 DOC GUI failures found on top 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

5.3 | DOC GUI failures validation

The DOC GUI failures found in the previous step were manually analyzed by a PhD student to remove the duplicate ones. Then we reported the unique failures to the Android customer support team offered by each app provider. In this case, we did not have direct contact with the app developers so we had to interpret and answer the emails autogenerated by the app providers to validate the reported failures. We got a final answer only from the Dropbox and Pinterest support teams; both of them accepted our issues as failures.

Therefore, for validating the remaining DOC GUI failures, we decided to refer to the GUI Consistency Design Principle stating that, "in a GUI, the same action should always yield the same result" [25]. According to this principle, we decided to check the app behavior exhibited after the DOC 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 8 shows the DOC GUI failures we 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 we found more failures than the others.

5.4 | DOC GUI failures classification

In this step, we classified the failures in terms of their mode and scope according to the proposed classification framework. Table 9 reports for each app the number of occurrences of the failure types they exposed.

On the other hand, for each type of DOC GUI failure, we evaluated how many times it occurred and the number of applications that exposed it. Table 10 shows the results of the classification. The results we obtained in 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

TABLE 9 Distribution of the DOC GUI failures found on the 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
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

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.^{***} We report 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.

^{***} <https://docs.google.com/spreadsheets/d/1xhOudp3FBj4MTHeRK4LWegwRA9s&uscore;ltuh6Pwkali-ZA/edit?usp=sharing>

TABLE 10 Classification of the DOC GUI failures found on the Google Play applications

Failure mode	Failure scope	No. of occurrences	No. of 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
	RecyclerView	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

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, we could conclude that this problem is widespread in the field of Android apps and impact both mature and less mature 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, we can see that 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.

6 | THREATS TO VALIDITY

This section discusses the threats that could affect the validity of the results obtained in Study 1 and Study 2.

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 [26]. In our studies, since we did not have access to the requirements of each app, there was the risk that the GUI failures we detected could not be actual failures, but the manifestation of apps' expected behavior. This may be a possible threat to the construct validity of our 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 us 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 does not definitely assure us that what we observed were aberrant app behaviors, it gives us additional evidence to assume it.

Internal Validity: This aspect of validity assesses that there are no uncontrolled variables of the experiment that had an effect on the outcome [26]. Such threats typically do not affect exploratory studies like the ones reported in this paper [27].

However, we cannot exclude con-causes besides DOC events that triggered the GUI failures we observed, eg, 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 performed to further investigate this aspect.

External validity: This aspect of validity is concerned with to what extent it is possible to generalize the findings to other contexts [26]. A possible threat to the generalizability of our experimental results could be the representativeness of the sample of Android apps.

In Study 1, 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 our analysis to the open-source world and confirmed and strengthened our findings by considering industrial-strength apps in Study 2. 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.

We cannot claim that our results generalize beyond the inclusion criteria we applied. Moreover, we cannot exclude that specific characteristics of the apps we analyzed (such as the types of GUI widget they rely on, or their category) may have influenced our experimental results. To further extend the validity of our study, a controlled experiment involving a larger set of apps with selected characteristics should be performed.

7 | RELATED WORK

In this paper, we 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. Here, we discuss some of the most related ones.

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 [28].

Several event-based testing techniques have been proposed in the literature to test mobile apps. These techniques span from random testing (eg, Machiry et al [22], Hu et al [29], Amalfitano et al [30]) to symbolic-execution-based test-case generation (eg, Anand et al [31], Mirzaei et al [32]), ripping-based testing (eg, Amalfitano et al [33], Azim et al [34], Choi et al [35]), pattern-based testing (eg, Cunha et al [36], Moreira et al [37]), model-based testing (eg, Amalfitano et al [4], Nabuco et al [38]), and combinations of model-based and combinatorial testing (eg, Jensen et al [39], Nguyen et al [40]). Unlike our 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 [5], security [6, 7], responsiveness [8], and energy consumption [9].

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 [15] 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.

It proposes a technique for testing and generating oracles focusing on a subset of features, ie, Double rotation, Killing and Restarting, Pausing and Resuming, and 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 [12] 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 DOC events. Moreover, they provide a classification of the failures and bugs that their technique is able to find. Among the 4 categories proposed in their classification, there are failures related to the GUI, ie, Unexpected Screen and Element disappears, that are similar to the ones we dealt with. They performed an experiment involving 4 real Android apps and the results showed that *THOR* was able to detect 66 distinct problems, most of which are due to the 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 Zaeem et al [15] and Adamsen et al [12] gave us a hint about the relevance of the problem addressed in our paper, since several failures discovered by their techniques were GUI-related and exposed by the orientation change. While these works focused only on failure detection and classification, we also investigated the faults that cause a relevant part of these failures.

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 Neamtiiu [29]. The authors proposed 8 types of bug by analyzing the faults they found in 10 open-source Android apps. Their fault classification is based on bug report analysis, whereas we abstract our Android fault classes by analyzing the causes of GUI failures we observed by testing 68 real apps. Moreover, their fault categories are described at a 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 [14] focused on a specific fault class because of 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 within this paper. But, unlike our work that distinguishes among different types of GUI failures, their paper generically classifies them as KR errors.

Banerjee et al [41, 42] focused on another type of problem in Android apps, ie, abnormal energy consumption, called energy hotspots. These authors proposed an automated test generation framework aimed at detecting energy hotspots. Like us, they also explored the causes of these failures in the Android app code and defined 4 fault classes (Resource Leak, Wakelock Bug, Vacuous Background Services, and Immortality Bug), each one corresponding to a different energy hotspot category. Similarly to our 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 [17, 43] 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. Our work instead introduces 6 classes of faults discovered by testing real Android apps rather than by mining bug tracking repositories. Most 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 us, since they designed a specific operator to force testing of orientation changes. Our work focused on this specific issue and provided a comprehensive analysis that spans from GUI failures to code faults.

8 | CONCLUSIONS AND FUTURE WORK

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

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 analyzed apps were affected by these failures, highlighting that this problem is widespread in the context of Android mobile apps. Our study is the first one to point out the relevance of this issue in mobile apps context.

Almost all the failures detected by our study were novel and not already present in issue trackers. We make available the set of collected GUI failures as open source since it provides the largest currently available dataset of this kind of failures. It 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, we 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 to verify this hypothesis. The study also highlighted 3 types of failures that were more common than others among mobile apps and provided us 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 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.

Our work targeted GUI failures triggered by the orientation change event. In the future, we will consider other mobile-specific events, which may cause GUI failures, such as Receiving a Call, Sending an application in Background, and Pressing the Back button of the device. The paper addressed GUI failures in the context of Android. However, according to the results of our preliminary investigation, we are aware that the problem affects other mobile platforms. Thus, we plan to extend this work by considering other mobile operating systems, such as iOS and Windows10.

ORCID

Anna Rita Fasolino  <http://orcid.org/0000-0001-7116-019X>

REFERENCES

1. Statista, Number of smartphone users worldwide from 2014 to 2019 (in millions), 2016. <https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/>.
2. Statista, Number of apps available in leading app stores as of June 2016, 2016. <http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/>.

3. S. Zein, N. Salleh, and J. Grundy, *A systematic mapping study of mobile application testing techniques*, *J. Syst. Softw.* **117** (2016), no. C, 334–356. <https://doi.org/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*, *IEEE Softw.* **32** (2015), no. 5, 53–59.
5. G. Canfora, F. Mercaldo, C. A. Visaggio, M. D'Angelo, A. Furno, and C. Manganelli, in *A case study of automating user experience-oriented performance testing on smartphones*, 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, 2013, pp. 66–69.
6. A. Avancini and M. Ceccato, in *Security testing of the communication among Android applications*, 2013 8th International Workshop on Automation of Software Test (AST), IEEE Press, San Francisco, California, 2013, pp. 57–63. <http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=6595792>.
7. G. Canfora, F. Mercaldo, and C. A. Visaggio, *An HMM and structural entropy based detector for android malware: An empirical study*, *Comput. Secur.* **61** (2016), 1–18. <https://doi.org/10.1016/j.cose.2016.04.009>.
8. S. Yang, D. Yan, and A. Rountev, in *Testing for poor responsiveness in android applications*, 2013 1st International Workshop on the Engineering of Mobile-Enabled Systems (MOBS), IEEE Computer Society, Washington, DC, 2013, pp. 1–6.
9. M. Linares-Vásquez, G. Bavota, C. E. B. Cárdenas, R. Oliveto, M. Di Penta, and D. Shybyanyk, in *Optimizing energy consumption of guis in android apps: A multi-objective approach*, Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, ACM, New York, NY, USA, 2015, pp. 143–154. <http://doi.org/10.1145/2786805.2786847>.
10. 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*, *J. Syst. Softw.* **125** (2017), 322–343. <https://doi.org/10.1016/j.jss.2016.12.017>.
11. H. Muccini, A. di Francesco, and P. Esposito, in *Software testing of mobile applications: Challenges and future research directions*, 2012 7th International Workshop on Automation of Software Test (AST), IEEE, Zurich, Switzerland, 2012, pp. 29–35. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6228987.
12. C. Q. Adamsen, G. Mezzetti, and A. Möller, in *Systematic execution of Android test suites in adverse conditions*, Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015), ACM, Baltimore, MD, USA, 2015, pp. 83–93. <http://doi.org/10.1145/2771783.2771786>.
13. K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Shybyanyk, in *Automatically discovering, reporting and reproducing android application crashes*, 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST), IEEE Computer Society, Washington, DC, 2016, pp. 33–44.
14. Z. Shan, T. Azim, and I. Neamtii, in *Finding resume and restart errors in android applications*, Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, october 30 - november 4, 2016, Amsterdam, the Netherlands, 2016, pp. 864–880. <http://doi.org/10.1145/2983990.2984011>.
15. R. N. Zaeem, M. R. Prasad, and S. Khurshid, in *Automated generation of oracles for testing user-interaction features of mobile apps*, Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation, ICST '14, IEEE Computer Society, Washington, DC, USA, 2014, pp. 183–192. <https://doi.org/10.1109/ICST.2014.31>.
16. V. Lelli, A. Blouin, and B. Baudry, in *Classifying and qualifying gui defects*, 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (icst), IEEE Computer Society, Washington, DC, 2015, pp. 1–10.
17. L. Deng, J. Offutt, P. Ammann, and N. Mirzaei, *Mutation operators for testing android apps*, *Inf. Softw. Technol.* **81** (2017), 154–168. <https://doi.org/10.1016/j.infsof.2016.04.012>.
18. Google Android, *Android—What to test*, 2015. <http://goo.gl/AL22tJ>.
19. *Ieee standard classification for software anomalies*, IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993), IEEE Computer Society, Washington, DC, 2010, pp. 1–23. <https://doi.org/10.1109/IEEESTD.2010.5399061>.
20. K. Holl and F. Elberzhager, in *Mobile application quality assurance: Reading scenarios as inspection and testing support*, 2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Vol. 00, IEEE Computer Society, Washington, DC, 2016, pp. 245–249.
21. S. R. Choudhary, A. Gorla, and A. Orso, in *Automated test input generation for android: Are we there yet? (E)*, 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, November 9–13, 2015, Lincoln, NE, USA, 2015, pp. 429–440. <https://doi.org/10.1109/ASE.2015.89>.
22. A. Machiry, R. Tahiliani, and M. Naik, in *Dynodroid: An input generation system for android apps*, Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, ACM, New York, NY, USA, 2013, pp. 224–234. <http://doi.org/10.1145/2491411.2491450>.
23. K. Mao, M. Harman, and Y. Jia, in *Sapienz: Multi-objective automated testing for android applications*, Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, ACM, New York, NY, USA, 2016, pp. 94–105. <http://doi.org/10.1145/2931037.2931054>.
24. P. Zhang and S. G. Elbaum, *Amplifying tests to validate exception handling code: An extended study in the mobile application domain*, *ACM Trans. Softw. Eng. Methodol.* **23** (2014), no. 4, 32:1–32:28. <http://doi.org/10.1145/2652483>.
25. D. A. Norman, *The Design of Everyday Things*, Basic Books, Inc., New York, NY, USA, 2002.
26. C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, and B. Regnell, *Experimentation in Software Engineering*, Springer:Berlin, Heidelberg, 2012. <https://doi.org/10.1007/978-3-642-29044-2>.
27. R. K. Yin, *Case Study Research: Design and Methods*, Applied Social Research Methods, SAGE Publications:Thousand Oaks, California, 2009. <https://books.google.it/books?id=FzawlAdiIHKC>.
28. F. Belli, M. Beyazit, and A. Memon, in *Testing is an event-centric activity*, 2012 IEEE Sixth International Conference on Software Security and Reliability Companion (SERE-C), IEEE Computer Society, Washington, DC, 2012, pp. 198–206.
29. C. Hu and I. Neamtii, in *Automating gui testing for android applications*, Proceedings of the 6th International Workshop on Automation of Software Test, AST '11, ACM, New York, NY, USA, 2011, pp. 77–83. <http://doi.org/10.1145/1982595.1982612>.
30. D. Amalfitano, N. Amatucci, A. R. Fasolino, P. Tramontana, E. Kowalczyk, and A. M. Memon, in *Exploiting the saturation effect in automatic random testing of android applications*, 2nd ACM International Conference on Mobile Software Engineering and Systems, Mobilesoft 2015, May 16–17, 2015, Florence, italy, 2015, pp. 33–43. <https://doi.org/10.1109/MobileSoft.2015.11>.
31. S. Anand, M. Naik, M. J. Harrold, and H. Yang, in *Automated concolic testing of smartphone apps*, Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12, ACM, New York, NY, USA, 2012, pp. 59:1–59:11. <http://doi.org/10.1145/2393596.2393666>.

32. N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood, *Testing android apps through symbolic execution*, SIGSOFT Softw. Eng. Notes **37** (2012), no. 6, 1–5. <http://doi.org/10.1145/2382756.2382798>.
33. D. Amalfitano, A. R. Fasolino, P. Tramontana, S. D. Carmine, and A. M. Memon, in *Using GUI ripping for automated testing of android applications*, IEEE/ACM International Conference on Automated Software Engineering, ASE'12, September 3–7, 2012, Essen, Germany, 2012, pp. 258–261. <http://doi.org/10.1145/2351676.2351717>.
34. T. Azim and I. Neamtiu, *Targeted and depth-first exploration for systematic testing of android apps*, SIGPLAN Not. **48** (2013), no. 10, 641–660. <http://doi.org/10.1145/2544173.2509549>.
35. W. Choi, G. Necula, and K. Sen, in *Guided gui testing of android apps with minimal restart and approximate learning*, Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA '13, ACM, New York, NY, USA, 2013, pp. 623–640. <http://doi.org/10.1145/2509136.2509552>.
36. M. Cunha, A. C. R. Paiva, H. S. Ferreira, and R. Abreu, in *PETTool: A pattern-based GUI testing tool*, 2010 2nd International Conference on Software Technology and Engineering (ICSTE), Vol. 1, IEEE, San Juan, PR, 2010, pp. V1–202–VI–206. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5608882.
37. RMLM Moreira, A. C. R. Paiva, and A. Memon, in *A pattern-based approach for GUI modeling and testing*, IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013, november 4–7, 2013, Pasadena, CA, USA, 2013, pp. 288–297. <https://doi.org/10.1109/ISSRE.2013.6698881>.
38. M. Nabuco and A. C. R. Paiva, in *Model-based test case generation for Web applications*, 14th International Conference on Computational Science and Applications (ICCSA 2014), Springer International Publishing: Cham, Switzerland, 2014.
39. C. S. Jensen, M. R. Prasad, and A. Møller, in *Automated testing with targeted event sequence generation*, Proceedings of the 2013 International Symposium on Software Testing and Analysis, ISSTA 2013, ACM, New York, NY, USA, 2013, pp. 67–77. <http://doi.org/10.1145/2483760.2483777>.
40. C. D. Nguyen, A. Marchetto, and P. Tonella, in *Combining model-based and combinatorial testing for effective test case generation*, Proceedings of the 2012 International Symposium on Software Testing and Analysis, ISSTA 2012, ACM, New York, NY, USA, 2012, pp. 100–110. <http://doi.org/10.1145/2338965.2336765>.
41. A. Banerjee, L. K. Chong, C. Ballabriga, and A. Roychoudhury, *Energypatch: Repairing resource leaks to improve energy-efficiency of android apps*, IEEE Trans. Softw. Eng. **PP** (2017), no. 99, 1–1.
42. A. Banerjee, H. Guo, and A. Roychoudhury, in *Debugging energy-efficiency related field failures in mobile apps*, Proceedings of the International Conference on Mobile Software Engineering and Systems, MOBILEsoft '16, May 14–22, 2016, Austin, Texas, Usa, 2016, pp. 127–138. <http://doi.org/10.1145/2897073.2897085>.
43. L. Deng, N. Mirzaei, P. Ammann, and J. Offutt, in *Towards mutation analysis of android apps*, 10th International Workshop on Mutation Analysis (Mutation 2015) Co-Located with IEEE EIGHTH International Conference on Software Testing, Verification and Validation (ICST 2015), IEEE Computer Society, Washington, DC, 2015, pp. 1–10.

How to cite this article: Amalfitano D, Riccio V, Paiva ACR, Fasolino AR. Why does the orientation change mess up my Android application? From GUI failures to code faults.. *Softw Test Verif Reliab.* 2018;28:e1654. <https://doi.org/10.1002/stvr.1654>