Is This the Lifecycle We Really Want? An Automated Black-Box Testing Approach for Android Activities

Vincenzo Riccio
University of Naples Federico II
Naples, Italy
vincenzo.riccio@unina.it

Domenico Amalfitano
University of Naples Federico II
Naples, Italy
domenico.amalfitano@unina.it

Anna Rita Fasolino
University of Naples Federico II
Naples, Italy
anna.fasolino@unina.it

ABSTRACT

Android is today the world's most popular mobile operating system and the demand for quality to Android mobile apps has grown together with their spread. Testing is a well-known approach for assuring the quality of software applications but Android apps have several peculiarities compared to traditional software applications that have to be taken into account by testers. Several studies have pointed out that mobile apps suffer from issues that can be attributed to Activity lifecycle mishandling, e.g. crashes, hangs, waste of system resources. Therefore the lifecycle of the Activities composing an app should be properly considered by testing approaches. In this paper we propose ALARic, a fully automated Black-Box Event-based testing technique that explores an application under test for detecting issues tied to the Android Activity lifecycle. ALARic has been implemented in a tool. We conducted an experiment involving 15 real Android apps that showed the effectiveness of ALARic in finding GUI failures and crashes tied to the Activity lifecycle. In the study, ALARic proved to be more effective in detecting crashes than Monkey, the state-of-the practice automated Android testing tool.

ACM Reference Format:
Vincenzo Riccio, Domenico Amalfitano, and Anna Rita Fasolino. 2018. Is This the Lifecycle We Really Want? An Automated Black-Box Testing Approach for Android Activities. In *(ISSTA Companion/ECOOP Companion'18), July 16–21, 2018, Amsterdam, Netherlands.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3236454.3236504

1 INTRODUCTION

The number of users of mobile technology and smartphones is steadily growing and is forecast to surpass 2.5 billion in 2019 [18]. Today, Android is the world's most popular mobile operating system [19]. More and more people around the world rely on mobile software applications (*apps*) to carry out various daily tasks. Thus, the demand for quality to mobile apps has grown together with their spread. As a consequence, mobile developers should give proper consideration to the quality of their applications by adopting suitable quality assurance techniques, such as testing. Several techniques and tools are currently available for testing an Android app before it is published to the market [3]. Test automation tools can facilitate software testing activities since they save humans from routine, time-consuming and error-prone manual tasks [7].

Mobile apps have several peculiarities compared to traditional software applications that have to be taken into account by testing techniques and tools [16]. In particular, the small size of mobile devices introduced the need to have on the screen one single focused app at a time. In Android an app is composed by one or more Activities; each Activity represents a single GUI that allows the user to interact with the app.

The Android Framework defines a peculiar lifecycle for Activity instances in order to manage them transparently to the user who can navigate through an app and switch between apps without losing his progress and data; at the same time, it allows not to waste the limited resources of a mobile device, such as memory and battery. The official Android Developer Guide[1] stresses the relevance of the Activity lifecycle feature and warns the developers of the threats it introduces in several sections; therefore it provides recommendations and guidelines to help programmers in the correct handling of the Activity lifecycle.

Despite this, several works in the literature have pointed out that mobile apps, including industrial-strength ones, suffer from issues that can be attributed to Activity lifecycle mishandling [4] [15] [17] [5]. Zein *et al.* [22] performed a systematic mapping study of mobile application testing techniques involving 79 papers and identified possible areas that require further research. Among them, they emphasized the need for specific testing techniques targeting Activity lifecycle conformance.

Some solutions have been presented in the literature to address this problem. A part of them proposed testing techniques that rely on existing testing artifacts [1, 8] or GUI models [21] to automatically generate test cases able to properly exercise the Activity lifecycle. Another work [17] detects through static analysis bugs that may cause a corrupt state when an app is paused, stopped, or killed. Their solution can also automatically generate test cases to reproduce bugs but it needs to modify the app code in order to verify the statically detected issues.

Another group of approaches leverages dynamic analysis to find issues tied to the Activity lifecycle [9, 15]. These dynamic techniques mostly focus on finding a specific type of failure, such as crashes [9, 15] or resource leaks [10]. However, none of them addressed GUI failures that consist in the manifestation of an unexpected GUI state. As pointed out in [5], GUI failures tied to the Activity lifecycle represent a widespread category of issues in Android

ISSTA Companion/ECOOP Companion'18 ,
July 16–21, 2018, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5939-9/18/07…$15.00
https://doi.org/10.1145/3236454.3236490

[1] https://developer.android.com/

apps and there is the need to define effective testing techniques to detect them.

To overcome these limitations, in this paper, we propose ALARic (Activity Lifecycle Android Ripper), a fully automated black-box event-based dynamic testing technique.

ALARic is able to detect both GUI failures and app crashes related to the lifecycle of the Activities of an app by systematically testing each Activity GUI state encountered during the automatic app exploration. To this aim, it leverages mobile-specific events able to exercise the Activity lifecycle and specifically designed testing oracles. Our solution does not require any prior knowledge of the app under test, app modification or manual intervention. The effectiveness of ALARic in detecting issues tied to the Activity lifecycle was demonstrated in an experiment involving 15 real Android apps. The experiment also showed that ALARic was more effective in detecting crashes tied to the Activity lifecycle than the state-of-the practice automated Android testing tool, *i.e.* Monkey, the most widely used tool of this category in industrial settings.

The paper improves the literature on automated GUI testing with the following contributions:

- a novel automated GUI testing technique to detect GUI failures and crashes tied to the Android Activity lifecycle. In particular, ALARic is the first dynamic testing technique that is able to address the issue of GUI failures;
- an experiment involving real Android apps showing the validity of the proposed technique.

The remainder of the paper is structured as follows. Section 2 describes the background and Section 3 provides an overview of the proposed testing approach. Section 4 presents design and implementation details about the tool we developed while in Section 5 we describe the experiment that was performed. Section 6 provides related work. Finally, Section 7 draws the conclusions and presents future work.

2 BACKGROUND

2.1 Activity Lifecycle

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

To provide this rich user experience, Android Activities have a proper lifecycle, transitioning through different states. Figure 1 shows the Activity lifecycle as it is illustrated in the official Android Developer Guide[2]. The rounded rectangles represent all the states an Activity can be in; the edges are labeled with the callback methods that are invoked by the Android platform when an Activity transits between states.

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

The Activity visible in the foreground of the screen and interacting with the user is in the `Resumed` state, either it is created for the first time or resumed from the `Paused` or `Stopped` states. When an Activity has lost focus but is still visible (e.g., a system modal dialog has focus on top of the Activity), it is in the `Paused` state; in this state, the app usually maintains all the user progress and information. When the user navigates to a new Activity, the previous one is put in the `Stopped` state; it still retains all the user information but it is no longer visible to the user. However, when an Activity is in `Paused` or `Stopped` states, the system can drop it from memory if the system resources are needed by other apps and therefore the Activity transits to the `Destroyed` state. When it is displayed again to the user, it is restarted and its saved state must be restored.

Figure 1 also highlights the three key loops of the Activity lifecycle. In the following, according to the Android Developer Guide, we call these loops *Entire Loop*, *Visible Loop*, and *Foreground Loop*, respectively, and report event sequences to exercise each of them:

(1) The *Entire Loop (EL)* of an Activity consists in the Resumed-Paused-Stopped-Destroyed-Created-Started-Resumed sequence of states. This loop can be exercised by events that cause a configuration change, e.g. an orientation change of the screen, that destroys the Activity instance and then recreates it according to the new configuration[3];
(2) The *Visible Loop (VL)* corresponds to the Resumed-Paused-Stopped-Started-Resumed sequence of states during which the Activity is hidden and then made visible again. There are several event sequences able to stop and restart an Activity, e.g. turning off and on the screen or putting the app in background and then in foreground again through the Overview or Home buttons;
(3) The *Foreground Loop (FL)* of an Activity involves the Resumed-Paused-Resumed state sequence. The transition Resumed-Paused can be triggered by opening non full-sized elements such as modal dialogs or semi-transparent activities that occupy the foreground while the Activity is still visible in background. To trigger the transition Paused-Resumed the user should discard this element.

2.2 Issues Tied to the Activity Lifecycle

Android App developers should correctly implement Activities, taking into account their lifecycle. This ensures the app works the way users expect and does not exhibit aberrant behaviors as it transitions through different lifecycle states at runtime. Good implementation of the lifecycle callbacks and the awareness of the Android Framework features can help the programmer to develop

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

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

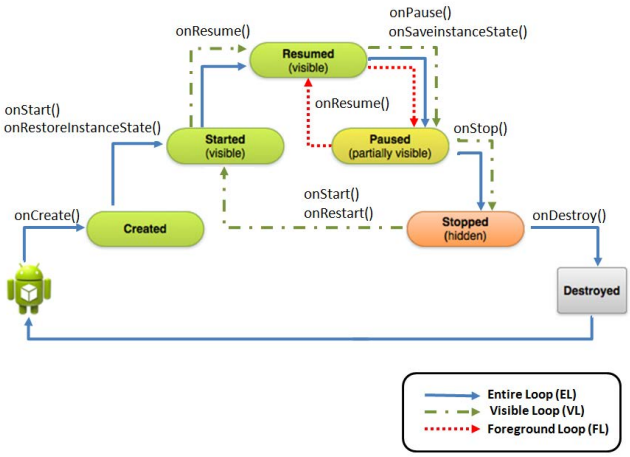
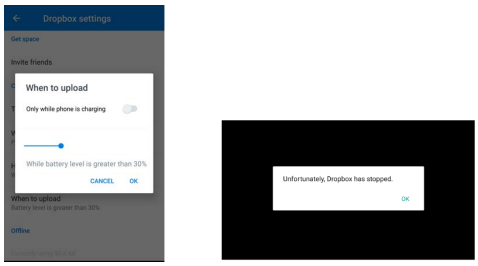


Figure 1: The Android Activity Lifecycle key loops



(a) Before the application of the orientation change

(b) After the application of the orientation change

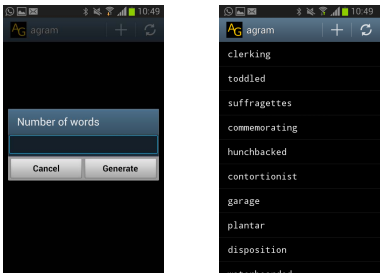
Figure 2: Crash exposed by Dropbox app

apps that behave as expected and prevent a number of issues, such as Crashes and GUI failures.

2.2.1 Crashes. A crash occurs when an app stops functioning properly and exits unexpectedly. When an app crashes, Android terminates its process and displays a dialog to let the user know that the app has stopped[4]. It is a very undesirable app behavior and, indeed, the most blatant one. We are interested in crashes tied to the Activity lifecycle, i.e. crashes triggered by events that exercise the Activity lifecycle. We report an example of crash that we detected in Dropbox version 27.1.2[5], the Android client app offered by the popular file hosting service. If the user selects the third item of the *Camera Uploads* settings, a modal dialog appears (see Figure 2(a)). When the user changes the orientation of the device, the app suddenly stops working, as shown in Figure 2(b).

2.2.2 GUI Failures. GUI failures are a relevant class of failures that may disrupt the user experience and consist in the manifestation of an unexpected GUI state [11]. We focus on GUI failures triggered by exercising the three Activity lifecycle key loops. In

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



(a) Before the application of the double orientation change

(b) After the application of the double orientation change

Figure 3: GUI Failure exposed by Agram app

particular, there may be a GUI failure tied with the Activity lifecycle when the GUI state before the Activity is stopped, paused or destroyed is different from the GUI state displayed after the user returns to the Activity [1, 5, 14, 17, 21]. GUI failures manifest in several ways, i.e. unexpected GUI objects may appear in wrong positions, objects may be rendered with wrong properties, or important objects may disappear from the GUI [5]. We report an example of GUI failure that we found in version 2.7.3 of QKSMS, an Android application that displays anagrams in English and that is available for free on the Google Play Store. If the user chooses to create random words, a modal Dialog appears prompting the number of words the user wants to generate (see Figure 3(a)). When the users change the orientation of the device twice, they naturally expect that the GUI state will remain the same. Instead, the app will exhibit an unexpected GUI state when the Activity is destroyed and then recreated due to the configuration change, i.e. the dialog disappears and a list of random words is rendered on the screen as shown in Figure 3(b).

3 THE ALARIC APPROACH

In this section we present the approach adopted by ALARic in testing Android apps.

ALARic implements a fully automated online testing technique since it explores the application under test (AUT) and at the same time detects aberrant behaviors tied to the Activity lifecycle, i.e. Crashes and GUI failures. ALARic exercises the AUT through input event sequences, being Android apps event-driven software systems [6].

The exploration strategy adopted by ALARic sends random input events to the AUT and systematically executes an input event sequence able to exercise one of the three key Activity lifecycle loops each time a new GUI is encountered for the first time during the app exploration. We define *Lifecycle Event Sequence* a sequence of events able to trigger one of the key loops of the Activity lifecycle. After the Activity lifecycle loop is exercised, ALARic evaluates whether the app exposes any issue related to the Activity lifecycle.

To exercise the three key lifecycle loops, we leverage three Lifecycle Event Sequences, i.e., the *Double Orientation Change (DOC)*, the *Background Foreground (BF)* and the *Semi-Transparent Activity*

Intent (STAI) event sequences. We chose these Lifecycle Event Sequences since each of them is able to exercise a different lifecycle loop. In order to detect GUI failures, we chose Lifecycle Event Sequences for which the GUI state of the Activity should be retained after their executions [21].

The *Double Orientation Change (DOC)* event sequence exercises twice the *EL loop* and consists in a sequence of two consecutive orientation change events. We used the DOC event sequence since 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, 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 [5].

The *Background Foreground (BF)* sequence puts the app in background through the tap of the Home button and then pushes the app again in foreground. It exercises the *VL loop*.

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

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

Figure 4 shows a real example of how ALARic works. In this example, we used the DOC Lifecycle Event Sequence to test the Amaze app version 3.1.2 RC4. The snapshots represent the GUI states encountered during the automatic exploration. The red edges represent Lifecycle Event Sequences, whereas the black edges are random planned events. At each exploration iteration, ALARic describes the current GUI state and verifies whether it has been explored before during the exploration. The GUI states encountered for the first time, i.e. A, C, E, H, L, are exercised by a DOC. Whereas, the GUI states already encountered, i.e. D, F, G, J, K, are exercised by random planned events. The tool compares the GUI states before and after the DOC event and checks whether they are not different. ALARic found 3 GUI failures in this exploration, i.e. after the 3th, 5th and 9th iteration. Moreover, the app crashed after the triggering of the 12th event. When a crash occurs, ALARic starts the app from scratch. The exploration terminates either after the triggering of a predefined number of events or after a given testing time.

4 THE ALARIC TOOL

The ALARic approach has been implemented in the *ALARic Tool*[6]. Fig. 5 shows the tool architecture that is composed by two components, i.e., the *ALARic Engine* and the *Test Executor*.

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

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

During the automatic app exploration, ALARic saves the descriptions of the encountered GUIs. At each exploration iteration, it loads these descriptions for comparing them against the GUI currently rendered on the screen. In this way, it tests only the GUI states encountered for the first time.

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

4.1 ALARic Engine

The online testing process implemented by the *ALARic tool* is described by the UML Activity diagram shown in the *ALARic Engine* component of Figure 5. It extends the generic online testing algorithm presented in the framework proposed by Amalfitano et al. [2]. The steps belonging to the original algorithm are reported in white, whereas the ones introduced by our approach are colored in gray.

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

In the *Describe Current GUI* step, a description of the current GUI state is inferred, according to a GUI description abstraction criterion. The description of a GUI state includes the $(attribute, value)$ pairs assumed by its components at runtime. The description of the current GUI state is compared with the ones previously encountered to evaluate whether it has never been met before during the exploration. The *Exercise Activity Lifecycle* and *Evaluate Oracles* steps are executed when a new GUI state is encountered for the first time. Otherwise, the *Plan Events* and *Run Events* steps are executed.

In the *Exercise Activity Lifecycle* step, one predefined Lifecycle Event Sequence is triggered. The *Evaluate Oracles* step allows the verification of oracles appositely crafted to detect the presence of specific types of Activity lifecycle issues. The current ALARic implementation is able to evaluate two oracles, i.e. GUI failures and crashes. As for GUI failures, similarly to [1, 5, 14, 17, 21], the tool is able to recognize failures that occur when the GUI states before and after the application of a Lifecycle Event Sequence are different. As for the crashes, ALARic checks whether an unhandled exception occurs after the execution of a Lifecycle Event Sequence.

The *Plan Events* step plans the input event sequences that will be fired on the current GUI according to a uniform random scheduling

[6]The ALARic tool is available for download at this link https://goo.gl/ypTMVs.

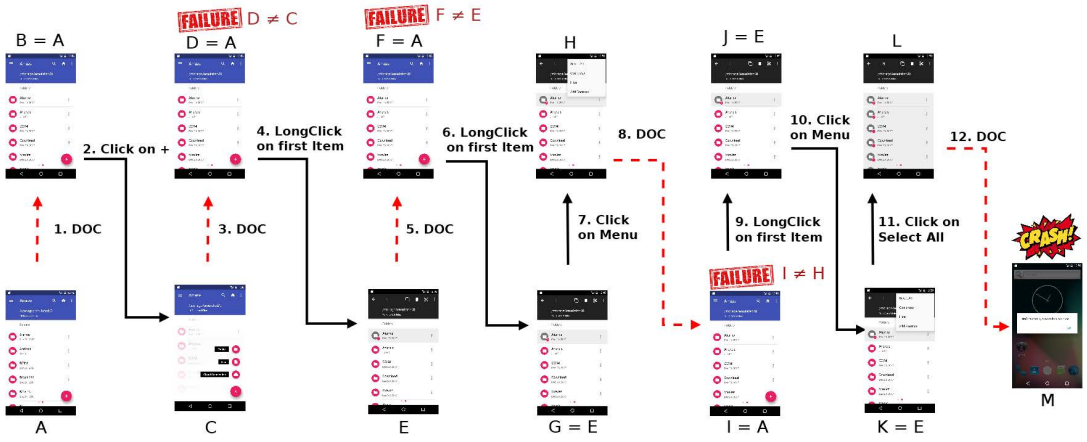


Figure 4: ALARic testing example

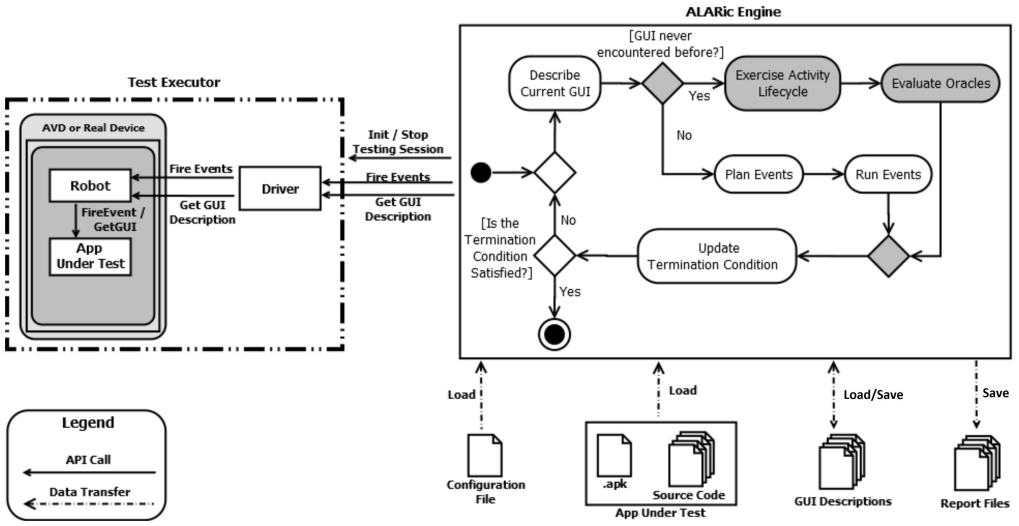


Figure 5: The ALARic tool architecture

strategy. In the *Run Events* step, the planned input event sequence is actually executed.

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

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

4.2 Test Executor

The *Test Executor* component is in charge of executing the testing activities of the *ALARic Engine* on the AUT. It is able to interact with both a physical device and a Android Virtual Device (AVD)[7]. It is made of two components, i.e. *Robot* and *Driver* that interact through *Java socket technology*.

The *Driver* component is in charge to decouple the business logic implemented in the ALARic Engine from the device where the AUT is installed. The *Robot* should run on the same device where the AUT is installed and interacts with it by firing events and describing the GUIs rendered at runtime. This component exploits the APIs provided by the Robotium library[8] the Android Debug Bridge (ADB)[9].

5 EXPERIMENTAL EVALUATION

In this section, we report the study we carried out to investigate the ability of the ALARic tool in detecting issues tied to the Activity

[7]https://developer.android.com/studio/run/emulator.html
[8]https://github.com/RobotiumTech/robotium
[9]https://developer.android.com/studio/command-line/adb.html

lifecycle. We consider as tied to the the Activity lifecycle the issues that are exposed by *Lifecycle Event Sequences*. The study aimed at answering the following research questions:

- **RQ$_1$** How effective is the ALARic tool in detecting issues tied to the Activity lifecycle in real Android apps?
- **RQ$_2$** How does the effectiveness of the ALARic tool in detecting issues tied to the Activity lifecycle in real Android apps compare to the state-of-the-practice?

Some tools have been proposed in literature that exploit dynamic analysis and can find crashes tied to the Activity lifecycle [9, 15]. However, we were unable to compare the ALARic tool against them, since they are either no longer available, or are not supported anymore and are unable to target the latest Android OS and SDK versions. Therefore, we considered the Monkey[10] tool since it is regarded as the current state-of-practice for automated Android app testing [12, 13], being the most widely used tool of this category in industrial settings [23, 24]. It is mainly used in robustness testing processes for revealing app crashes. Moreover, we could not compare the ALARic effectiveness in finding GUI failures against other existing dynamic techniques since our work is the first one to address this issue.

5.1 Objects

As objects of the evaluation, we selected 15 apps that are distributed by the official Google app store[11] whose source code is available in the F-Droid repository[12]. In this way we selected apps that were representative of the typical apps available to Android users. The availability of the source code allowed us to better analyze the detected failures. We chose F-Droid since it is a well-known repository of Free and Open Source Software (FOSS) applications for the Android platform that has been widely used in other studies on Android testing proposed in literature [7, 12–15]. Table 1 reports for each selected app its name, the version we considered and its size.

Table 1: Object Apps

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

[10]https://developer.android.com/studio/test/monkey.html
[11]https://play.google.com/store/apps
[12]https://f-droid.org/

5.2 Metrics

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

- DGF_{DOC}: distinct GUI Failures triggered by the DOC event sequence
- DGF_{BF}: distinct GUI Failures triggered by the BF event sequence
- DGF_{STAI}: distinct GUI Failures triggered by the STAI event sequence
- DGF_{Total}: distinct GUI Failures tied to the Activity lifecycle triggered by either DOC, BF, or STAI event sequences

Analogously, to evaluate the effectiveness of the tools in finding crashes, we considered the number of elements of each of the following sets:

- DC_{DOC}: distinct Crashes triggered by the DOC event sequence
- DC_{BF}: distinct Crashes triggered by the BF event sequence
- DC_{STAI}: distinct Crashes triggered by the STAI event sequence
- DC_{Total}: distinct Crashes tied to the Activity lifecycle triggered by either DOC, BF, or STAI event sequences

Since the same issue may be exposed multiple times during a testing process, we decided to count only the occurrences of distinct issues. We made these assumptions: (1) GUI failures are distinct if they involved not equivalent start states or not equivalent end states [5], and (2) crashes are distinct if they do not refer to the same type of unhandled exception or do not have the same stack trace [9, 15].

5.3 Experimental Procedure

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

5.3.1 App Testing. this step was conducted in two phases. In the former phase, three different testing processes were executed. In each process, all the objects were tested by three runs of the ALARic tool configured to plan and execute only one Lifecycle Event Sequence type, *i.e.* DOC, BF, and STAI. We did three runs for each configuration to mitigate the non determinism of the apps and of the random exploration techniques [7]. Each run lasted an hour. A total of nine one-hour testing runs for each app were performed.

In the latter phase, we tested the object apps using the Monkey tool, in order to compare the tools effectiveness. Monkey is an automated testing tool for Android apps, belonging to the Android SDK. This tool adopts a random testing approach, which sends a random stream of UI and system-level events to the app under test.

We performed a testing process where nine one-hour Monkey testing runs were executed for each app. We executed nine Monkey runs as with ALARic in order to ensure a fair comparison among the tools.

In this phase, we set the maximum verbosity level of the Monkey tool in order to produce a more accurate and rich output containing information about the seeded events and the detected crashes.

All the testing processes have been performed on the same testing infrastructure which consists of a desktop PC having an Intel(R) Core(TM) i7 4790@3.60GHz processor and 8 GB of RAM, running

a standard Nexus 5 AVD with Android 6 (API 23). The host PC was equipped with the Ubuntu OS, version 16.04. In order to assure that each run was executed in the same conditions, all the runs were executed on AVDs created from scratch.

5.3.2 Data Collection & Validation. At the end of the testing processes we gathered all the reports produced by the considered tools and recruited a team composed by two Ph.D. students and a Postdoctoral Researcher having knowledge on Software Debugging and Android Testing.

The team was asked to analyze the failures exposed by the tools and to validate them. To this aim the team examined the reports produced by the considered tools to identify the distinct failures exposed by each tool. Unlike ALARic, Monkey does not offer a detailed description of each failure exposed at runtime [15]. To extract the crashes detected by Monkey from the output it generated, the team manually inspected these files to find the exception stack traces instances and the events that led to them. Then they exploited the information contained in the issue reports to reproduce all the distinct failures. To this aim they manually tried to reproduce the reported issues on a real LG G4 H815 device equipped with Android 6.0. In this way we can consider only real failures caused by incorrect application logic and discard the ones caused by issues tied to the testing infrastructure and the virtual device. As regards the GUI failures, they also assessed whether each failure was actually the manifestation of an incorrect GUI state rather than an intended behavior of the application, *e.g.* a timer that continues to count down or a news feed that adds new elements may cause the GUI state to be different after the execution of a Lifecycle Event Sequence. To guarantee that the issues were actually tied to the Activity lifecycle, the team performed a debugging activity to verify that the issues were a manifestation of faults that are exercised by executing the Activity lifecycle. The validated distinct failures have been used to calculate the values of the metrics.

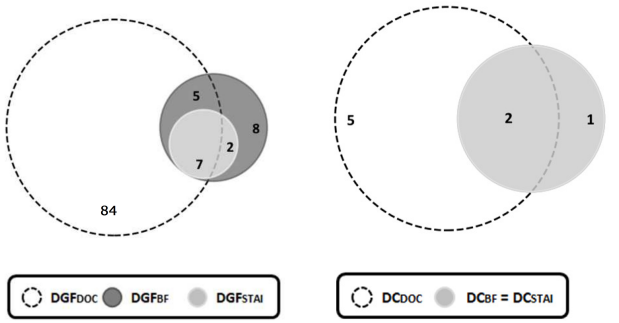
5.4 Results and Analysis

Table 2 reports, for each app, the total number of GUI failures and crashes that have been found by ALARic and validated by the team, grouped by the Lifecycle Event Sequence type that triggered them.

Table 3 shows for each app the number of total crashes tied to Lifecycle Event Sequences detected by ALARic and Monkey, respectively. We did not compare the results regarding GUI failures since Monkey is not able to detect them.

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

We analyzed the relations among the sets of issues exposed by each of the three considered Lifecycle Event Sequences. As shown by the Venn Diagrams reported in Fig. 6, in some cases the same issue was exposed by more than one type of Lifecycle Event Sequence, whereas other issues were triggered by only one Lifecycle Event Sequence type.



(a) GUI Failures triggered by the Lifecycle Events

(b) Crashes triggered by the Lifecycle Events

Figure 6: Issues detected by ALARic

As Fig. 6(a) shows, 7 out of the 106 GUI failures detected by ALARic were found by all the three considered Lifecycle Event Sequences. Among the 96 GUI failures triggered by DOC, 84 were not found by BF and STAI. 8 GUI failures have been triggered only by BF. Fig. 6(b) illustrates that only the 2 crashes exposed by *A*9 were triggered by all the three considered Lifecycle Event Sequences. 5 out of 8 crashes were triggered only by DOC. Instead, the crash exposed by *A*11 was triggered by BF and STAI but not by DOC.

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

On the basis of the obtained results we were able to answer the first research question RQ_1 and conclude that:

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

Regarding the comparison between ALARic and Monkey, the data in Table 3 shows that for 6 out of 7 apps ALARic was able to find more crashes tied to the Activity lifecycle than Monkey. In *A*4 both the tools exposed the same crash. Moreover, both the tools detected an additional crash in *A*9 that was not tied to the Activity lifecycle. To better understand this result we analyzed in detail the reports produced by Monkey. It was able to seed events that exercise the Activity lifecycle, e.g. orientation changes, back button, but it applied them without a proper strategy, failing in discovering several issues tied to the Activity lifecycle that were found by ALARic, instead. On the basis of this data we could answer to RQ_2 concluding that:

Table 2: Experimental Results

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

Table 3: Experimental Comparison

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

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

5.5 Lesson Learned

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

As an example, the crash found in *A*11 occurs when the `onSaveInstanceState()` callback method of the `EditProjectActivity` is called, but its cause is located inside the `LinkEditWidget` class that defines a custom GUI object. The programmer indeed overrode the `onSaveInstanceState()` callback method to save at runtime the state of the `LinkEditWidget` custom GUI object contained in the `EditProjectActivity` Activity. To this aim, the programmer correctly serialized the `LinkEditWidget` objects and properly implemented the `Serializable` interface in the class that defines the `LinkEditWidget` object. However, the user-defined `LinkEditWidget` contains `android.widget.Spinner` GUI components that do not implement the `Serializable` interface. Therefore a `java.io.NotSerializableException` is thrown at runtime when the lifecycle of the `EditProjectActivity` Activity is exercised. Another example is related to a failure that regarded 57 out of the 106 GUI failures detected by ALARic. It involved `Dialog` objects disappearing from the GUI after the execution of a Lifecycle Event Sequence. This failure affected most of the considered apps since 12 out of 15 apps exposed it. A `Dialog` is a small window that does not fill the screen and is normally used for modal events that require users to take an action before they can proceed. In most cases, the fault causing these failures has been localized in objects calling directly the public `show` method offered by the `Dialog` or the `AlertDialog Builder` classes to display a `Dialog` on screen. This will correctly pop up the dialog on the screen but the dialog will disappear when the Activity is destroyed and recreated in its lifecycle. Instead, Android guidelines explicitly prescribe that the control of a dialog GUI object (deciding when to show, hide, dismiss it) should be managed by the `DialogFragment` class, which ensures a correct handling of Lifecycle Event Sequences[13].

Thanks to this analysis, we learned two lessons that could be useful for Android developers. The former lesson is that they should correctly use the Android framework components since they may cause inconsistencies in the app behavior at runtime when Lifecycle Event Sequences occur. The latter is that they should look for faults that may affect the lifecycle of the Activities also outside the methods that override the lifecycle callbacks.

5.6 Threats to Validity

This section discusses the threats that could affect the validity of the results obtained in the study [20].

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

[13] https://developer.android.com/reference/android/app/DialogFragment.html

5.6.2 External validity. We are aware that the small sample of considered Android apps may affect the generalizability of our experimental results and we intend to confirm our findings in the future by performing a wider experimentation involving a larger number of apps.

6 RELATED WORK

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

Franke *et al.* [8] presented a unit testing based approach for testing conformance of app lifecycle. Their approach considers Activities as units to be tested. Lifecycle-dependent properties have to be manually extracted from functional requirement specification and the Activity lifecycle callback methods are used to test such properties exploiting assertions. Unlike our fully-automated black-box technique, this approach heavily relies on manual effort to extract requirements and to define assertion-based unit test cases and requires the availability of the app source code.

The work of Zaeem *et al.* [21] is based on the intuition that different mobile apps and platforms share a set of features referred to as User-Interaction Features and that there is a common sense of expectation of how an app should respond to these features. Among these features they considered also the triggering of the key lifecycle loops. They propose an automated model-driven test suite generation approach and implement it in QUANTUM, a framework that automatically generates a test suite to test the user-interaction features of a given app leveraging app agnostic test oracles based on the GUI states exposed by the app. Differently from our approach, QUANTUM needs a prior knowledge of the app under test since it requires a user-generated app GUI model as input.

Adamsen *et al.* [1] proposed a tool named `THOR` that systematically amplifies test cases by injecting *neutral* event sequences that should not affect the functionality of the app under test and the output of the original test cases. They focus on event sequences that are usually neglected in traditional testing approaches, including the ones that exercise the key lifecycle loops. THOR leverages existing test cases. Instead, our approach does not require existing testing artifacts.

Shan et al. [17] focused on a specific fault class due to the incorrect handling of the data that should be preserved when the key Activity loops are exercised. They named *KR errors* the failures caused by these faults. These authors proposed an automated static analysis technique for finding KR errors. They also designed a tool that generates a sequence of input events that lead to the app state where the KR error manifests. Unlike our work, this solution needs app modification to verify the failures tied to Activity lifecycle by tracking app fields and dumping GUI states in the Activity lifecycle callback methods.

G. Hu *et al.* [9] introduced `AppDoctor`, a testing system able to perform a quick dynamic analysis of the app under test that aims at revealing app crashes. Our testing approach is able to detect also GUI failures. Their proposed app analysis, called *approximate execution*, is faster than real execution since it exercises an app by invoking directly event handlers. Our technique instead triggers real events because they represent better real user interactions. Since their approximation may introduce several false positives, AppDoctor automatically tries to verify the detected bugs by reproducing them using real events. Like us, they pointed out the relevance of exercising the Activity lifecycle in mobile testing. Therefore they introduced approximations of lifecycle events among the events supported by AppDoctor.

Moran *el al.* [15] designed Crashscope, a fully automated testing approach for discovering, reporting, and reproducing Android app crashes. They also propose a fully automated black-box testing approach but they focus only on a specific failure type, i.e. app crashes, whereas our approach is able to find also GUI failures. They identify the double orientation change lifecycle event as a major source of crashes and thus their automated app exploration performs a double rotation each time they encounter a rotatable Activity. Whereas, our approach is able to perform three different types of lifecycle events able to cover the three key Activity lifecycle loops. Moreover, our exploration strategy performs a lifecycle event each time it encounters a GUI state never encountered before during the exploration.

Jun *et al.* [10] proposed LeakDAF, a fully automated testing approach for detecting memory leaks. Like our work, they propose a testing technique targeting Android app components lifecycle conformance that exploits an automated app exploration technique and does not need app modification or manual interaction. They test the apps with two lifecycle events that exercise only the Entire Lifecycle loop. Whereas, our approach studies the effect of events that exercise also the Visible Lifecycle and Foreground Lifecycle loops. They aim at detecting a specific memory leak type, i.e. the leakage of Activity and Fragment Android app components. Instead, we aim at proposing a testing technique able to detect different types of issues related to the lifecycle of Android app Activities.

7 CONCLUSIONS & FUTURE WORK

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

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

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

REFERENCES

[1] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. 2015. Systematic Execution of Android Test Suites in Adverse Conditions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, Baltimore, MD, USA, 83–93. http://doi.acm.org/10.1145/2771783.2771786

[2] Domenico Amalfitano, Nicola Amatucci, Atif M. Memon, Porfirio Tramontana, and Anna Rita Fasolino. 2017. A general framework for comparing automatic testing techniques of Android mobile apps. *Journal of Systems and Software* 125 (2017), 322–343.

[3] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, and Bryan Robbins. 2013. Testing Android Mobile Applications: Challenges, Strategies, and Approaches. *Advances in Computers* 89 (2013), 1–52.

[4] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon. 2015. MobiGUITAR: Automated Model-Based Testing of Mobile Apps. *IEEE Software* 32, 5 (Sept 2015), 53–59.

[5] Domenico Amalfitano, Vincenzo Riccio, Ana C. R. Paiva, and Anna Rita Fasolino. 2018. Why does the orientation change mess up my Android application? From GUI failures to code faults. *Softw. Test., Verif. Reliab.* 28, 1 (2018).

[6] F. Belli, M. Beyazit, and A. Memon. 2012. Testing is an Event-Centric Activity. In *Software Security and Reliability Companion (SERE-C), 2012 IEEE Sixth International Conference on*. 198–206.

[7] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet? (E). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE '15)*. IEEE Computer Society, Washington, DC, USA, 429–440.

[8] D. Franke, S. Kowalewski, C. Weise, and N. Prakobkosol. 2012. Testing Conformance of Life Cycle Dependent Properties of Mobile Applications. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. 241–250.

[9] Gang Hu, Xinhao Yuan, Yang Tang, and Junfeng Yang. 2014. Efficiently, Effectively Detecting Mobile App Bugs with AppDoctor. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM, New York, NY, USA, Article 18, 15 pages.

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

[11] V. Lelli, A. Blouin, and B. Baudry. 2015. Classifying and Qualifying GUI Defects. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. 1–10.

[12] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: Segmented Evolutionary Testing of Android Apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 599–609.

[13] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective Automated Testing for Android Applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, New York, NY, USA, 94–105.

[14] Abel Méndez-Porras, Giovanni Méndez-Marín, Alberto Tablada-Rojas, Mario Nieto Hidalgo, Juan Manuel García-Chamizo, Marcelo Jenkins, and Alexandra Martínez. 2017. A distributed bug analyzer based on user-interaction features for mobile apps. *Journal of Ambient Intelligence and Humanized Computing* 8, 4 (01 Aug 2017), 579–591.

[15] K. Moran, M. Linares-Vasquez, C. Bernal-Cardenas, C. Vendome, and D. Poshyvanyk. 2016. Automatically Discovering, Reporting and Reproducing Android Application Crashes. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 33–44.

[16] Henry Muccini, Antonio di Francesco, and Patrizio Esposito. 2012. Software testing of mobile applications: Challenges and future research directions. In *Automation of Software Test (AST), 2012 7th International Workshop on*. IEEE, Zurich, Switzerland, 29–35.

[17] Zhiyong Shan, Tanzirul Azim, and Iulian Neamtiu. 2016. Finding Resume and Restart Errors in Android Applications. *SIGPLAN Not.* 51, 10 (Oct. 2016), 864–880.

[18] Statista. 2016. Number of smartphone users worldwide from 2014 to 2019 (in millions). https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/

[19] Statista. 2017. Global market share held by the leading smartphone operating systems in sales to end users from 1st quarter 2009 to 1st quarter 2017. https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/

[20] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. 2012. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated.

[21] Razieh Nokhbeh Zaeem, Mukul R. Prasad, and Sarfraz Khurshid. 2014. Automated Generation of Oracles for Testing User-Interaction Features of Mobile Apps. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation (ICST '14)*. IEEE Computer Society, Washington, DC, USA, 183–192.

[22] Samer Zein, Norsaremah Salleh, and John Grundy. 2016. A Systematic Mapping Study of Mobile Application Testing Techniques. *J. Syst. Softw.* 117, C (July 2016), 334–356.

[23] Xia Zeng, Dengfeng Li, Wujie Zheng, Fan Xia, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. 2016. Automated Test Input Generation for Android: Are We Really There Yet in an Industrial Case?. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 987–992.

[24] Haibing Zheng, Dengfeng Li, Beihai Liang, Xia Zeng, Wujie Zheng, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. 2017. Automated Test Input Generation for Android: Towards Getting There in an Industrial Case. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP '17)*. IEEE Press, Piscataway, NJ, USA, 253–262.