Iranian Journal of Science and Technology, Transactions of Electrical Engineering (2020) 44:1013–1029
https://doi.org/10.1007/s40998-019-00270-y

RESEARCH PAPER

An Empirical Study on the Effectiveness of Monkey Testing for Android Applications

Samad Paydar[1]

Received: 24 July 2018 / Accepted: 14 September 2019 / Published online: 23 September 2019

Abstract
Android application development has attracted the attention of many software developers as a mainstream software platform. Despite its interesting characteristics, there are a number of issues that result in developing applications with poor performance. Hence, it is very crucial to evaluate quality of Android applications, specially their robustness and responsiveness. Monkey testing is a traditional technique used for testing these aspects in applications. In this paper, two experimental studies are conducted to investigate how effective monkey testing is in revealing robustness and responsiveness faults in Android applications. The results demonstrate that monkey testing of Android applications for finding robustness and responsiveness errors is very effective and highly recommended. Further, the Android applications published in Iranian app stores have turned out to be more vulnerable to these types of errors, compared to the applications published in international app stores. This means that robustness and responsiveness testing needs to be taken into consideration more seriously by the Iranian developers. The experimental results also demonstrate that the default monkey tool available in Android Studio IDE has a very low focus, i.e., 5.5%, and is much less sensitive to the responsiveness faults that cause delays less than 5 s.

Keywords Android · Monkey testing · Crash · Responsiveness · Robustness

1 Introduction

Android has become a mainstream software platform which has attracted the attention of many software developers (VisionMobile 2016). Despite the interesting characteristics of this platform which provides new opportunities for both developers and users, there are some issues that result in the development of poor performance applications: for instance, fast-changing API (Vásquez et al. 2013; McDonnell et al. 2013), lack of precise knowledge of the developers about the underlying mechanisms like application of lifecycle management or permission management mechanisms (Amalfitano et al. 2012) and the fact that many applications are network-centric and need to work under a wide range of hardware/software execution profiles (Wei et al. 2012). As a result, many Android applications have poor quality, specifically in terms of robustness, resulting in frequent application crashes (Canfora et al. 2016). Another aspect of application performance is its responsiveness, and it is required to evaluate and test the responsiveness of the Android applications. If an application fails to respond in less than a predefined time (the default is 5 s) to the events that the user generates on the application's user interface (UI), the Android system considers the application to be not responding and shows an Application Not Responding (ANR) error to the user, asking the user to either wait for the program or stop it forcefully.[1] The ANR errors are traditionally caused by a time-consuming operation being executed on the main thread of the application.

One approach to evaluating robustness and responsiveness of Android applications is to employ monkey testing technique where a stream of random events is automatically generated on the user interface (UI) of the application with the goal of making the application crash. The longer

✉ Samad Paydar
s-paydar@um.ac.ir

[1] Computer Engineering Department, Ferdowsi University of Mashhad, Mashhad, Iran

[1] https://developer.android.com/training/articles/perf-anr.html.

the application resists against the monkey events, the greater the assurance that the application would not crash under real-world circumstances, i.e., when it is used by the users.

Due to the importance of monkey testing for Android applications, Google has equipped its software development kit (SDK) with two facilities, the Monkey[2] and the Monkey Runner[3] tools. The first tool is a dumb monkey that performs random interactions with an Android application that is launched in an emulator. However, the second tool requires a test script that explicitly defines the stream of events to be generated. This script needs to be created manually by a human tester using a specific syntax. So it is difficult and time-consuming to manually create scripts for effectively exploring the UI and testing the application. Hence, in practice many developers rely on the dumb monkey for monkey testing of their applications. This raises the question of how effective the Android Monkey is in testing Android applications.

In this paper, we have conducted two experiments to (1) investigate the necessity of monkey testing on Android applications and (2) to evaluate the effectiveness of Android Studio monkey in terms of being able to reveal errors that make the application crash.

The rest of the paper is organized as follows. Section 2 briefly reviews the related works, and Sect. 3 discusses the experimental studies. Finally, Sect. 4 concludes the paper.

2 Related Work

Parallel to the increasing interest of software developers in Android application development, testing Android applications has also started to attract more attention from researchers and practitioners. Consequently, there are many works which focus on different aspects or types of testing for Android applications, including GUI testing (Amalfitano et al. 2012), performance testing (Liu et al. 2014), responsiveness testing (Kang et al. 2016), mutation testing (Deng et al. 2017a, b) and visual GUI testing (Ardito et al. 2019). In this section, the works that are more related to the current paper are briefly reviewed.

A review of the existing techniques for automated testing of Android applications is presented in Kong et al. (2019), where a set of 103 relevant research papers published before 2016 are analyzed. The authors have discussed these works based on their test objective (e.g., bug/defect detection or energy consumption analysis), test target [e.g., GUI/Event or inter-component communication (ICC)], test levels (e.g., unit testing or system testing) and test techniques (e.g., mutation testing or random testing). Further, the authors have made different conclusions including the fact that bug/defect detection and GUI testing are, respectively, the most common test objective and test target. The authors have also identified different weaknesses in the evaluation of the proposed techniques in the literature; for instance, most works have performed their experiments on datasets with no more than 100 apps, and the median is eight apps. Further, reproducibility is a major concern regarding the evaluations of the existing works. Similarly, a systematic mapping study is conducted in Tramontana et al. (2019) on the functional testing of mobile applications.

Choudhary et al. (2015) have conducted a thorough study of the existing test input generation tools and techniques for Android applications, to identify the weaknesses and potentials of each one. The authors have concluded that Android Monkey generally outperforms the other existing tools, e.g., in terms of code coverage and ease of use. On the other hand, a study in Vásquez et al. (2017) concludes that monkey testing is not yet a common practice between Android developers. Despite the advantages of Android Monkey, it also has noticeable weaknesses. Due to the importance and advantage of monkey testing for Android applications, there is a line of research dedicated to development of new monkey testing or random testing techniques for these applications. Next, we briefly discuss the more important works in this group.

To investigate potentials and limitations of monkey in industrial settings, Zeng et al. (2016) have studied the performance of Android Monkey on the popular messaging application, i.e., WeChat. The results have revealed noticeable limitations that resulted from the random exploration strategy of monkey. Specifically, monkey has covered only 19.5% of the code lines and 10.3% activities of WeChat. Zeng et al. have recognized two root causes for this exploration inefficiency: (1) widget obliviousness: monkey does not know the location of the widgets and hence generates many events that have no effect and (2) state obliviousness: monkey does not remember visited states of the application, and hence, it generates events that repeatedly change the state of the application between a few states, reducing the chance of visiting new states. In order to address this limitation, Zeng et al. have then developed a new monkey that has widget awareness, through considering the events supported by widgets and allowing the user to specify weights for different types of events on different types of widgets. This new monkey has state awareness since it monitors application states and focuses on generating events that change the state. Through these improvements, the monkey has achieved 30.6% line coverage and 28.7% activity coverage on WeChat. In Sun

[2] https://developer.android.com/studio/test/monkey.html.

[3] https://developer.android.com/studio/test/monkeyrunner/index.html.

et al. (2016), a technique is introduced to improve monkey testing by employing visual analysis techniques to identify parts of the GUI that are interactable, hence addressing the widget obliviousness issue.

Since Android Monkey is a dumb monkey which does not have any specific knowledge about AUT, some researchers have introduced new random testing techniques to address the limitations of the monkey. For instance, Machiry et al. have proposed Dynodroid (Machiry et al. 2013) which enhances random exploration strategy of monkey by incorporating a frequency strategy which considers the frequency of generating each event type in deciding the type of the next event. It also provides a biased strategy to take into consideration the relevancy of the events to the context. The evaluation results have shown that Dynodroid and Monkey have had, respectively, 55% and 53% code coverage, but Monkey has achieved this by creating, on average, 20 $\times$ more events than Dynodroid. Song et al. (2017) proposed a new approach for GUI testing of Android applications. The main idea of this approach, named EHBDroid, is to directly call the event handler methods, instead of indirectly invoking them through interactions with the GUI of the application. This provides two main benefits: (1) it improves effectiveness by directly triggering events that might be hard to fire, e.g., require specific input data, by the monkey from the GUI layer, and (2) it increases efficiency since the overhead imposed by the GUI and its underlying message passing mechanisms is eliminated. This is also due to the systematic invocation of the event handlers which prevents repeatedly invoking the same event handler. EHBDroid is compared with Android Monkey and Dynodroid (Machiry et al. 2013), and the results have demonstrated that EHBDroid outperforms both tools in terms of efficiency and statement coverage. Further, EHBDroid has been successful in finding 12 new bugs.

In another work (Wetzlmaier and Ramler 2017), a hybrid technique is introduced which seeks to enhance an existing GUI regression test suite by injecting interactions randomly generated by the monkey. This is intended to improve bug-finding capability of the regression tests, which in the lack of the random interactions always execute the same sequences of steps and just confirm that previous behaviors are not altered, but do not challenge newly added behaviors. The evaluation results have shown that this hybrid technique improves application window coverage by 23.6% and code coverage by 12.9%. It has also found three new bugs.

Wetzlmaier et al. (2016) presented a monkey testing framework which through its reusable components and extension points, enables development of customized monkeys for different application scenarios or technical requirements, e.g., requirements of the nightly build integration. Further, to demonstrate applicability of this framework, it is used to create a monkey for testing an industrial application. The results have acknowledged the adaptability and extendibility of the proposed framework. The authors in Haoyin (2017) have developed a new monkey testing technique for Android applications. This technique includes two main novelties: (1) it supports a wider range of user and system events by dynamic analysis of the application's GUI and generates widget-aware events, and (2) it dynamically builds a GUI transition graph and uses a biased random walk algorithm to explore this graph for the purpose of event generation. The evaluation results have demonstrated that compared to Android Monkey, the proposed technique is more efficient, in terms of both the execution time and the number of events required to find the first failure of AUT.

In Mao et al. (2016), Sapienz is introduced as a method for monkey testing of Android applications which employs search-based optimization techniques to find the test sequences with minimum length and maximum coverage for testing Android applications. The proposed method has outperformed Dynodroid and Android Monkey in terms of code coverage and fault revealing.

Another line of research focuses on employing model-based techniques for random testing of Android applications. For instance, Amalfitano et al. have conducted crash testing and regression testing based on models that are constructed by crawling the application GUI (Amalfitano et al. 2011). Crash testing is performed by creating test cases from the possible paths on the application GUI model with the goal of finding execution paths that result in uncaught exceptions. Regression testing is also accomplished by adding assertions to the source code, running test cases and finding points where an assertion is failed. As another model-based technique, Baek et al. have proposed a method that creates a GUI model based on the notion of abstraction level, which is determined by comparison criteria that can be in five levels (package, activity, layout widgets, executable widgets and content). This model is then used to generate test cases for the purpose of detecting runtime errors (Baek and Bae 2016). Another model-based technique is FSMdroid (Su 2016) which initially models the application as a stochastic finite state machine and then iteratively mutates this model by a Markov Chain Monte Carlo sampling technique to improve code coverage through increasing diversity of the generated event sequences.

Since the number of available apps is increasing rapidly, if the user is not satisfied with his experience of using a specific app, he/she might quickly switch to another app that provides the same functionality. As a result, the topic of user experience has gained huge importance in the development of Android applications. Among the factors that affect user experience is the responsiveness of the application. Kang et al. (2016) have conducted an experience report on users' tolerance in dealing with poor responsiveness. Their report shows that after a 500 ms latency in UI without any explicit feedback, the users perceive an application having poor responsiveness. This emphasizes the need for taking responsiveness testing of Android applications into account. As a result, there is a group of works that seek to evaluate responsiveness of an application or identify the causes of its low responsiveness. In contrast to random testing techniques that use dynamic analysis, these works mainly employ static analysis.

For instance, Ongkosit and Takada (2014) propose a method that uses static analysis to automatically find the methods that execute heavy and time-consuming operations and hence can cause responsiveness issues. Through analysis of the official documents of Android API, the authors have identified four groups of Android APIs as containing heavy operations: network operations, database interactions, I/O operations and bitmap processing. From each group, a set of methods is manually identified as heavy operations. The authors have implemented a tool that examines an application to find the places where one of these methods is called in the main thread of the application. Those places are reported as sensitive areas that need to be further analyzed by the human expert. It is worth noting that it is possible that a piece of code does not call any of the so-called heavy methods, but it performs time-consuming computations and hence it can introduce performance errors, e.g., ANR error or application lags. This type of responsiveness error is not considered in the method proposed in Ongkosit and Takada (2014). In a similar study, Yang et al. (2013) have used the same set of heavy methods and have introduced a fault-based technique for injecting delays after each of these API calls in order to test their effects on responsiveness and GUI lagging. The results demonstrate that their technique, called test amplification, can effectively improve response times for GUI events and detect responsiveness problems. Specifically, it has identified 61 responsiveness faults in eight open-source applications.

Gomez et al. have proposed a metric-based approach for detecting possible degradation in the UI responsiveness of an application among different versions (Gómez et al. 2016). Wang et al. have conducted a study with resource amplification, trying to characterize responsiveness of Android applications under API calls such as bitmap processing, shared preference operations and SQLite database operations (Wang and Rountev 2016). They have proposed a technique for amplifying processing whenever one of these API calls happens. The purpose is to see whether code transformation is necessary for better performance or not. The experimental results have shown that it is not necessary to simply remove every instance of these API calls from the application's main thread, since it increases the code complexity without having noticeable effect on responsiveness in real-world scenarios. For instance, not every SharedPreferences API is an issue unless the corresponding store has too many stored pairs, something that might not occur according to the application's logic. Therefore, such modifications need to be performed based on precise profiling information.

Thiagarajan (2016) has considered responsiveness bugs to be any operation that takes more than 100 ms. In addition, a technique is proposed for detecting those bugs. First, every event in the UI main thread is recorded along with its start and end time stamp. Then, by monitoring the event queue, the events that execution of their handler is taking more than 100 ms are identified and the corresponding stack trace is reported.

Liu et al. (2014) have studied the performance of bug features and identified a number of common bug patterns including lengthy operation in the main thread, wasted computation for invisible GUI and frequently invoked heavy-weight callbacks. They have also proposed a static analysis tool for detecting (1) lengthy operations in the main thread and (2) violation of the ViewHolder pattern as a traditional cause of performance issues in applications that display a probably long list of items.

From the review of the related works, it can be concluded that there is still room for improving testing techniques for Android applications. Kochhar et al. (2015) have examined the culture of testing among mobile application developers, concluding that many Android applications are poorly tested and in addition, many Android developers still depend on monkey for testing their applications. This can be attributed to the fact that the cost of applying monkey is very low, since it does not require specific configuration, preparation or learning. Further, it is application-independent and can be run on different applications. More importantly, it is integrated with Android Studio which is the official IDE for Android development. Hence, we conclude that the other monkey testing techniques introduced in the literature have a long way to become established in the field and frequently used in the industry. Consequently, it is necessary to conduct more investigations on the effectiveness of Android Monkey, especially in terms of its capability to reveal responsiveness and robustness faults. The purpose is to provide more precise evidence on the benefits and weaknesses of the monkey and the ways in which it needs to be improved. This can be helpful in motivating practitioners to employ monkey testing for their Android applications.

This paper conducts an experimental study to address this issue. The most related work to the current paper is Patel et al. (2018), where the authors have studied the effectiveness of Android Monkey on 15 widely used apps and 65 open-source apps. They have made these conclusions: (1) on average, the monkey takes about 85 s to make the application under test crash, (2) fine-tuning of the event distribution parameters of the monkey does not yield noticeable improvement on the coverage, (3) there is a strong positive correlation between coarse-grained coverage, i.e., class or method coverage, and fine-grained coverage, i.e., block of line coverage, (4) the throttle parameter, i.e., the delay between the consecutive events generated by the monkey, does not have a noticeable effect on coverage and finally (5) the coverage of the monkey is about 2–4% lower than the coverage of a 5-min session of manual exploration of the application under test.

Compared to Patel et al. (2018), we have performed a more thorough analysis on two larger sets of Android apps, and we have investigated Android effectiveness in more detail, e.g., considering application crashes and ANRs separately and analyzing the crash causes. In addition, we have employed a skeleton-based approach to generate specific Android applications with purposefully injected faults to provide a better understanding of the monkey behavior, e.g., its focus on the application under test, and its location distribution of its events. This fault injection-based analysis has not been considered in previous works, and we believe it can be employed for more reliable analysis of the monkey. We think that the results of these investigations can help researchers by providing concrete evidence on the specific weaknesses of Android Monkey that can be addressed in new techniques. Additionally, the results will be interesting for the practitioners and help them have more realistic expectations and conclusions about their use of Android Monkey.

3 Experimental Study

In this section, we discuss two experiments conducted to investigate whether monkey testing is effective in revealing robustness and responsiveness errors in Android applications. In the first experiment, we focus on monkey testing of real-world Android applications collected from two Android app stores: F-Droid[4] and CafeBazaar.[5] In the second experiment, we evaluate effectiveness of monkey testing on a benchmark of Android applications specifically designed for the purpose of evaluating monkey testing.

Next, these two experiments are described.

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

[5] https://cafebazaar.ir/.

3.1 Experiment 1: Real-World Applications

The main goal of this experiment is to investigate whether monkey testing is able to reveal robustness and responsiveness issues in Android applications published on well-known online app stores. For monkey testing, we use Android Monkey provided by the Android SDK. Further, we consider application crashes and ANR errors, respectively, as evidence of some robustness and responsiveness issues in the application under test. As the target app stores, we have considered F-Droid as a repository of open-source Android applications and CafeBazaar as the leading app store hosting Iranian Android applications. The reason for using the second app store is that it would provide interesting information for determining the status of Iranian Android applications in comparison with international open-source applications. Hence, the result of the experiment is expected to be helpful for both Iranian and international Android developers.

Considering the goal described above, the first experiment is designed to address the following research question:

RQ1.1 Is monkey testing effective in revealing responsiveness and robustness errors of real-world Android applications?

Next, we describe the datasets prepared in this study.

3.1.1 Datasets

For the purpose of this experiment, we have prepared two datasets. The first dataset includes 150 Android applications downloaded in APK format from F-Droid an online catalogue of free and open-source Android applications. It is worth noting that this dataset initially included about 290 applications, i.e., the top 20 applications from each F-Droid category with the most downloads. However, after initial analysis, the number of applications was reduced to 150 by removing the applications for which monkey testing was considered to be not applicable without manual intervention. For instance, some applications need log-in and without a successful log-in, the application functionalities are not provided for the user. Since we do not intend to provide any hint or extra information for the monkey, e.g., a valid account to make log-in successful, we have ignored such applications. Further, there have been some other applications that are ignored due to technical difficulties like the applications that were not successfully installed on the target emulator. The second dataset includes 150 Iranian Android applications, in APK format, downloaded from CafeBazaar.

Fig. 1 Test script used in the experiments

```
1. Run Android emulator.
2. Install the target application on the emulator.
3. Run monkey with n events.
4. Uninstall the application from the emulator.
5. Shut down the emulator.
```

3.1.2 Test Process

For the purpose of this experiment, a test script is created for monkey testing on the applications in the datasets. An abstract representation of this test script, which is actually a Windows Batch file, is shown in Fig. 1. It simply runs a pre-configured emulator and installs the application under test (AUT) on the emulator. Then, it runs the monkey tool on AUT to perform monkey testing by injecting n events on the application. When the execution of the test script is finished, it creates two log files: (1) the output of the application under test which is logged by Android logging facility, i.e., LogCat tool, and (2) the output of the monkey stored in a separate log file.

In this experiment, we have used $n = 10$ K as the number of events injected during monkey testing. Further, the test script is executed 100 times on each target application to increase reliability of the results.

3.1.3 Metrics

Since the goal of the experiment is to investigate effectiveness of monkey testing for revealing robustness and responsiveness issues in Android applications, we have defined the following metrics to be used for analysis of the experiment results:

- CrashRate_n determines the percentage of the monkey testing sessions[6] with n events that have resulted in AUT having crashed, hence revealing a robustness error in AUT.
- ANRRate_n determines the percentage of the monkey testing sessions with n events that have resulted in an ANR error being issued, hence revealing a responsiveness issue in AUT.
- FailureRate_n determines the percentage of the monkey testing sessions with n events that have resulted in AUT having crashed or issued an ANR error, hence causing a failure in AUT being observed.
- CrashDelay_n determines the number of the first event in a monkey testing session with n events that has resulted in AUT having crashed.
- ANRDelay_n determines the number of the first event in a monkey session with n events that has resulted in AUT having issued an ANR error.
- FailureDelay_n determines the number of the first event in a monkey testing session with n events that has resulted in AUT having failed due to either a crash or an ANR error.

The first two metrics, i.e., CrashRate_n and ANRRate_n, show how frequently robustness and responsiveness errors of AUT are revealed by monkey testing. The second two metrics, i.e., CrashDelay_n and ANRDelay_n, measure how long it takes for the monkey testing to reveal the first responsiveness or robustness error in AUT.

3.1.4 Results

After executing the test process on the two application datasets, the resulting log files are processed to analyze the effectiveness of the monkey testing sessions by computing the metrics defined above. The results are shown in Tables 1 and 2. The most notable observations in Table 1 are:

- Sixty-seven percent of the applications in dataset1 and 52% of the applications in dataset2 have had $\text{CrashRate}_n = 0\%$, meaning that they have had no crash during the monkey testing sessions. This means that about one-third of the applications in dataset1 and about half of the applications in dataset2 have crashed in at least one monkey testing session. This means that monkey testing can be considered to be effective in detecting robustness issues in the applications. Further, it is observed that the applications in dataset2, i.e., Iranian Android applications, have been more vulnerable to crashes. This emphasizes the necessity of awareness of Iranian Android developers about the importance of more effectively testing their applications before release.

[6] We refer to each execution of the test script as a monkey testing session.

Table 1 Results of the first experiment with n = 10 K

Rate R	Dataset1			Dataset2		
	CrashRate_n Distribution (%)	ANRRate_n Distribution (%)	FailureRate_n Distribution (%)	CrashRate_n Distribution (%)	ANRRate_n Distribution (%)	FailureRate_n Distribution (%)
$R = 0$	67	73	49	52	48	15
$0 < R \leq 25$	9	23	22	22	28	32
$25 < R \leq 50$	6	3	7	5	11	17
$50 < R \leq 75$	5	0	8	4	7	10
$75 < R \leq 100$	13	1	14	17	6	26

Table 2 Results of the first experiment with n = 10K

Delay N	Dataset1			Dataset2		
	CrashDelay_n Distribution (%)	ANRDelay_n Distribution (%)	FailureDelay_n Distribution (%)	CrashDelay_n Distribution (%)	ANRDelay_n Distribution (%)	FailureDelay_n Distribution (%)
$0 < N \leq$ 2K	54	39	50	59	31	47
2 K $< N \leq$ 4 K	16	18	17	18	25	21
4 K $< N \leq$ 6 K	11	14	12	9	22	14
6 K $< N \leq$ 8 K	10	13	11	7	11	9
8 K $< N \leq$ 10 K	9	16	10	7	11	9

- About 9% of the applications in dataset1 and 22% of the applications in dataset2 have crashed in up to 25% of the monkey testing sessions.
- Considering the last two rows in Table 1, about 18% of the applications in dataset1 and about 21% of the applications in dataset2 have crashed in more than half of their monkey testing sessions. This again acknowledges the effectiveness of the monkey for crash testing of Android applications.
- Seventy-three percent of the applications in dataset1 have experienced no ANR issue during the monkey testing sessions. For the applications in dataset2, this is only 48%. This means that about 23% of the applications in dataset1 and about half of the applications in dataset2 have issued an ANR error in at least one of their monkey testing sessions. Hence, it can be concluded that the monkey has been effective in revealing responsiveness errors in the applications. Another observation is that the applications in dataset2, i.e., Iranian Android applications, have been much more vulnerable to responsiveness errors, compared to international applications.
- Comparing CrashRate_n and ANRRate_n, it is observed that the applications in dataset1 have experienced more crashes than ANR errors, but the applications in dataset2 have had more ANRs than crashes.
- Only about half of the applications in dataset1 have had FailureRate_n = 0%, meaning that about half of the applications have had at least one failure during the monkey testing sessions. For the applications in dataset2, only 15% of the applications have had FailureRate_n = 0%, meaning that 85% of the applications have had at least one failure.
- Considering the last two rows in Table 1, it is surprising that about 22% of the applications in dataset1 and 36% of the applications in the dataset2 have had failures in more than 50% of their monkey testing sessions.

In addition, the following observations in Table 2 are interesting:

- Considering the applications in dataset1, in 54% of the monkey testing sessions that have made AUT crash, the crash has happened during the first 2K events generated by the monkey. This means that for these applications, the monkey has revealed the issues quickly. For the applications in dataset2, this ratio is about 59%.
- Among the monkey testing sessions that have finished due to crash of the applications in dataset1, only in 9% of the cases, it has taken the monkey between 8K and 10K events to make the application crash. For applications in dataset2, this ratio is only 7%.
- Considering ANRDelay_n for applications in dataset1, it is observed that among the monkey testing sessions that

have finished due to an ANR error, in 39% of the sessions, the ANR has occurred during the first 2 K events generated by the monkey. For the applications in dataset2, this ratio is 31%.

- For 50% of the applications in dataset1 and 47% of the applications in dataset2, we have $\text{FailureDelay}_n \leq 2$ K, i.e., the failure happened during the first 2 K events generated by the monkey. This means that the monkey testing session does not necessarily need to be very long in order to be effective.

Based on the observations mentioned above, we give a positive answer to RQ1.1 and conclude that monkey testing is effective in revealing responsiveness and robustness errors in Android applications. Therefore, it is highly recommended to Android developers to apply monkey testing before releasing their applications.

In addition to evaluating the effectiveness of the monkey in crash testing of the target applications, it is interesting to identify the cause of the crashes in these applications through analysis of the log files. The results in terms of the top ten most frequent crash causes are presented in Table 3. As it is shown in this table, the two types of exceptions `NullPointerException` and `ActivityNotFoundException` are the two most frequent causes for crashes of the applications in the first dataset, respectively, responsible for 47% and 11% of the crashes. For dataset2, the first crash cause is Native Crash which has been responsible for 59% of the crashes. The next two crash causes in dataset2 are `NullPointerException` and `ActivityNotFoundException` These results confirm the results reported in other research, e.g., Mao et al. (2016), demonstrating that `NullPointerException` is the most frequently occurring exception in Android applications.

Another important observation is that while NativeCrash is responsible for only 1% of the crashes in the application of dataset1, it is the cause of 59% of the crashes in dataset2. This can be concluded as NativeCrash is noticeably more frequent in Iranian Android applications, compared to international Android applications. Further, it is worth noting that among the exception types that have caused the applications crash, there are only two Android-specific exception types, i.e., `ActivityNotFoundException` and `Resources$NotFoundException`. The rest of the exception types are generic Java exceptions.

To sum up, the results of the first experiment demonstrate that even not very long monkey testing sessions are effective in revealing potential robustness and responsiveness issues in Android applications. Hence, considering the very low technical and setup cost for running the monkey on an Android application, it is highly recommended that Android developers take into consideration the monkey

Table 3 Top ten most frequent crash causes in the first experiment

Dataset1		Dataset2	
Crash cause	Freq. (%)	Crash cause	Freq. (%)
`java.lang.NullPointerException`	47	Native crash	59
`android.content.ActivityNotFoundException`	11	`java.lang.NullPointerException`	21
`java.lang.ArrayIndexOutOfBoundsException`	11	`android.content.ActivityNotFoundException`	5
`java.lang.IllegalStateException`	8	`java.lang.IllegalArgumentException`	5
`android.content.res.Resources$NotFoundException`	4	java.lang.ArrayIndexOutOfBoundsException	3
`java.lang.IndexOutOfBoundsException`	4	`java.lang.IllegalStateException`	3
`java.lang.ClassCastException`	3	`java.lang.IndexOutOfBoundsException`	1
`java.util.NoSuchElementException`	2	`java.lang.RuntimeException`	1
Native crash	2	`java.lang.InternalError`	1
`java.lang.NumberFormatException`	1	`java.util.concurrent.RejectedExecutionException`	1

testing of their applications before their release. In addition, while this experiment has shown that the monkey is able to find responsiveness or robustness issues in real-world applications, it is not possible to determine what percentage of the existing issues it has detected, since the actual errors and vulnerabilities of the applications are not known. Therefore, we have conducted another experiment based on a set of applications that are specifically developed for the purpose of evaluating the monkey. The interesting point about these applications is that they contain intentionally injected faults, and hence, they enable more precise analysis of the effectiveness of the monkey. The next section describes this second experiment.

3.2 Experiment II: Fault-Injected Applications

The goal of this experiment is to investigate how effective monkey testing is in revealing robustness and responsiveness errors injected in an Android application. The idea is that a fault-injected application, where the errors are known a priori, provides more opportunity, compared to a real-world application, for precisely evaluating the effectiveness of the monkey. Based on this idea, we have created an application skeleton to enable instantiation of concrete Android applications with *Crash Points* or *ANR Points*. We define a Crash Point and an ANR Point as a reachable and visible UI element in which the execution of its associated *onClick* handler causes the application to, respectively, crash or issue an ANR error. By reachable, we mean that it is possible to navigate from the starting activity of the application to the activity that contains the corresponding UI element. Further, by visible, we mean that the UI element is visible when its parent activity is in the resumed state, i.e., it is running in the foreground. It is worth noting that our definition of crash point is actually simple, since, for example, we have considered that the corresponding *onClick* handler unconditionally forces the application to crash. Our point of view is that if the monkey is not effective for revealing such simple crash points, it would have difficulties in detecting more complex crash conditions.

An important point is that during the monkey testing session, not all the events generated by the monkey are fired on the UI of the application under test. The monkey generates some events outside the UI of the application under test. For instance, the monkey might change the volume or enable/disable Wi-Fi. It also might press the system back button to send the application under test to the background. Other events like capturing the screen are also possible. Taking this point into account, the first step in evaluating the effectiveness of the monkey is to investigate how focused the monkey is on the application under test. Hence, we consider the following research question:

RQ2.1 Does the monkey appropriately focus on the application under test?

Having answered this research question, the next step is to investigate whether the monkey is successful in exploring the application under test and cover different execution paths and conditions. At a high level, we are interested in evaluating what percentage of the activities of the application under test are visited by the monkey. Hence, we consider the following research question:

RQ2.2 Does the monkey cover all the activities of the application under test?

Next, it is required to investigate whether the monkey is successful in revealing the faults injected in the application under test. Hence, we consider the following research question:

RQ2.3 Is the monkey successful in revealing the robustness and responsiveness faults injected in the application under test?

Having defined the research questions of the experiment, next we describe different elements of this experiment.

3.2.1 Application Skeleton

The application skeleton is a specification of an Android application with N activities A_i ($1 \leq i \leq N$) where the UI of each activity is a *GridLayout* containing R rows and C columns. Further, there is a button in each cell of this layout. The skeleton defines five types of buttons:

- *Crash Button* this type of button is intended to inject a crash point in an application. The *onClick* handler of this type of button includes logging a message and then executing a statement that produces an unhandled exception, forcing the application to crash. A simple example of such a statement is an expression containing division by zero.
- *ANR Button* this type of button is intended to inject an ANR point in an application. The *onClick* handler of this type of button first logs a message and then calls Thread.sleep(n) method to inject a delay of n milliseconds in the main thread of the application, preparing the conditions for an ANR error being issued.
- *Next Button* this type of button is intended to enable forward transition between the activities of an application, i.e., for moving from A_i to A_{i+1} ($1 \leq i < N$). The *onClick* handler of a next button logs a message and then starts the target activity.
- *Back Button* this type of button is intended to enable backward transition between the activities, i.e., for moving from A_i to A_{i-1} ($1 < i \leq N$). The *onClick*

Table 4 Specification of the applications in the second experiment

Group	App*k*	AppConf [*N*, *R*, *C*]	ActivityConf [CB_i, AB_i, NB_i, BB_i, DB_i, *T*]
*G*1	$1 \le k \le 5$	[1, *k*, 5]	[0, 0, 0, 0, $R * C$]
*G*2	$6 \le k \le 10$	[$k - 4$, 5, 5]	[0, 0, 1, 0, $R * C - 1$, *N*/*A*] for $i = 1$
			[0, 0, 1, 1, $R * C - 2$, *N*/*A*] for $1 < i < N$
			[0, 0, 0, 1, $R * C - 1$, *N*/*A*] for $i = N$
*G*3	$11 \le k \le 15$	[5, $k - 10$, 5]	[0, 0, $N - 1$, 0, $R * C - (N - 1)$, *N*/*A*] for $i = 1$
			[0, 0, 0, 1, $R * C - 1$, *N*/*A*] for $1 < i \le N$
*G*4	$16 \le k \le 20$	[1, $k - 15$, 5]	[1, 0, 0, 0, $R * C - 1$, *N*/*A*]
*G*5	$21 \le k \le 25$	[2, $k - 20$, 5]	[0, 0, 1, 0, $R * C - 1$, *N*/*A*] for $i = 1$
			[1, 0, 0, 1, $R * \mathrm{C} - 2$, *N*/*A*] for $i = 2$
*G*6	$26 \le k \le 30$	[$k - 23$, 5, 5]	[0, 0, 1, 0, $R * C - 1$, *N*/*A*] for $i = 1$
			[0, 0, 1, 1, $R * C - 2$, *N*/*A*] for $1 < i < N$
			[1, 0, 0, 1, $R * C - 2$, *N*/*A*] for $i = N$
*G*7	$31 \le k \le 35$	[5, $K - 30$, 5]	[0, 0, $N - 1$, 0, $R * C - (N - 1)$, *N*/*A*] for $i = 1$
			[0, 0, 0, 1, $R * C - 1$, *N*/*A*] for $1 < i < N$
			[1, 0, 0, 1, $R * C - 2$, *N*/*A*] for $1 = N$
*G*8	$36 \le k \le 40$	[1, $K - 35$, 5]	[0, 1, 0, 0, $R * C - 1$, 5]
*G*9	$41 \le k \le 50$	[1, 5, 5]	[0, 1, 0, 0, $R * \mathrm{C} - 1$, $2 + ((k - 41) * 0.5)$]

handler of a back button logs a message and then starts the target activity.

- *Dummy Button* the *onClick* handler associated with this type of button simply logs a message. This is intended to identify that the button is clicked.

The message logged by each button includes the location of the button, i.e., its row and column number, and also the number of the parent activity. Hence, by analysis of the log file, it is possible to identify which buttons are clicked by the monkey.

In order to instantiate an application from the skeleton, a Java program is developed which reads configuration of the target application from a file and creates the resulting application from the skeleton. Here, we consider the application configuration to be specified as a set of tuples. The first tuple *AppConf* is of the form [*N*, *R*, *C*] where *N* is the number of activities and *R* and *C* are, respectively, the number of rows and columns in each activity. Further, for each activity A_i ($1 \le i \le N$), there is a tuple $\mathit{ActivityConf}_i$ of the form [CB_i, AB_i, NB_i, BB_i, DB_i, *T*] where CB_i, AB_i, NB_i and DB_i, respectively, represent the number of crash buttons, ANR buttons, next buttons, back buttons and dummy buttons in the activity A_i. If $\mathrm{AB}_i > 0$, i.e., there is an ANR button, then *T* represents the sleep in seconds that is injected by the ANR button. When creating the application, each button is put in a randomly selected cell of the underlying layout. It is worth noting that the layout of the UI is stretched and all margin and padding parameters are set to zero, so that all the available space of the screen is filled with the buttons, assuming that there is at least one button in each activity.

3.2.2 Application Instances

Using the skeleton, we have created 50 concrete Android applications App*k*, $1 \le k \le 50$, divided into nine groups *Gi*, $1 \le i \le 9$. The configurations of these applications are presented in Table 4.

The first group *G*1 includes applications with one activity and different numbers of rows where all the buttons are dummy buttons. This group is considered in order to evaluate how much the monkey focuses on the application under test and also to analyze how the interaction of the monkey is distributed over the buttons that are located in different positions. The second group *G*2 includes applications with different numbers of activities where each activity is a 5×5 grid containing a next button to the next activity and a back button to the previous activity. The other buttons are dummy buttons. The first and the last activities have, respectively, no back and next buttons. This group is considered to analyze how successful the monkey is in exploring the application by navigating to different activities. The applications in *G*3 have five activities with five columns and different numbers of rows where the first activity has a next button to every other activity and the other activities have a back button to the first activity. There are no crash buttons in the activities. This group is taken into consideration in order to simulate a traditional navigation model of the applications, where the first activity includes a menu for navigating to other activities.

The applications of *G*4 have one activity with five columns and different numbers of rows, where the activity contains a single crash button and the rest of the buttons are dummy buttons. The group *G*5 includes applications with two activities that there is a next button from the first activity to the second activity, and there is a back button from the second activity to the first activity. Further, there is a crash button in the second activity. The group *G*6 is similar to *G*5, but its applications have different numbers of activities. Again there are next and back buttons between the activities, and there is a crash button in the last activity. In the *G*7 group, each application has five activities where the first activity has next button to each of the other activities, and the other activities have a back button to the first activity. Further, the last activity has a crash button. The four groups *G*4–*G*7 are considered to evaluate how successful the monkey is in revealing crash points. The difference is in the path that needs to be traversed by the monkey to find the crash button, e.g., in *G*4, the crash button is in the first and the only activity of the application, but in *G*5, *G*6, and *G*7 revealing the crash point requires the monkey to first navigate through the next buttons to the last activity and then click the crash button.

The last two groups, i.e., *G*8 and *G*9, include applications for evaluating the monkey from the perspective of revealing ANR points. In both groups, the applications have a single activity containing one ANR button. The difference is that in *G*8, the ANR buttons all have 5 s delay, but in *G*9 applications, the ANR buttons have different delays. For instance, in app41 and app50 the ANR button injects, respectively, a 2 and a 6.5 s delay. The applications in *G*9 are considered to evaluate the sensitivity of Android to responsiveness issues.

As an example of a concrete application created from the application skeleton, the first activity of App35 is shown in Fig. 2. This application has five activities, and the UI of each activity has five rows and five columns. The first activity has 21 dummy buttons and four next buttons. For instance, the next button at the third column of the first row moves the control to the second activity of the application. The second activity of App35 is shown in Fig. 3. It has a back button to the first activity in the last column of the fourth row.

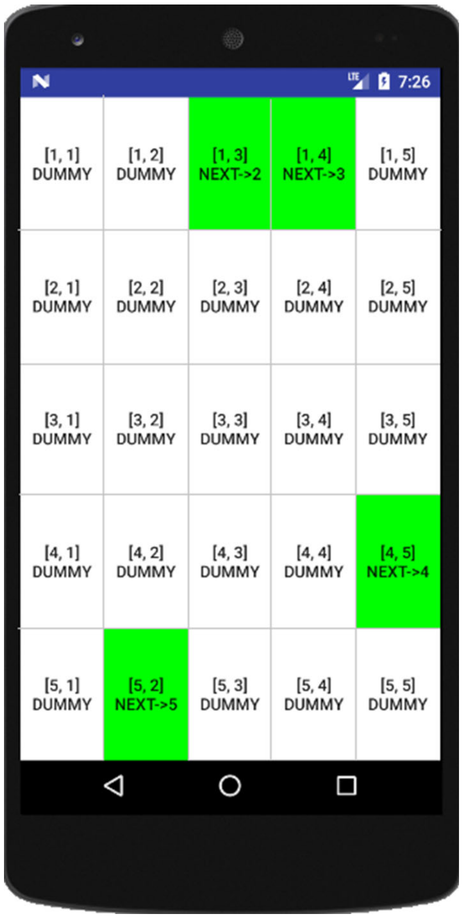

Fig. 2 First activity of App35

3.2.3 Test Process

In order to execute the experiment for each target application, the same test script as the first experiment is used. For each application, the test script is executed 100 times with the number of events set to 10 K.

3.2.4 Metrics

In the first experiment, we have used real-world applications without any modification in their source code. In the second experiment, since the source code of the application includes appropriate fine-grained logging functionality, i.e., each click on each button of the application is logged, it is possible to perform analysis that is more precise. Hence, we distinguish between two notions of *Monkey Event* and *App Event*. A Monkey Event is any event that the monkey fires, whether on the target application or on other applications running on the device. An App Event is an event that the monkey has fired on the UI of the application under test. Considering this distinction, we revise definition of CrashDelay$_n$ and define crashDelayA$_n$ for a monkey testing session of length n as the number of App Events that has made the application under test crash. Additionally, we define crashDelayM$_n$ as the number of Monkey Events that has forced the application under test to crash. Similarly, we define ANRDelayA$_n$ and ANRDelayM$_n$.

In addition to the customized version of the previous metrics, we also define the following metrics in this experiment.

- Focus$_n$. An important factor that is expected to affect the ability of the monkey in revealing crash points or ANR points is the percentage of the events generated by the monkey that are fired on the UI of the application

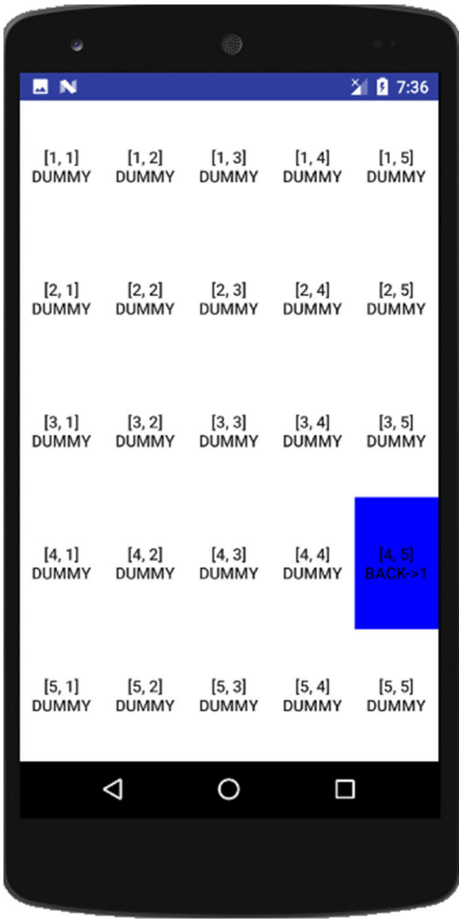

Fig. 3 Second activity of App35

under test. Hence, we define *focus*$_n$ as the percentage of the App Events to the Monkey Events for a monkey testing session of length n events. The idea is that generally the greater this value is, the higher the chance of the crash points or ANR points being revealed.

- ActivityCoverage$_n$ is the percentage of the activities of the application under test that have been visited by the monkey during a monkey testing session of length n events. The monkey has visited an activity if it has fired at least one event on the UI of the corresponding activity. This metric is defined since covering the activities is a necessary condition for revealing crashes that might happen during interacting with the UI of the application.

3.2.5 Results

We have executed the test script on the target applications and have analyzed the results. Here, the results are discussed for different groups of applications separately.

(a) $G1$, $G2$, $G3$

The three groups $G1$, $G2$ and $G3$ are considered to evaluate focus of the monkey on an application under test. Analysis of the log files associated with the applications in $G1$ that have only one activity reveals that on average we have focus$_n$ = 5.5% for n = 10 K, with a standard deviation of 0.66%. This means that, surprisingly, only 5.5% of the events that the monkey has created have been fired on the UI of the application under test. So, we conclude that the research question RQ2.1 should be answered with a no, since the focus of the monkey is less than expected. We admit that the monkey should not limit its interactions only to the application under test, because some external interactions, like enabling Wi-Fi, might affect the execution of the application under test, and hence, provide opportunity for the faults to be revealed. However, we believe that 5.5% focus is a very low focus.

Considering ActivityCoverage$_n$, since the applications in $G1$ have only one activity, we have ActivityCoverage$_n$ = 100%, i.e., the monkey has been successful in covering the only activity of the $G1$ applications in all monkey testing sessions.

The applications in $G1$ all have a single activity with five columns and different numbers of rows. It is interesting to investigate how the monkey's events are distributed over different buttons in the UI of this activity. We have analyzed the log files and computed for each button, the number of times it is clicked by the monkey. Then, the average value over the 100 monkey testing sessions is computed. The results are shown in Fig. 4. In this figure, UI of each application is modeled as a rectangle containing a set of cells. Each cell is representative of a button in the UI of the application. Further, the value written in each cell shows the average percentage of the App Events generated by the monkey that are fired on that button. It is a surprising observation that as shown in this figure, the distribution of the interactions of the monkey over the UI of the application under test is biased to the top left corner of the screen. For instance, for App1, 35% of the events fired on the UI of the application are fired on the first button, i.e., the leftmost button. Other buttons have had a nearly equal share from the App Events. Further, in App5, the first button at the top left corner of the UI has received about 16% of the App Events, while other buttons have received between 4 and 7% of the events. Currently, we do not have any justification for this issue, and we plan to discuss it with the developers of the monkey tool.

For applications in $G2$ and $G3$ groups, we have computed average ActivityCoverage$_n$ over 100 monkey testing sessions. The results are shown in Table 5. For instance, for App6, the average ActivityCoverage$_n$ is 100%, meaning that in all the monkey testing sessions of App6, all the two

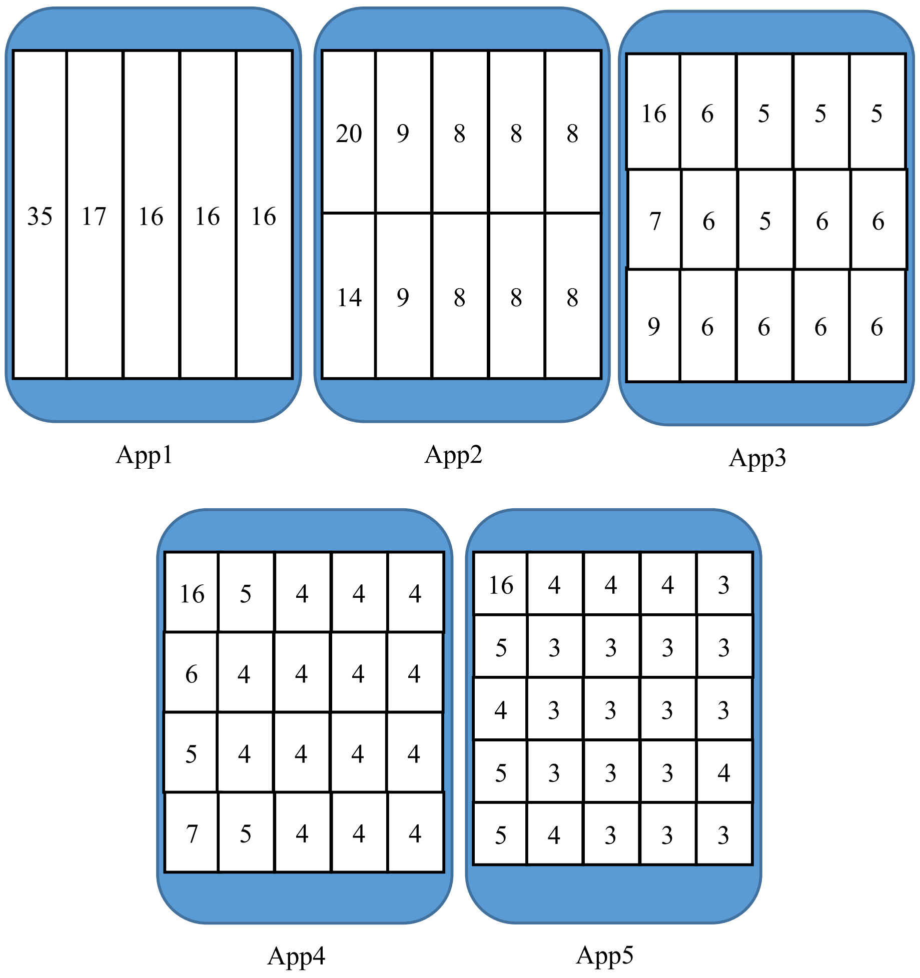

Fig. 4 Average distribution of the ratio of App Events fired on each button in ***G*1** applications

activities of this application have been covered. However, for App10, the average ActivityCoverage$_n$ is about 92%, meaning that on average, 92% of the activities of this application have been covered. This means that in some monkey testing sessions, not all the activities of App10 are covered. In addition to ActivityCoveragen, we have also computed the average distribution of the App Events on different activities of the applications. The results are presented in Table 5. For instance, for App6 that has only two activities, on average, 56% of the App Events generated by the monkey are fired on the first activity, and 44% are fired on the second activity. For App10 that has six activities, on average, 24% of the App Events are generated on the first activity, and 10% are fired on the sixth activity. It is worth noting that the applications are started from the first activity and hence, covering the second or next activities means that the monkey has been successful in clicking on the next buttons to move to next activities. Although the monkey has not fired equal number of events on different activities, there is not noticeable difference between the distributions of the App Events over different activities. The most unfair distribution is associated with App9 and App6 where the standard deviation of the distribution values is, respectively, about 9 and 8.

Based on the results discussed above, we conclude that the monkey is successful in covering activities of the application under test. Hence, we give a positive answer to RQ2.2. Comparing the results in Table 5 for the applications in *G*2 with the applications in *G*3, it is observed that the monkey has performed nearly similarly in covering different activities in these applications. This is interesting since these two groups simulate different navigational models in the applications. The applications in *G*2 implement a sequential navigation model where each activity can activate its next activity, while the applications in *G*3 have a star model where the first activity can transfer control to any other activity. The results demonstrate that the monkey has performed well for both these models.

(b) *G*4, *G*5, *G*6, *G*7

The four groups *G*4, *G*5, *G*6 and *G*7 are considered to investigate how effective is the monkey in revealing crash

Table 5 Results of the second experiment for G2 and G3 with n = 10 K

Group	Application	Average ActivityCoverage$_n$ (%)	Average distribution of App Events on activities (%)					
			Activity 1	Activity 2	Activity 3	Activity 4	Activity 5	Activity 6
G2	App6	100	56	44	N/A	N/A	N/A	N/A
	App7	100	34	35	31	N/A	N/A	N/A
	App8	100	28	28	25	19	N/A	N/A
	App9	92	34	25	17	14	11	N/A
	App10	91.66	24	22	17	15	12	10
G3	App11	100	22	29	16	17	16	N/A
	App12	100	22	24	21	15	18	N/A
	App13	100	24	24	18	16	18	N/A
	App14	99.33	22	25	20	15	18	N/A
	App15	99.33	25	20	19	15	21	N/A

Table 6 Results of the second experiment for G4, G5, G6 and G7 with n = 10 K

Group	Application	CrashRate$_n$ (%)	Average CrashDelayA$_n$	Average CrashDelayM$_n$
G4	App16	100	5	128
	App17	96	11	242
	App18	100	27	525
	App19	100	17	333
	App20	100	27	472
G5	App21	100	13	260
	App22	100	27	526
	App23	100	42	800
	App24	100	80	1312
	App25	100	75	1260
G6	App26	90	174	2724
	App27	93	293	4635
	App28	83	275	4363
	App29	53	315	4782
	App30	40	425	6906
G7	App31	100	61	1127
	App32	100	102	1690
	App33	96	152	2323
	App34	90	163	2520
	App35	93	168	2642

points injected in an application under test by the use of crash buttons. By analysis of the log files, we have calculated the CrashRate$_n$ and average CrashDelayA$_n$ and CrashDelayM$_n$ for the applications in these groups. The results are shown in Table 6. As shown in this table, for all the applications in $G4$ and $G5$, except App17, we have CrashRate$_n$ = 100%, meaning that for these applications, in 100% of the monkey testing sessions, the monkey has been able to reveal the injected crash point by clicking on the Crash Button of the application. For App17, this has happened in 96% of the monkey testing sessions.

For the applications in $G6$, the value of CrashRaten is reduced. For instance, for App30, we have CrashRaten = 40%. This means that in more than the half of the money testing sessions, the monkey has not been able to reveal the crash point. For $G7$, the results are much better than $G6$.

Considering the average CrashDelay$_n$, it is observed that for the applications in $G4$, the maximum average CrashDelayA$_n$ is 30. This means that in those monkey testing sessions that have successfully revealed the crash point, the crash button is clicked by the monkey during the first 30 events it has generated on the UI of the application under test. Further, the average CrashDelayM$_n$ for $G4$ is

Table 7 Results of the second experiment for G8 and G9 with n = 10 K

Group	Application	ANRRate$_n$ (%)	Average ANRDelayA$_n$	Average ANRDelayM$_n$	Average number of times ANR button is clicked
G8	App36	100	19	356	3
	App37	100	43	716	4
	App38	100	74	1222	4
	App39	100	68	1302	4
	App40	100	116	2301	5
G9	App41	6	494	7504	10
	App42	6	261	4013	10
	App43	10	367	6353	12
	App44	20	200	3748	10
	App45	10	236	4522	12
	App46	17	234	4245	8
	App47	100	91	1685	4
	App48	100	39	676	2
	App49	100	29	451	1
	App50	100	36	707	1

525, meaning that on average, the crash point is revealed during the first 525 events generated by the monkey.

Based on the results discussed above, we conclude that the research question RQ2.3 can be answered with a yes. The monkey is successful in revealing the robustness faults injected in the application under test. Actually, the rate in which the faults are detected is generally high, and also the time that it takes for the monkey to reveal the fault can be considered to be short.

(c) *G*8 and *G*9

The two groups *G*8 and *G*9 are considered for evaluating the effectiveness of the monkey in revealing ANR points in the application under test. We have analyzed the log files and have calculated the ANRRate$_n$ and average ANRDelay$_n$ for the applications in these groups. The results are shown in Table 7. As shown in this table, for all the applications in *G*8, we have ANRRate$_n$ = 100%. This means that for these applications, in all the monkey testing sessions, the monkey has successfully revealed the ANR point by clicking on the ANR button of the application and forcing an ANR error being issued. Further, for *G*8 applications, the maximum value of the average ANRDelayA$_n$ is 116, meaning that on average, 116 events are fired on the UI of the application before the ANR error is issued. In addition, the maximum value of the average ANRDelayM$_n$ is 2301. This means that on average, 2301 events are fired by the monkey before the ANR error is issued.

Another interesting observation is reflected in the last column of Table 7. While in all the monkey testing sessions for all the applications of *G*8 and *G*9, the ANR button is clicked at least once, the first click on the ANR button of the application has not necessarily resulted in an ANR error being issued immediately. For instance, for App40, the average number of times that the ANR button is clicked is 5. It shows that it has taken some time for the Android runtime system to identify unresponsiveness of the application under test, and during this time, the monkey has had time to do further clicks on the ANR button. This makes it interesting to investigate the threshold of Android's sensitivity to unresponsiveness, although announced as 5 s in Android official documents. The applications in *G*9 are considered for this purpose, since their ANR button injects a sleep of different lengths, from 2 to 6.5 s.

The results for *G*9 applications are interesting from the point of view of an Android developer who is interested in developing responsive applications. For App41, the ANR button is configured to inject a 2 s delay, simulating a heavy operation that takes about 2 s. From the point of view of developing responsive applications, performing a 2-s operation in the application's main thread is not acceptable. Hence, a developer who relies on monkey testing for checking this issue might expect that the monkey is able to reveal this defect in his application. However, the results in Table 7 show that ANRRate$_n$ for App41 is 6%, meaning that only in 6% of the monkey testing sessions is an ANR error issued. On average, the ANR error is issued after the monkey has generated 7504 events, 494 of which have been fired on the UI of the application under test. Further, on average, the ANR button is clicked ten times before the ANR error is issued. For the next application, the length of the injected delay is increased and hence the value of $\mathrm{ANRRate}_n$ and $\mathrm{ANRDelay}_n$ is improved, i.e., respectively, increased and decreased. Only for App49 and App50 that have 6 s and 6.5 s delay, the first click of the monkey on the ANR button has been sufficient for the ANR error being issued.

Based on the results discussed above, we answer the research question RQ2.3 with a yes, and we conclude that the monkey is successful in revealing injected responsiveness faults in the application under test. However, we also conclude that the responsiveness fault should inject a delay of about 5 s in order to be detected by the monkey.

4 Conclusion

Android application development is rapidly gaining more attention from the software development community. So it is crucial to consider evaluation of these applications more carefully. In this paper, we have conducted experimental studies, demonstrating that monkey testing of Android applications for finding robustness and responsiveness errors is very effective and highly recommended. Further, it is observed that the Android applications published in Iranian app stores are more vulnerable to robustness and responsiveness errors, compared to the applications published in international app stores. This emphasizes the need for robustness and responsiveness testing to be taken into consideration more seriously by the Iranian developers. In addition, we have shown that while the default monkey tool available in Android Studio IDE has a very low focus, i.e., 5.5%, it is still effective in revealing robustness and responsiveness faults injected in the application under test. Regarding the responsiveness faults, however, the monkey is much less sensitive to the faults that cause delays less than 5 s. Based on the results concluded from the experiments, our future work includes more thorough investigation of the low focus of the monkey. Further, we plan to perform more experiments on analysis of the distribution model of the location of the events generated by the monkey.

References

Amalfitano D, Fasolino AR, Tramontana P (2011) A GUI crawling-based technique for Android mobile application testing. In: Fourth IEEE international conference on software testing, verification and validation, ICST 2012, Berlin, Germany, 21–25 March 2011, workshop proceedings, pp 252–261

Amalfitano D, Fasolino AR, Tramontana P, Carmine SD, Memon AM (2012) Using GUI ripping for automated testing of Android applications. In: IEEE/ACM international conference on automated software engineering, ASE'12, Essen, Germany, 3–7 September 2012, pp 258–261

Ardito L, Coppola R, Morisio M, Torchiano M (2019) Espresso vs. EyeAutomate: an experiment for the comparison of two generations of Android GUI testing. In: Proceedings of the evaluation and assessment on software engineering, EASE 2019, Copenhagen, Denmark, 15–17 April 2019, pp. 13–22

Baek YM, Bae D (2016) Automated model-based Android GUI testing using multi-level GUI comparison criteria. In: Proceedings of the 31st IEEE/ACM international conference on automated software engineering, ASE 2016, Singapore, 3–7 September 2016, pp 238–249

Canfora G, Sorbo AD, Mercaldo F, Visaggio CA (2016) Exploring mobile user experience through code quality metrics. In: 17th international conference on product-focused software process improvement, PROFES 2016, Trondheim, Norway, 22–24 November 2016, proceedings, pp 705–712

Choudhary SR, Gorla A, Orso A (2015) Automated test input generation for Android: are we there yet?. In: 30th IEEE/ACM international conference on automated software engineering, ASE 2015, Lincoln, NE, USA, 9–13 November 2015, pp 429–440

Deng L, Offutt J, Ammann P, Mirzaei N (2017a) Mutation operators for testing Android apps. Inf Softw Technol 81:154–168

Deng L, Offutt J, Samudio D (2017b) Is mutation analysis effective at testing Android apps? In: 2017 IEEE international conference on software quality, reliability and security, QRS 2017, Prague, Czech Republic, 25–29 July 2017, pp 86–93

Gómez M, Rouvoy R, Adams B, Seinturier L (2016) Mining test repositories for automatic detection of UI performance regressions in Android apps. In: Proceedings of the 13th international conference on mining software repositories, MSR 2016, Austin, TX, USA, 14–22 May 2016, pp 13–24

Haoyin L (2017) Automatic Android application GUI testing, a random walk approach. In: 2017 international conference on wireless communications, signal processing and networking (WiSPNET). IEEE, pp 72–76

Kang Y, Zhou Y, Gao M, Sun Y, Lyu MR (2016) Experience report: detecting poor-responsive UI in Android applications. In: 27th IEEE international symposium on software reliability engineering, ISSRE 2016, Ottawa, ON, Canada, 23–27 October 2016, pp 490–501

Kochhar PS, Thung F, Nagappan N, Zimmermann T, Lo D (2015) Understanding the test automation culture of app developers. In: 8th IEEE international conference on software testing, verification and validation, ICST 2015, Graz, Austria, 13–17 April 2015, pp 1–10

Kong P, Li L, Gao J, Liu K, Bissyande TF, Klein J (2019) Automated testing of Android apps: a systematic literature review. IEEE Trans Reliab 68(1):45–66

Liu Y, Xu C, Cheung S (2014) Characterizing and detecting performance bugs for smartphone applications. In: 36th international conference on software engineering, ICSE'14, Hyderabad, India, 31 May–07 June 2014, pp 1013–1024

Machiry A, Tahiliani R, Naik M (2013) Dynodroid: an input generation system for Android apps. In: Joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, 18–26 August 2013, pp 224–234

Mao K, Harman M, Jia Y (2016) Sapienz: multi-objective automated testing for Android applications. In: Proceedings of the 25th international symposium on software testing and analysis, ISSTA 2016, Saarbrücken, Germany, 18–20 July 2016, pp 94–105

McDonnell T, Ray B, Kim M (2013) An empirical study of API stability and adoption in the Android ecosystem. In: 2013 IEEE international conference on software maintenance, Eindhoven, The Netherlands, 22–28 September 2013, pp 70–79

Ongkosit T, Takada S (2014) Responsiveness analysis tool for Android application. In: Proceedings of the 2nd international workshop on software development lifecycle for mobile, DeMobile 2014, Hong Kong, China, November 17, 2014, pp 1–4

Patel P, Srinivasan G, Rahaman S, Neamtiu I (2018) On the effectiveness of random testing for Android: or how I learned to stop worrying and love the monkey. In: Proceedings of the 13th international workshop on automation of software test, AST@ICSE 2018, Gothenburg, Sweden, 28–29 May 2018, pp 34–37

Song W, Qian X, Huang J (2017) EHBDroid: beyond GUI testing for Android applications. In: Proceedings of the 32nd IEEE/ACM international conference on automated software engineering, ASE 2017, Urbana, IL, USA, 30 October–03 November 2017, pp 27–37

Su T (2016) FSMdroid: guided GUI testing of Android apps. In: Proceedings of the 38th international conference on software engineering, ICSE 2016, Austin, TX, USA, 14–22 May 2016, companion volume, pp 689–691

Sun C, Zhang Z, Jiang B, Chan WK (2016) Facilitating monkey test by detecting operable regions in rendered GUI of mobile game apps. In: 2016 IEEE international conference on software quality, reliability and security, QRS 2016, Vienna, Austria, 1–3 August 2016, pp 298–306

Thiagarajan D (2016) Efficient detection of hang bugs in mobile applications. Ph.D. thesis, The Ohio State University

Tramontana P, Amalfitano D, Amatucci N, Fasolino AR (2019) Automated functional testing of mobile applications: a systematic mapping study. Softw Qual J 27(1):149–201

Vásquez ML, Bavota G, Bernal-Cárdenas C, Penta MD, Oliveto R, Poshyvanyk D (2013) API change and fault proneness: a threat to the success of Android apps. In: Joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, 18–26 August 2013, pp 477–487

Vásquez ML, Bernal-Cárdenas C, Moran K, Poshyvanyk D (2017) How do developers test Android applications? In: 2017 IEEE international conference on software maintenance and evolution, ICSME 2017, Shanghai, China, 17–22 September 2017, pp 613–622

VisionMobile (2016) Developer economics q3 2016: state of the developer nation. https://www.visionmobile.com/reports/developer-economics-state-developer-nation-q3-2016. Accessed 20 July 2018

Wang Y, Rountev A (2016) Profiling the responsiveness of Android applications via automated resource amplification. In: Proceedings of the international conference on mobile software engineering and systems, MOBILESoft'16, Austin, Texas, USA, 14–22 May 2016, pp 48–58

Wei X, Gomez L, Neamtiu I, Faloutsos M (2012) ProfileDroid: multi-layer profiling of Android applications. In: The 18th annual international conference on mobile computing and networking, Mobicom'12, Istanbul, Turkey, 22–26 August 2012, pp 137–148

Wetzlmaier T, Ramler R (2017) Hybrid monkey testing: enhancing automated GUI tests with random test generation. In: Proceedings of the 8th ACM SIGSOFT international workshop on automated software testing, A-TEST@ESEC/SIGSOFT FSE 2017, Paderborn, Germany, 4–5 September 2017, pp 5–10

Wetzlmaier T, Ramler R, Putschögl W (2016) A framework for monkey GUI testing. In: 2016 IEEE international conference on software testing, verification and validation, ICST 2016, Chicago, IL, USA, 11–15 April 2016, pp 416–423

Yang S, Yan D, Rountev A (2013) Testing for poor responsiveness in Android applications. In: 2013 1st international workshop on the engineering of mobile-enabled systems (MOBS), pp 1–6. IEEE

Zeng X, Li D, Zheng W, Xia F, Deng Y, Lam W, Yang W, Xie T (2016) Automated test input generation for Android: are we really there yet in an industrial case? In: Proceedings of the 24th ACM sigsoft international symposium on foundations of software engineering, FSE 2016, Seattle, WA, USA, 13–18 November 2016, pp 987–992