International Journal of Multimedia and Ubiquitous Engineering
Vol.11, No.9 (2016), pp.1-10
http://dx.doi.org/10.14257/ijmue.2016.11.9.01

An Empirical Analysis of Android Apps Bug and Automated Testing Approach for Android Apps

Yi Bie[1], Sheng Bin[2], Gengxin Sun[1*] and Xicheng Zhou[1]

[1]Software Technical College of Qingdao University, Qingdao, China*
[2]International College of Qingdao University, Qingdao, China
[1]rjxy@qdu.edu.cn,[2]binsheng@qdu.edu.cn,[1]sungengxin@qdu.edu.cn*

Abstract

Android platforms and its applications (apps) have gained tremendous popularity recently, hence the reliability of Android apps is becoming increasingly important. Due to the novelty of the Android platform, apps are prone to errors, which would affect user experience and requires frequent bug fixes. In this paper, an empirical study on bugs in some widely-used open-source Android apps from diverse categories such as media, tools and communication were performed. Based on the findings of the empirical study, an approach for automating the testing process for detecting Android apps GUI bugs was presented. We show how the approach helped to re-discover existing bugs and find new bugs, and how it could be used to prevent certain bug categories. Our empirical study and automated testing approach have the potential to help developers increase the quality of Android apps.

***Keywords**: Google Android; Android apps; empirical bug analysis; test automation; bug reports*

1. Introduction

Android platform and the apps that run on the platform have gained tremendous popularity recently. A major draw of Android platform is its ability to run applications, it leads to an increasing impetus for ensuring the reliability of Android apps. Reliability is particularly important for sensitive apps such as online banking, business management, health care and so on. However, the low barrier to enter the Android Market means apps are subject to limited scrutiny before dissemination, allowing error-prone apps through and therefore affecting user experience.

In this paper we focus on ensuring the reliability of Android apps. The first step towards ensuring the reliability is to understand the nature of bugs and the bug-fixing process associated with Android apps. The open-source nature of a great deal of Android apps provides an opportunity to conduct empirical researches and provide a quantitative basis for improving the quality of open-source Android apps.

The main contribution of our empirical analysis is that several metrics are defined to understand the quality of bug reports. We conduct an investigation of the categories of GUI bugs in Android apps and perform an in-depth study of GUI bugs in our examined Android apps. We categorized all confirmed bugs in the bug database. To detect and fix these categories of bugs, we proposed an automated test approach. Our approach uses a combination of case and event generation with runtime monitoring and log file analysis.

Most existing tools and techniques for automating the testing have so far focused mostly on desktop and server applications [1]. However, the physical constraints of mobile devices would make mobile applications prone to new kinds of bugs. For example, an Android application consists of activities, services, broadcast receivers and content providers. It is very different from standard server applications or desktop applications.

ISSN: 1975-0080 IJMUE
Copyright ⓒ 2016 SERSC

The tendency of mobile applications to have bugs is evidenced by their high defect density [2].

This paper is organized as follows. Section 2 is the overview of Android platform and Android apps. A bug empirical analysis on 24 popular open-source Android apps is introduced in Section 3. An automated test approach is proposed in Section 4, we employ test and event generators to construct test cases and events sequences, then these test cases are run to the application. Once a test case is running, detailed information about the application is recorded in the system log file and a log file analysis is performed to detect potential bugs. For the effectiveness of our approach, in Section 5 we compared bugs which we found by generating test cases with bugs reported by users. We detected most bugs reported, and found new bugs which have never been reported.

2. Overview of Android Apps and Android Bugs

As shown in Figure 1, the Android platform is composed of 4 layers: Applications at the top, the Application Framework layer that provides services to applications, the Library/VM layer and the Linux kernel at the bottom.

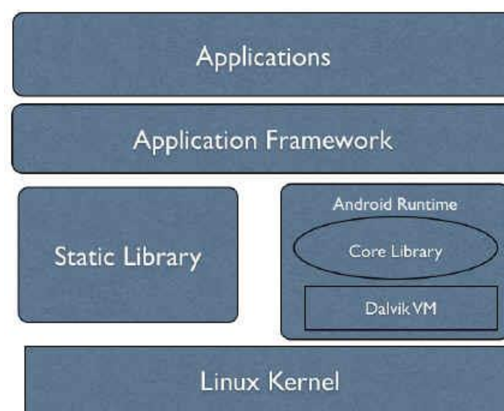


Figure 1. 4 Layer Architecture of Android Platform

Our work covers the top three layers in Figure 1. To test programs running in the Application layer, system services from the Application Framework layer and instrumentation tools in the Dalvik VM are used.

There are four types of components used to construct applications in Android: Activity, Broadcast Receiver, Content Provider and Service, which require specific management rules and a particular lifecycle. Activities are focused user interfaces in which the user interaction takes place. Only one activity can be active at a time. All visible portions of applications are Activities. Services run in the background and do not interact directly with the user. Other components can bind to a Service, which lets the binder invoke methods that are declared in the target Service's interface. A Content Provider manages data for a certain application and controls the accessibility of the data. It can be acted as a mechanism for sharing information between applications. Broadcast Receivers listen and react to broadcast announcements. They are triggered by the receipt of an appropriate Intent and then run in the background to handle the event.

A description of typical bugs encountered in Android apps shows that frequent bugs are due to incorrect management of the Activity component lifecycle [3]. This component provides crucial functions for the application's user interface and reacts to events generated by users and other system components [4]. Incorrect management of these events often results in wrong or unsatisfactory application behavior.

Activities are the main GUI components of an Android app, an activity error usually occurs due to incorrect implementations of the activity protocol. Event errors occur when

 Copyright ⓒ 2016 SERSC

the application performs a wrong action as a result of receiving an event. Dynamic type errors arise from runtime type exceptions. Unhandled exceptions are exceptions the user code does not catch and lead to an application crash. API errors are caused by incompatibilities between the API version assumed by the application and the API version provided by the system. I/O errors stem from I/O interaction. Concurrency errors occur due to the interaction of multiple processes or threads. There are some bugs categorized as other due to errors in the program logic.

Though our approach can be general enough to facilitate bug detection for all Android components, in this paper we focus on bugs related to activities and events generated by users and system.

3. Empirical Analysis of Android Apps Bug

To identify the most frequent Android bugs, the empirical study on 24 popular open-source Android apps was performed. We chose Android apps from a wide range of categories to reduce selection bias. At the same time, the Android apps that form the target of our bug empirical study are available for free in Android Market, have high download counts, which ensures we get a broad range of representative bugs.

In our empirical analysis, we mainly analyze the bug reports of Android apps. Some prior work has shown that bug report quality critically affects software quality [5]. Empirical studies on desktop applications such as Mozilla and Apache have shown that bug reports with certain qualities significantly help developers to understand the problem and reduce the bug-fix time [6]. We define five metrics to measure bug report quality for the Android apps we considered:

- *Description Length*, which counts the number of words in the bug description.
- *Reproduce Steps*, which represents the percentage of bug reports that have steps to reproduce the bug in the bug description.
- *Output Details*, which represents the percentage of bug reports that contain details of expected output and actual output.
- *Additional Information*, which represents the percentage of bug reports containing additional information about the bug, besides the standard bug report.
- *All Details*, which measures the percentage of bugs that have all three details: Reproduce Steps, Output Details and Additional Information.

High values of these metrics indicate high quality bug reports. In Table 1, we present the values of all these metrics for each Android app.

Table 1. Bug Description Metrics

Application	Description Length (words)	Reproduce Steps (%)	Output Details (%)	Additional Information (%)	All Details (%)
Firefox Mobile	323.91	0.02	0.12	N/A	0
ADW Launcher	116.50	31.71	28.99	13.55	10.03
DealDroid	327.09	49.46	47.08	9.29	8.86
SMSPopup	128.32	58.36	57.34	31.06	28.67
WiFiTether	153.39	80.94	74.04	46.51	44.36
XBMC Remote	157.69	60.86	58.38	23.38	21.56
AnkiDroid	136.26	32.43	26.05	12.76	11.30
CallMeter3G	95.99	42.02	40.28	15.32	13.90
ConnectBot	169.52	53.38	49.36	25.78	24.50
CSipSimple	152.97	62.92	57.75	38.72	36.49
CyanogenMod	281.16	74.55	0.19	10.04	0.02
GAOSP	153.62	53.46	48.85	11.35	11.35
IMSDroid	201.43	69.44	66.98	31.17	30.56
JustPictures	114.05	36.39	33.33	0	0
Rokon	209.78	62.90	49.38	31.73	30.01
My Tracks	157.23	38.13	33.11	9.76	8.97
OpenIntents	165.11	51.59	47.81	25.30	24.10
OsmAnd	140.05	37.88	30.88	9.53	8.81
OSMDroid	161.10	31.56	28.57	14.95	14.29
Sipdroid	200.28	50.37	48.27	28.65	26.23
SoftKeyboard	118.30	42.73	39.98	27.49	26.69

Copyright ⓒ 2016 SERSC

TransDroid	106.75	29.49	0.54	0.27	0.27
WebSMSDroid	93.97	51.47	45.89	19.85	18.92
CMIS	175.38	46.49	43.42	24.05	22.34

From Table 1 we can see that *Description Length* varies significantly, from a low of 93.97 words to a high of 327.09 words, and Reproduce Steps which contain steps to reproduce the bug in the bug description varies from 0.02 to 80.94%. Note that the steps to reproduce a bug are not mandatory but have the potential to increase the chances of the bug being fixed quickly. Most Android apps have the highest percentage of bug reports that contain information about the expected and actual output (*Output Details*). To summarize, we found that bug reports of most Android apps have high quality. Bug reporters usually provide long textual descriptions of the problem, steps to reproduce the bug, and explanation of the difference between expected and the actual outputs.

To understand the effects of bug report quality for bug fixing time, a regression analysis on the bug report quality metrics is performed. We observed that *Description Length* is highly correlated with the other metrics. Therefore, to understand the effects of bug report quality for bug fixing time, a linear regression with *Description Length* as the single independent variable is performed. We found a negative correlation for the apps we considered between *Description Length* and bug fixing time. The negative values of the correlation coefficient indicate that *Description Length* is a good predictor of bug-report quality and that high-quality bug reports get fixed faster.

In our empirical analysis, the bug counts categorized by bug type for each Android app is also analyzed, the analysis results are presented in Table 2.

Table 2. Bug Categories and Bug Counts

Application	**Bug category**				
	Unhandled exception	**API**	**I/O**	**Concurrency**	**Other**
Firefox Mobile	1	0	0	0	4
ADW Launcher	2	0	0	0	6
DealDroid	0	1	0	0	4
SMSPopup	5	1	3	1	57
WiFiTether	0	0	1	0	3
XBMC Remote	2	1	0	0	8
AnkiDroid	3	0	4	0	14
CallMeter3G	0	0	1	0	2
ConnectBot	6	1	0	0	2
CSipSimple	0	2	0	0	5
CyanogenMod	4	1	0	0	3
GAOSP	2	0	0	0	2
IMSDroid	3	2	1	1	2
JustPictures	0	0	0	0	1
Rokon	2	1	0	0	5
My Tracks	2	0	0	0	1
OpenIntents	2	0	0	0	1
OsmAnd	1	0	0	0	1
OSMDroid	0	0	0	0	1
Sipdroid	1	7	0	0	1
SoftKeyboard	2	0	1	0	2
TransDroid	1	1	0	0	1
WebSMSDroid	0	0	0	0	1
CMIS	1	0	0	0	1

From Table 2 we can see, many errors are program logic related errors, some of which can be found using standard techniques, such as static analysis or model checking. But these techniques can't be applied directly to Android apps because of their structure and libraries differ substantially. Therefore, we proposed an automated approach that can detect a variety of Android bugs and show how it can be used to detect activity, event and type errors.

Copyright ⓒ 2016 SERSC

4. Automated Testing Approach for Detecting Android Apps Bugs

Our automated testing approach consists of automatic test case and event generation tools to log file analysis. Firstly, JUnit [7] which is a Java test case generation tool is used to generate test cases for Android apps. Because most Android apps are GUI-based, for each test case, we may need to add some events which are used for simulating user interaction to make the application move from one state to another. So we use Monkey [8], an automatic event generation tool, to produce events in both random and deterministic ways and feed these events to the application. Once a test case is running, detailed information about the application is recorded in the system log file. After each test case run, a log file analysis to detect potential bugs is performed. The overview of our approach is shown as Figure 2.

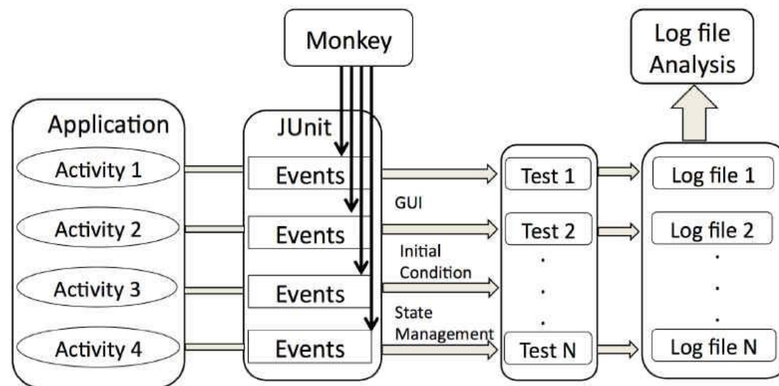


Figure 2. Overview of Our Approach

4.1. Test Case Generation

In our approach, JUnit is used to generate several classes of test cases based on source code of Android apps. Because activities are the main entry points and control flow drivers in Android apps, our test case generation is based on activities.

We first identify all activities in an application and then use the Activity Testing which is shipped with the Android SDK in JUnit to generate test cases for each activity. Activity Testing works in conjunction with JUnit and provides three kinds of testing:

- *Initial condition testing* tests whether the activity is created properly.
- *GUI testing* tests whether the activity performs according to the GUI specification.
- *State management testing* tests whether the application can properly enter and exit a state.

Above three kinds of testing are used for identifying activity bugs. For more effective GUI tests, we used an event generation tool. For helping generate GUI events, the Monkey event generator, which comes with the Android SDK, was used. Monkey can generate random or deterministic event sequences and feed these events to the Android application.

Once the test cases are generated, they would be run on the application through the Dalvik VM. To monitor the execution of test cases, we configure the Dalvik VM to log the details of each test case into a trace file. Our traces capture three kinds of events: GUI events, method calls, and exceptions.

4.2. Log File Analysis and Bug Detection

Once test cases are generated for a certain application, we run the application on these test cases and log the performance of each test case so that we can detect errors. With the

log file, we use patterns to identify potential bugs. Each kind of errors has an associated pattern, these patterns can indicate proper operation or bug.

Implementations of Activities consist of responding to events generated by users and the system. Activity bugs stem from incorrect implementation of the Activity class, for example, one activity might be destroyed in the wrong way so that it will make the application crash. In practice, almost every Android application we analyzed has activity bugs because it is hard to check whether each base function has been properly implemented.

An activity has a life cycle described by a state machine shown as Figure 3, Hence violations of this state machine lead to activity bugs.

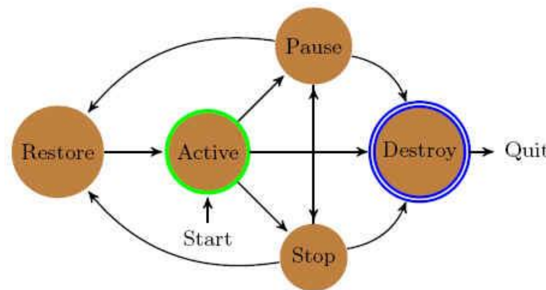


Figure 3. State Machine of an Android Activity

Each activity can be in one of five states: Active, Pause, Stop, Restore or Destroy. To ensure a correct state sequence, the corresponding user defined event methods should be called in a valid order as specified by the state machine. Activity base class contains event methods that govern the life cycle of an activity as follows:

- *onCreate():* Called when the activity is first created
- *onStart():* Called when the activity becomes visible to the user
- *onResume():* Called when the activity starts interacting with the user
- *onPause():* Called when the current activity is being paused and the previous activity is being resumed
- *onStop():* Called when the activity is no longer visible to the user
- *onDestroy():* Called before the activity is destroyed by the system
- *onRestart():* Called when the activity has been stopped and is restarting again

The called relations between event methods are shown as Figure 4.

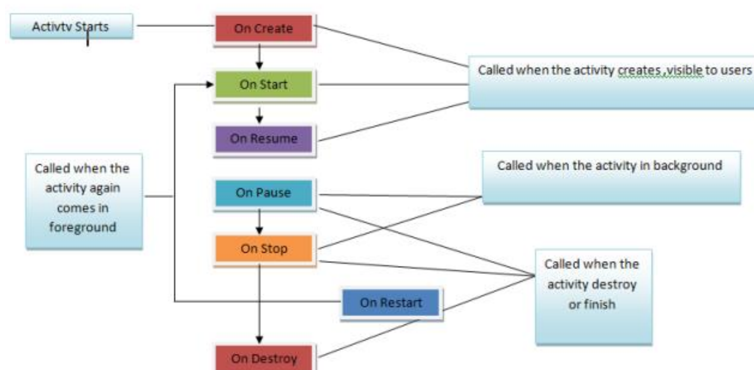


Figure 4. Called Relations Between Event Methods

Copyright ⓒ 2016 SERSC

The state machine as a specification and match event method calls from log file entries against it. Violations of the state machine would be flagged as potential bugs. For example, the correct state sequence Start-> Active-> Pause-> Restore-> Active-> Destroy corresponds to a valid event method order onCreate()-> onPause()-> onResume()-> onDestroy().

For example, we found a new activity bug in ConnectBot. The bug manifests itself as an onCreate() without a subsequent onPause(), which is a violation of the state machine specification. The bug corresponds to a situation where the user sets up a default shell host beforehand and then starts the application, which would crash the application. Figure 5 is a screen shot of the application crash when the scenario described above unfolds.

Figure 5. Screenshot of ConnectBot Activity Failure

Besides Activity bugs, our approach can also detect event bugs and type errors. Android apps should be prepared to receive events and react to events in any state of an activity. If developers fail to provide proper implementations of event handlers associated with certain states, the application can either enter an incorrect state or crash outright. In our approach, we can detect that the application crashes when Monkey feeds it with an unhandled event. While detecting type errors is quite simple, once the type error has been triggered, a ClassCastException entry will appear in the log file.

5. Testing Results

Our verification approach turned out to be effective in practice. We report the number of bugs we found using our approach in Table 3. For each class of bugs we were able to re-discover bugs already reported (the Old columns) as well as new bugs that have not been reported yet (the New columns).

Table 3. Old (Re-Discovered) Bugs and New (Not Previously Reported) Bugs

Application	Activity bugs		Event bugs		Type errors	
	Old	New	Old	New	Old	New
Firefox Mobile	3	0	2	0	1	4
ADW Launcher	1	0	1	2	2	1
DealDroid	2	1	2	3	2	1
SMSPopup	2	1	2	2	2	1
XBMC Remote	2	1	0	0	0	0
AnkiDroid	1	0	0	1	0	0
ConnectBot	0	1	0	0	1	0
CSipSimple	0	2	0	0	1	0
CyanogenMod	1	0	0	1	0	0
GAOSP	2	0	0	0	0	1
IMSDroid	1	2	1	1	0	2
Rokon	2	1	0	0	1	0

We took each event sequence which our approach has found automatically and played it manually, through GUI interaction, to make sure the bug can actually be reproduced in practice. We had reported the new bugs to the developers, two of the bugs have been confirmed, while others are in the process of confirmation.

Acknowledgments

This paper is granted by the national training programs of innovation and entrepreneurship for undergraduates (No. 201511065006).

References

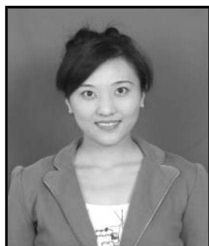
[1] A. Zeller, “Why programs fail: a guide to systematic debugging”, Elsevier, **(2009)**.
[2] A. Kumar Maji, K. Hao, S. Sultana and S. Bagchi, “Characterizing failures in mobile oses: A case study with android and symbian”, Proceedings of Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on, **(2010)**.
[3] C. Hu and I. Neamtiu, “Automating GUI testing for Android applications”, Proceedings of the 6th International Workshop on Automation of Software Test, **(2011)**.
[4] F. Belli, C. J. Budnik and L. White, “Eventbased modelling, analysis and testing of user interactions: approach and case study: Research Articles”, Software Test. Verif. Reliab., vol. 16, no. 1, **(2006)**, pp. 3-32.
[5] P. Hooimeijer and W. Weimer, “Modeling bug report quality”, Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, **(2007)**.
[6] N. Bettenburg and S. Just, “What makes a good bug report”, Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, **(2008)**.
[7] JUnit, May 2010. http://www.junit.org/.
[8] Monkey UI/Application Exerciser, May 2010. http://developer.android.com/guide/developing/tools/

Authors

Yi Bie, is currently an undergraduate in the S College at Qingdao University. Her main research interests include Electronic Commerce, Android programming, digital media technology and data mining.

 Copyright ⓒ 2016 SERSC

Gengxin Sun, received his Ph.D. degree in Computer Science from Qingdao University, China in 2013. He is currently an Associate Professor in the School of Computer Science and Engineering at Qingdao University. His main research interests include digital media technology, Android programming, complex networks, web information retrieval and data mining.



Sheng Bin, received her Ph.D. degree in Computer Science from Shandong University of Science and Technology, China in 2009. She is currently a lecturer in the School of Software Technology at Qingdao University, China. Her main research interests include digital media technology, Android programming, complex networks, cloud computing and data mining.

Xicheng Zhou, is currently an undergraduate in the International College at Qingdao University. His main research interests include Electronic Commerce, Android programming, digital media technology and data mining.

Copyright ⓒ 2016 SERSC