Systematic Exploration of Android Apps' Events for Automated Testing

Ibrahim Anka Salihu
Universiti Tun Hussein Onn Malaysia (UTHM)
86400 Parit Raja, Batu Pahat
Johor, Malaysia
hi130015@siswa.uthm.edu.my

Rosziati Ibrahim
Universiti Tun Hussein Onn Malaysia (UTHM)
86400 Parit Raja, Batu Pahat
Johor, Malaysia
rosziati@uthm.edu.my

ABSTRACT
The popularity of mobile devices is ever increasing which led to rapid increase in the development of mobile applications. GUI testing has been an effective means of validating Android apps. However, it still suffers a strong challenge about how to explore event sequence in the GUIs. This paper proposes a hybrid approach for systematic exploration of mobile apps which exploit the capabilities of both static and dynamic approaches while trying to improve app's state exploration. Our approach is based static analysis on app's bytecode to extract events supported by an app. The generated events are used to dynamically explore an app at run-time. The experimental results show that our approach can explore significant number of app's state for the generation of high quality test case.

Categories and Subject Descriptors
D.2.4 [Software Engineering]: Software/Program Verification—Reliability, Validation; D.2.5 [Software Engineering]: Testing and Debugging—Testing tools

General Terms
Algorithms, Reliability, Verification.

Keywords
Android app, GUI testing, Hybrid testing, Dynamic analysis, Static analysis, Systematic exploration, Test case generation, Code coverage.

1. INTRODUCTION
The introduction of mobile devices (smart phones and tablets) has brought a paradigm shift from desktop computers to mobile devices. Nowadays, mobile devices are briskly replacing traditional computers (in popularity and usage) for an increasing number of users in several areas such as access to emails, mobile banking, e-services, social networks etc. The popularity of these devices has brought a huge increase in the development of mobile applications (Mobile Apps) by software developers in recent years [19] to meet the respective needs of their users.

Mobile applications (Mobile Apps) are software systems designed for mobile devices (smart phones, tablets and other handheld devices). Though, mobile apps are initially simpler and smaller with less complex design architecture and having small set of functionalities, they are recently increasing in capacity, functionality, structure and behavior [14], hence becoming more and more complex [17]. Mobile apps development has a significant impact from both economic and social perspectives. It has generated revenue of $4.5 billion in 2009, and a recent report estimated that the global applications business will be worth $77 billion by 2017 [10, 17]. Due to the reliance on mobile apps in everyday life, there is a need to ensure mobile apps' quality such as correctness, performance and security [5, 8, 13, 21]. Nonetheless, the increased complexity of mobile apps has brought several challenges for the software engineering researchers such as understanding apps behavior and testing [4, 24, 26].

Software testing as one of the important stage in application's development lifecycle can play a significant role in improving the quality of software systems [2]. Specifically test automation is essential in testing today's software application due to its ability to save time and cost, and it is error prone. Test automation tool can only be efficient when it explores an app extensively to cover reasonable number of app state. Several techniques were proposed for automated testing of Android apps [1, 3, 4]. However, the coverage by existing techniques is not comprehensive due inability to explore apps state extensively [25]. This is due to the nature of the Android platform (framework-based) where many of the app's behaviors reside in the Android framework. Furthermore, most of the techniques are based on dynamic analysis/exploration to extract event sequence that can be fired to explore an app. However, the information extracted by pure dynamic approaches is incomplete which affects app's coverage [6, 15, 23].

To deal with these challenges, we proposed a hybrid approach that combines static and dynamic analysis to improve the exploration of application's state. Our proposed technique is built on top GATOR [11] which performs a control flow analysis on the bytecode code of an app to generate windows transition graph (WTG). The WTG is used to extract all supported events that are used to explore an app at run-time.

The paper is organized as follows. Section 2 presents an overview of Android apps and the structure of Android apps. We discussed our proposed approach in section 3. Section 4 presents an overview of our tool that implement the hybrid approach. In section 5, we discussed our experimental results. Section 6 discus

MoMM '16, November 28-30, 2016, Singapore, Singapore.

DOI: http://dx.doi.org/10.1145/3007120.3011072

sed the related works and section 7 is the conclusion.

2. Overview

This section presents an insight into the android framework and apps development. The Android operating system designed for mobile devices is based on the Linux Kernel by Google. According to the Android developers' website [34], the platform provides an API with several components (namely, Activity, Service, Broadcast Receiver and Content Provider) that can be re-used to define and support the development of Android apps. Figure 1a shows the architecture of Android platform.

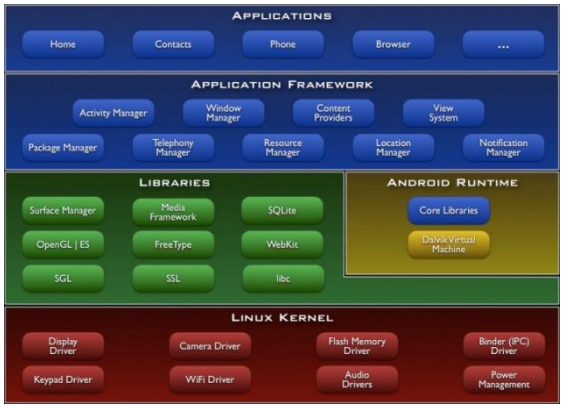


Figure 1a. Architecture of Android platform

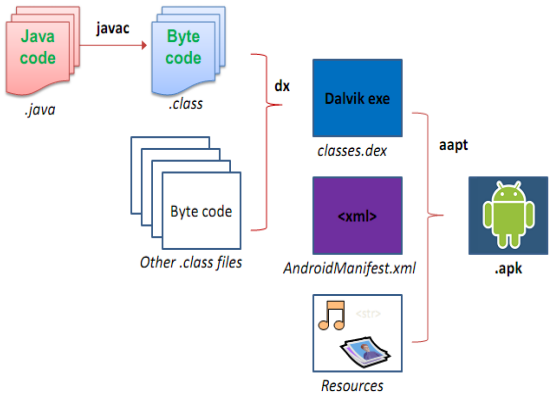
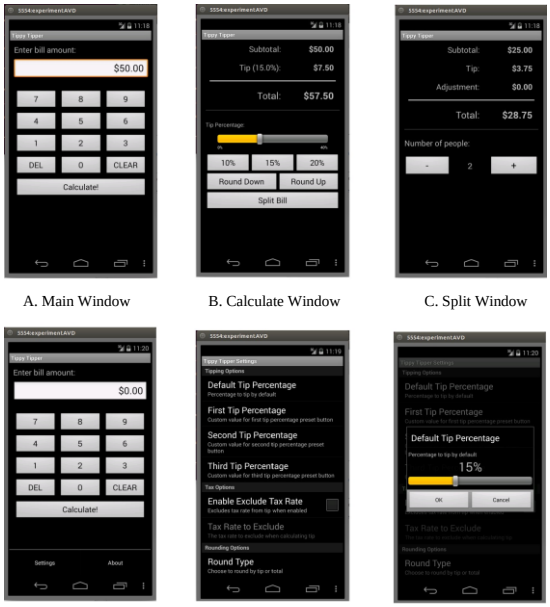


Figure 1b. Application Package (APK) structure

Android apps are developed using Java programming language which runs on a special virtual machine called Dalvik Virtual Machine. They are compiled to the .dex file which comprises of the Java code, the resource files and AndroidManifest.xml file. Figure 1b shows the structure of Android package. Unlike apps on typical systems, the android apps do not have a single program entry point for everything. Instead, the essential components (Activities and Services) provided by the API can be instantiated and run when needed. An app usually comprises of one or several activities that extend the base activity class. The Activity component presents a visual user interface for each focused task a user can perform. We refer to the activities as app's windows. An app also consists of other windows such as menus and dialogs which can significantly affect its lifecycle. An activity instance can programmatically start other activities which are usually referred to as intent in the android framework. Android apps also support a wide range of user gestures such as long-press, double-taps, swiping and pinching by using the Gesture Detector class in Android framework.



A. Main Window B. Calculate Window C. Split Window

D. Main menu E. Sub-menu F. Dialog

Figure 2. An illustration of windows transition in TippyTipper app

Interaction with an app by the user triggers transition across various windows of the app. Figure 2 illustrates windows transition in tippytipper app. Figure 2d shows click on the menu. Clicking the settings triggers the sub-menu window while selecting a check box from the sub-menu triggers the dialog window.

3. PROPOSED APPROACH

This section presents our approach for the systematic exploration. First, we used static analysis on app's bytecode to generate Windows Transition Graph (WTG). The static analysis is designed to extract events supported by the app which can be used to invoke the activities. Then we perform the run-time exploration of an app using the WTG as input.

3.1 Static analysis

The static analysis was implemented using GATOR to generate WTG. The WTG comprises of all windows (including menus and dialogs) that made up an app and events supported by the app. A node in the WTG represents app's window/activity while an edge (considered as path) is representing transition from one window to another. As highlighted by several studies [22] traditional graph algorisms such as DFS and BFS are not efficient in traversing weighted di-graphs, therefore our proposed approach is based greedy algorithm to traversed the extracted WTG and generate a sequence of events that can be used to invoke all activities/windows of an app. Greedy algorithm makes sequential decision based on a defined heuristics (choosing the local best option at each step) with the aim to yield optimal value. It is good

for selection/branching problems and can reduce memory usage as well as run-time.

3.2 Systematic Exploration

We now describe how we perform the systematic exploration using WTG as input. Algorithm 1 presents a precise description of the systematic exploration. Parts of the algorithm are responsible for traversing the graph and the other parts perform the exploration on an emulator. It essentially traverses the graph to extract all paths in the graph and creates event list supported by the app (Line 2-7) and then starts the exploration from the launcher node of an app (Line 11-13).

Algorithm 1. Systematic Exploration

```
Input: WTG g = (P, S), A: app under test
1  Procedure SysExploration(wtg)
2    P ← getAllpaths(g)
3    for all P in g  do
4      while P ≠ explored then
5        EventSet E_s ← get_min_P of wtg
6        for all events ∈ E_s do
7          W ← getSourceOfEvent()
8          SystematicEXPLORATION(g, A)
9        end for
10       GUIWidgetSet ← getGUIWidgets(W)
12       foreach fireable widget ∈ widgetSet do
13         fire GUIWidgets
14       end
15     end
15   end
16 end
```

The algorithm starts with the WTG as input. First, we extract all events supported by an app (Line 5) and then backtrack to locate the source of each event (Line 7). We then extract the GUI widgets from the source of an event W (Line 11) and start the exploration at line 13.

4. TOOL IMPLEMENTATION

We implemented our systematic exploration in a tool called AMOGA. It comprises of an event mapping section that is responsible for extracting all app's supported events statically. Figure 2 presents an overview of AMOGA.

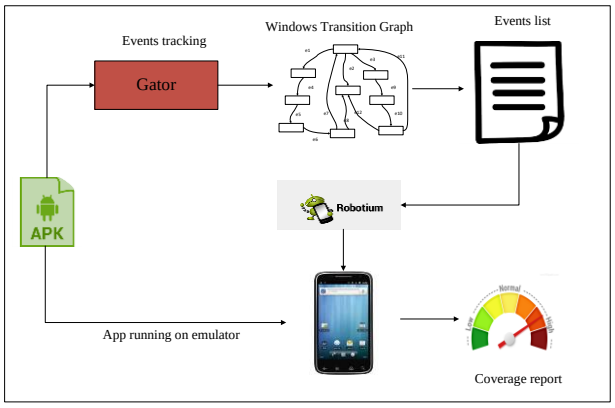


Figure 2. Overview of Systematic Exploration in AMOGA

4.1 Event Mapping

The static analysis is implemented using GATOR [11] a static analysis tool for Android. GATOR performs control flow analysis Android GUIs and callback methods. To analyze an app AMOGA, receives and reverse engineer the .apk of an app to bytecode using GATOR (which is in turn using apktool). The analysis then starts on the app's byte code to generate the WTG of app.

4.2 Events Exploration

The run-time exploration is built on top of Robotium testing framework [12] which has the capability to extract GUIs (such as test views, check boxes, buttons, spinners etc.) and fire action on the event handlers. It also has functionality for editing and clearing text boxes, clicking on home, menu and back button. To dynamically explore an app, GUI widgets extraction is required. Our tool statically extracts the GUI widgets of an app using Robotium. As described in previous section, the exploration algorithm backtracks on each path of the WTG to systematically extract the GUI widgets associated with each event and fire action on the GUI to trigger the event.

4.3 App's Coverage

A variety of mobile apps testing tools leverage the Emma [7] code coverage libraries to measure the coverage attained. It is an open-source tool for measuring and recording the code coverage during testing of Java applications which has been recently included in the Android SDK. Emma provides coverage reports at the class, method, line and basic block levels, and the coverage statistics are collected at method, class, package, and "all classes" levels [7]. We integrated Emma in AMOGA to generate coverage report used for accessing the efficiency of the tool.

5. RESULTS AND DISCUSSION

To assess the efficiency of our approach, we conducted experiments on several mobile applications from Google Play. We selected five (5) open source apps across different categories (such as productivity, business etc) that cover a range of popular, real-world mobile apps and were used in previous works on Android automation testing. Table 1 presents the characteristics of selected apps. The lines of code in an app is shown in column 2, the Activities in column 3 and the last column shows the number of downloads based on Google Play analytics as of September, 2016.

Table 1. Overview of apps used in evaluation

App	#LOC	Activities	Category	Download
TippyTipper	2248	6	Tool	100K – 500K
OpenManager	3647	6	Business	5M – 10M
Notepad	8172	8	Productivity	500K – 1M
TomDroid	5038	5	Business	10K – 50K
AardDict	5097	7	Books & Ref.	10K – 50K

Code coverage and exploration time: Table 2 presents the percentage code coverage obtained on 5 different apps and the total exploration time. Column 1 shows the code coverage result obtained which shows that our tool achieved coverage of 70% - 91% on the set of five (5) apps used. Column 2 show the static analysis time required to generate the event list and column 3 contains the dynamic exploration time which we measured by allowing the exploration to run to completion time.

Table 3. Apps code coverage and exploration time

App	Coverage (%)	Events Mapping (Sec)	Exploration (Min)
TippyTipper	81	5.2	91
OpenManager	75	5.0	278
Notepad	91	5.5	49
Tomdroid	80	4.9	27
AardDict	71	6.8	53

We compared our tool against the code coverage with other existing Android exploration tools. Table 3 shows comparison of our tool with other tools (Android monkey tool, Android GUI Ripper and Orbit). The code coverage reports from each run indicated that our tool obtained higher coverage than other tools.

Table 3. Comparison of coverage with other tools

App	Monkey	Android GUIRipper	Orbit	AMOGA
TippyTipper	41	-	78	81
OpenManager	29	-	63	75
Notepad	60	-	82	91
Tomdroid	46	40	70	80
AardDict	51	27	65	71

6. RELATED WORKS

Several approaches and tools were proposed both from the industry and academia for automated testing of Android apps. The work in [16] has categorized these tools into five main groups. They are; Fuzz/Random testing (RT), record and replay (R&R), systematic exploration-based testing, event-sequence generation and frameworks for GUI tests automation. Our work falls under the group of systematic exploration-based testing. This is an approach where exploration of the AUT is driven automatically by a GUI model (extracted before the exploration or iteratively after a new event is executed) that describes the UI, GUI widgets, and feasible actions on each widget [16]. A variety of tools are available based on dynamic analysis of hybrid static/dynamic approaches.

Memon et al. proposed GUITAR [20] tool for testing desktop applications which is centered on event-based modeling of applications to automate GUI exploration. Android GUITAR [1] is an extension of the GUITAR designed for the Android apps. The tool models the GUI of an app as event integration graph (EIG) representing sequences of user actions that can be fired on the GUI. Firstly, GUITAR is not able to capture the rich set of user inputs associated with mobile app (such as swiping, pinching etc). Secondly, it produces many false event sequences which may need to be weeded out later and it is not able to explore all the GUIs due to the presence of infeasible paths.

Amalfitano et al. [3] proposed AndroidRipper tool for automated testing of Android apps. AndroidRipper dynamically analyses app's GUI to obtain event sequences that can be fired on the GUI, with each sequence representing an executable test case that can be used for regression test and crash testing. The techniques can sometimes lead to unexpected faults and the exploration is not comprehensive which can be associated with the limitations of pure dynamic analysis to identify the events.

Monkey [18] is a testing tool available in the Android SDK that can produce and send events in both random and deterministic ways to an app. Random event are effectively used for stress testing and fuzz testing of an app. However, they are not effective for systematic explorations as in our case.

Some techniques that utilized the hybrid approach also exist such as ORBIT [26] and A3T [4]. The ORBIT tool performs static analysis on the source code of an app to generate set of user actions supported by an app. A dynamic crawler (built on top of Robotium) is used to fire actions on the GUI objects of a running app and extract a state model that can be used for model-based testing. In our technique, we used static analysis on the bytecode of an app (considering that source code of mobile app is rarely or not available) to extract all supported events and used them for the systematic exploration.

Automatic Android App Explorer (A3T) used static analysis specifically data flow analysis (taint tracking) on the bytecode of an app to construct static activity transition graph (SATG) with nodes representing activities and edges indicate the possible transitions between the activities (UI screens). The tool used the static analysis in ScanDroid [9] to construct the SATG, which is subsequently used for the automatic exploration of an app. However, this graph representation does not capture menus/dialogs, does not model the windows stack and its state changes as such therefore it is not comprehensive. In A^3T tool the evaluation is based on activity and method coverage, but our tool is evaluated in-terms of code coverage, therefore our technique is not directly comparable.

7. CONCLUSION

In this paper, we have presented our approach for systematic exploration of Android applications for automated testing. It consists of static analysis that generates a list of application's supported events and performs a systematic exploration on the application at run-time. We described our prototype tool called AMOGA that implements our approach and presented an empirical evaluation of the tool. The experimental results show that our tool can explore significant number of app's state compared with existing tools. Our next line of future work would be to apply our approach for automated model generation from mobile app to support model-based testing of an app.

8. ACKNOWLEDGMENT

We would like to acknowledge the support from UTHM in undertaking the research, under the Graduate Research Incentive Grants (GIPS), Vote U308, Universiti Tun Hussein Onn Malaysia (UTHM).

9. REFERENCES

[1] Atif Memon. Android GUITAR. September, 2016. https://sourceforge.net/projects/guitar/.

[2] Amalfitano, D., A.R. Fasolino, and P. Tramontana. 2011. A gui crawling-based technique for android mobile application testing. 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), IEEE.

[3] Amalfitano, D., et al. 2012, Using GUI ripping for automated testing of Android applications, in Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. ACM: Essen, Germany. p. 258-261.

[4] Azim, T. and I. Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of android apps. in Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications. ACM.

[5] Bhattacharya, P., et al. 2013. An empirical analysis of bug reports and bug fixing in open source android apps. in Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on. IEEE.

[6] Coimbra Morgado, I., A.C. Paiva, and J. Pascoal Faria, 2012. Dynamic Reverse Engineering of Graphical User Interfaces. International Journal On Advances in Software, 5(3 and 4): p. 224-236.

[7] Emma, An open source Java code coverage tool. September, 2016. http://emma.sourceforge.net/.

[8] Enck, W., et al., TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. 2014.ACM Transactions on Computer Systems (TOCS), 32(2): p. 5.

[9] Fuchs, A.P., A. Chaudhuri, and J.S. Foster, 2009. Scandroid: Automated security certification of android applications. Manuscript, Univ. of Maryland, http://www. cs. umd. edu/avik/projects/scandroidascaa, 2(3).

[10] Gartner, I., Mobile Apps Will Be a Vehicle for Cognizant Computing. September, 2016. http://www.gartner.com/newsroom/id/2654115.

[11] GATOR, Program Analysis Toolkit For Android. September, 2016. http://web.cse.ohio-state.edu/presto/software/gator/.

[12] GoogleCode, Robotium. September, 2016. http://code.google.com/p/robotium.

[13] Hu, C. and I. Neamtiu. 2011. Automating GUI testing for Android applications. in Proceedings of the 6th International Workshop on Automation of Software Test. ACM.

[14] Islam, R., R. Islam, and T. Mazumder, 2010. Mobile application and its global impact. International Journal of Engineering & Technology (IJEST).

[15] Kull, A. 2012. Automatic GUI Model Generation: State of the Art. IEEE 23rd International Symposium on in Software Reliability Engineering Workshops (ISSREW).

[16] Linares-Vásquez, M. 2015. Enabling testing of android apps. in Proceedings of the 37th International Conference on Software Engineering-Volume 2. IEEE Press.

[17] Minelli, R. and M. Lanza. 2013. Software Analytics for Mobile Applications-Insights & Lessons Learned. in Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on Software Maintenance and Reengineering.

[18] Monkey, UI Application Exerciser. September, 2016. http://developer.android.com/guide/developing/tools/monkey.html.

[19] Nayebi, F., J.-M. Desharnais, and A. Abran. 2012. The state of the art of mobile application usability evaluation. in CCECE.

[20] Nguyen, B., et al. 2014. GUITAR: an innovative tool for automated testing of GUI-driven software. Automated Software Engineering, 21(1): p. 65-105.

[21] Rountev, A. and D. Yan, 2014. Static Reference Analysis for GUI Objects in Android Software, in Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization. ACM: Orlando, FL, USA. p. 143-153.

[22] Salva, S. and S.R. Zafimiharisoa, 2014. Model Reverse-engineering of Mobile Applications with Exploration Strategies. In Proceedings of the 9th International Conference on Software Engineering Advances (ICSEA), October 12-16, 2014, Nice, France.

[23] Silva, C.E. and J.C. Campos, 2013. Combining static and dynamic analysis for the reverse engineering of web applications, in Proceedings of the 5th ACM SIGCHI symposium on Engineering interactive computing systems. ACM: London, United Kingdom. p. 107-112.

[24] Yang, S., et al. 2015. Static control-flow analysis of user-driven callbacks in Android applications. in International Conference on Software Engineering (ICSE).

[25] Yang, S., et al. 2015. Static Window Transition Graphs for Android (T), in Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE Computer Society. p. 658-668.

[26] Yang, W., M.R. Prasad, and T. Xie, 2013. A grey-box approach for automated GUI-model generation of mobile applications, in Fundamental Approaches to Software Engineering. Springer. p. 250-265.