2015 IEEE 39th Annual International Computers, Software & Applications Conference

PATS: A Parallel GUI Testing Framework for Android Applications

Hsiang-Lin Wen, Chia-Hui Lin, Tzong-Han Hsieh, and Cheng-Zen Yang
Department of Computer Science and Engineering
Yuan Ze University, Chungli, Taiwan, 320
{hlwen13,chl15}@syslab.cse.yzu.edu.tw; s1001421@mail.yzu.edu.tw; czyang@syslab.cse.yzu.edu.tw

***Abstract*—Android is currently the most widely used operating system for mobile devices. GUI testing for Android applications becomes an important research area in which many studies have been conducted. The past studies show that testing a complicated GUI design may need a large number of test cases which increases exponentially due to the complexity of the GUI. Developers then need to spend a large amount of testing time in executing the test cases to explore the potential software defects. Unfortunately, the testing efficiency issue has not been comprehensively discussed in related studies. In this paper, we describe a parallel GUI testing platform called PATS (Parallel Android Testing System) which performs GUI testing based on a master-slave model. In PATS, the application under test is analyzed dynamically under the cooperation of the master and the slaves. Since the test cases are also generated in parallel at the runtime, the testing efficiency can be improved. We have implemented a prototype and conducted experiments with Android apps. The experimental results show that PATS can effectively improve the testing time with 18.87-35.78% performance improvements.**

Keywords*—*Android GUI testing; parallel testing; GUI ripping; fine-grained event sequences

I. Introduction

Android is currently the most widely used operating system for mobile devices. The development of Android applications proliferate due to its popularity. As shown in the recent statistics at the end of 2014 [1], the number of Android applications exceeds 1.4 million. Many courses have also been provided to teach the Android system and Android software programming [2], [3]. Since more and more users download and use these applications to manage their activities, such as daily schedules and online shopping, the issues of reliability and user-friendliness are important design requirements for these applications. In such a circumstance with the proliferation of Android applications, *GUI testing* has become a vital role to verify the operation requirements of GUI components. As described in [4], GUI testing is the testing process based on the GUI components of the application under test (AUT), and is the only approach for GUI software testing at the system level. For any GUI software, GUI testing is effective for detecting both GUI and non-GUI defects [5], [6]. Recently, many studies have been conducted in the GUI testing field.

In Google's development environment of Android programming, several testing tools have been provided, such as Monkeyrunner [7] and JUnit [8]. In addition, various automated testing tools and frameworks have been devised to help to mitigate the tedious testing process, such as Robotium [9] and GUITAR [4]. However, most GUI testing tools and frameworks simply focus on mitigating manual effort in the GUI testing process rather than improving testing efficiency for the overwhelming testing complexity of modern GUIs. For example, the experimental results in [10] show that a GUI testing process needs less than three hours for a dictionary app with 12 testing sessions. As pointed out in [4], the number of GUI test cases may increase exponentially due to the size and complexity of the GUI and the user events. Therefore, exhaustive and analytical testing approaches is generally infeasible.

A few research studies have addressed the efficiency issue by parallelizing the testing and analysis process with a distributed framework. However, the distributed testing processes in the current well-known frameworks are mainly of a coarse-grained testing model by separately executing each complete testing event sequence on a standalone AUT. In such a coarse-grained scenario, slave nodes are independent in the testing process. For example, Andlantis is a dynamic analysis framework that can schedule and execute many Android app instances in parallel on a cluster of analysis slave nodes [11]. Each app instance is independently executed on a standalone virtual machine. Therefore, it is possible to repeatedly execute the same testing cases on different slaves in Andlantis. In GUITAR, a distributed testing process has been support to dispatch the test cases to a cluster of slave nodes for AUT testing [4]. However, all test cases are generated at a centralized controller. The controller is responsible for generating and scheduling all event sequences of test cases based on the GUI tree and the event-flow graph. Therefore, the controller may become the performance bottleneck in GUITAR.

In this paper, we describe a fully distributed GUI testing framework design called PATS (Parallel Android Testing System) which parallelizes the testing process in a fine-grained model on a cluster of testing nodes. That is, executing a single testing event sequence may involve a set of distributed testing nodes. Each testing node uses a black-box approach to generate a segment of the testing sequence. The generation of the event-flow graph in PATS is accomplished under the collaboration of the controller and all slave nodes. The slave nodes are responsible for analyzing the assigned UI interface and dynamically deciding the short-term testing event sequences. A coordinator congregates these short-term event sequences and dispatches them to the slave nodes for further testing and generating new short-term event sequences. The redundancy of event sequences can be recognized to improve the testing efficiency by avoid repeating these redundant sequences.

We have implemented a prototype and conducted testing experiments with four Android apps to investigate the effec-

0730-3157/15 $31.00 © 2015 IEEE
DOI 10.1109/COMPSAC.2015.80

tiveness of the proposed framework. These four apps comprise a student programming exercise and three publicly released apps in Google Play. The experimental results have shown that PATS can effectively improve the testing time with 18.87-35.78% performance improvements.

The rest of this paper is organized as follows. Section 2 first gives a brief overview of relate work. The details of the design of PATS are elaborated in Section 3. Section 4 presents the experiments and the evaluation results. Finally, Section 5 concludes this paper and presents the future work.

II. Related Work

Recently, research on Android GUI testing has highly obtained attention. Many approaches have been proposed and devised recently. In 2011, Hu and Neamtiu proposed an automated GUI testing scheme based on JUnit and Monkey to execute the random and deterministic events in the AUT. [12]. However, random testing is unlikely to explore every combination of user events. Takala et al. designed a model-based GUI testing tool based on TEMA [13] for Android applications [14]. In TEMA, a test designer manually specifies the state machines for tests. Then, optical character recognition (OCR) is used to verify the state of the GUI. This approach focuses on the system-level model-based GUI testing, rather than the testing efficiency.

A reverse engineering approach is used in GUI Ripper [10], [15]. The GUI structure is crawled dynamically to create a GUI tree as the state model. Then, GUI Ripper automatically generates test cases based on the GUI structure. To avoid infinitely exploring visited GUI states, GUI Ripper checks the state equivalence and the depth threshold. This approach provides a broad testing coverage by going through all possible event sequences. However, the crawling approach in GUI Ripper needs a considerable amount of time to execute the tests [10].

Anand et al. propose a Android GUI testing system focusing on event generation based on the concolic testing technique [16]. This system avoids the path-explosion problem by checking the execution conditions. Mirzaei et al. develop Symbolic Pathfinder (SPF) based on Java PathFinder (JPF) to test Android applications using the symbolic execution technique [17]. Therefore, SPF can systematically generate test cases and a mock class is introduced in SPF to tackle path-divergence problem. Due to the characteristics of symbolic execution, SPF needs to analyze the application source code. This limits the applicability of SPF if it is difficult to obtain the source code.

Choi et al. propose a testing mechanism SwiftHand to learn the GUI model based on the State-merging DFA (Deterministic Finite Automaton) induction machine learning algorithm [18]. In SwiftHand, The application source code needs to be inject some instrumentation code to learn the behavior pattern. Although SwiftHand can get detailed behavior information, the behavior may be changed due to the instrumentation code. Lin et al. develop a SPAG/SPAG-C (Smart Phone Automated GUI testing tool) testing framework with the record-replay approach for Android GUI testing [19], [20]. They use the image processing technique to record the testing process and replay the recorded test cases. The SPAG/SPAG-C framework focuses on the generation of accurate test oracles. The replay process still needs manual operations.

Recently, a few research studies have noticed the testing efficiency issue. Mahmood et al. propose a cloud architecture for security testing [21]. This approach is a white-box approach which needs source code to add signatures. In [11], Andlantis is proposed to schedule and execute many Android app instances in parallel on a cluster of analysis slave nodes. However, there is no cooperative mechanism between the slave nodes in Andlantis. If the same AUT is executed on two different slaves, the same testing cases may be repeatedly executed. In [4], GUITAR is comprehensively introduced with an distributed extension to distribute and execute test cases on a cluster of slave testing nodes. However, GUITAR uses a centralized controller to generate all testing event sequences and dispatch them to the slaves. Each event sequence is then executed individually on a slave node. Although the problem of repeatedly executing the same testing sequences is avoided in GUITAR, the controller may become the centralized performance bottleneck.

III. Testing Framework Design

In this section, we first present the architecture overview of PATS. PATS consists of two kinds of testing nodes: a testing coordinator (TC) and a set of testing slaves (TS). As GUI Ripper, PATS dynamically analyzes the GUI components to generate test cases. The main difference is that PATS generates short-term testing event sequences on the set of slave nodes. TC then congregates these short-term event sequences and schedules them to the slave nodes. The detailed node structures and operations are then elaborated.

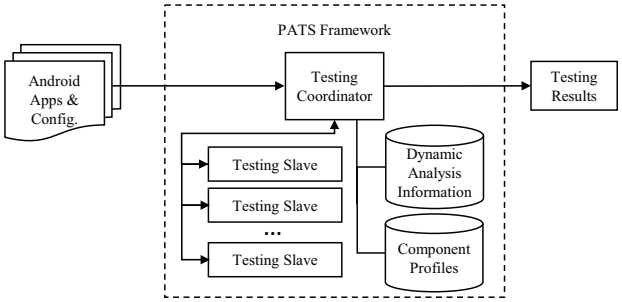

Fig. 1. The testing architecture of PATS.

A. Architecture and Workflow Overview

Figure 1 illustrates the architecture overview of the proposed PATS framework. When TC gets the AUT file and the testing configuration, it uses the ripping technique to analyze the first GUI screen from app's (*MainActivity*) to initiate the following parallel GUI testing. In PATS, the *UI hierarchy* of GUI components in each different GUI screen is defined as a *UI-state*. Each UI-state has the corresponding set of user events, such as clicking and scrolling, according to the GUI components in the UI hierarchy. For example, Figure 2 shows a snippet of the parsed XML file of the root UI-state for app HandyTimetable [1]. Each `node` tag contains the

[1]https://play.google.com/store/apps/details?id=com.newbitmobile.handytimetable

```
<hierarchy rotation="0">
   <node index="0" text="" resource-id="" class="android.widget.FrameLayout"
   package="com.newbitmobile.handytimetable" content-desc="" checkable="false"
   checked="false" clickable="false" enabled="true" focusable="false"
   focused="false" scrollable="false" long-clickable="false" password="false"
   selected="false" bounds="[0,0][768,1184]">
      <node index="0" text="" resource-id="" class="android.widget.LinearLayout"
      package="com.newbitmobile.handytimetable" content-desc="" checkable="false"
      checked="false" clickable="false" enabled="true" focusable="false"
      focused="false" scrollable="false" long-clickable="false" password="false"
      selected="false" bounds="[0,0][768,1184]">
      ...
      </node>
   </node>
</hierarchy>
```

Fig. 2. A snippet of the parsed XML file of the root UI-state for HandyTimetable.

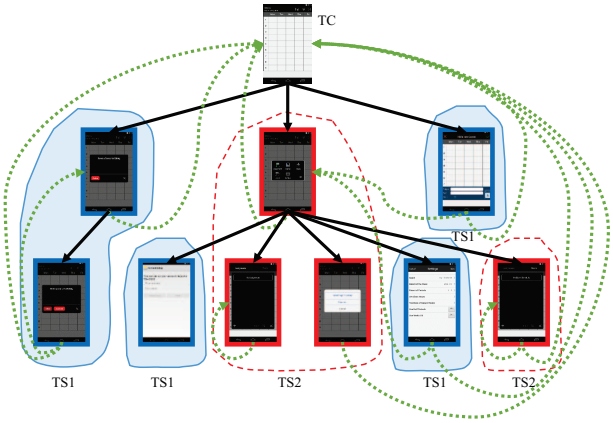

Fig. 3. A GUI-tree subset example of an Android app.

action information for a GUI component. In Figure 2, the first component is a `FrameLayout`, and the second is a `LinearLayout`. Since the GUI components of Android can be the containers of other GUI components, each UI-state is represented with the analyzed hierarchy. The dynamic analysis information, such as UI-states and user events, is stored for the following testing process.

Figure 3 illustrates a GUI-tree subset in PATS with HandyTimetable which has 10 UI-states in the top-tree levels of the GUI tree. The dashed arrows represent the possible user events that will change the UI-state. TC first analyzes the app's MainActivity, finds three second-level UI-states, and generates their testing scripts. Then TC dispatches these second-level UI-states with the corresponding scripts to two testing slaves TS1 and TS2 in order to explore the following UI-states. TS1 and TS2 find six third-level UI-states and reply them with testing scripts to TC for state aggregation. TS1 and TS2 also verify the testing requirement for these UI-states. In the example illustrated in Figure 3, TS1 tests 5 UI-states and TS2 tests 4 UI-states.

For the path-explosion problem, PATS will assign a unique identifier to each newly generated UI-state according to its structure information. In UI-state aggregation, TC checks the state equivalence by comparing each new UI-state ID with the historical UI-state IDs. If the new UI-state ID has appeared, PATS does not explore this UI-state further to avoid infinite path exploration in GUI tree traversal. However, the GUI design in some applications may innately have infinite UI-states. In this case, a time limit is used to control the maximum testing time. After TC inspects the new UI-states, it schedules the untested UI-states to testing slaves for further state exploration. This parallel GUI ripping process is repeated until all individual GUI-states are explored or the time limit is reached. Along with the GUI ripping process, fine-grained test cases are executed for all GUI components.

B. Node Operations

The detailed node structures of TC and TS are illustrated in Figure 4. The components in the TC and TS nodes are introduced as follows:

- **TC task controller** The task controller in TC is responsible for the primary management work in PATS. It has three main tasks in the testing process. First, the TC controller maintains the AUT files and the testing results. It retrieves the AUT file and the testing configuration from the repository and saves the testing results to the repository. Second, it analyzes the app's MainActivity to extract the root UI-state and initiates the testing process by invoking the script generator to generate the corresponding testing scripts. Finally, it schedules the testing jobs by congregating unexplored UI-states and dispatching them with the corresponding testing scripts to the available TS controllers.

  TC also checks the state equivalence to avoid the path-explosion problem. If there is no unprocessed testing script, and all TS nodes are idle, TC closes the current testing process. If the app has infinite UI-states, TC will stop the testing process when the time limit is reached.

- **TS controller** The TS controller is responsible for managing the testing work in PATS with a stateless design. A replay mechanism is adopted to initialize the UI-state of TS for testing. When the TS controller receives a UI-state and a test script from TC, it

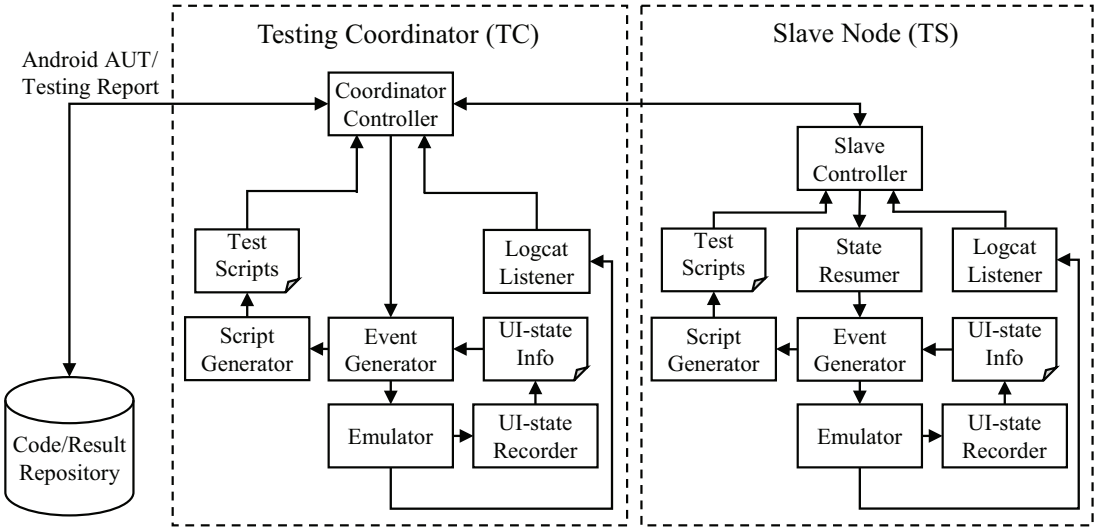

Fig. 4. The structures of the coordinator and the slave node.

passes them to the state resumer to invoke the event generator which initializes the emulator to the UI-state by replaying the testing script. In addition, it replies the generated UI-states and testing scripts to TC for state aggregation.

- **Event generator** The event generator is used to generate user events according to the GUI components in the UI hierarchy and the commands of the state resumer. In addition, it is also used to detect the changes of UI-states.
- **Emulator** The emulator provides a virtual machine environment to react the user events issued by the event generator.
- **Script generator** When the script generator receives the state-change information from the event generator, it will generate the corresponding testing script based on the events of the previous UI-state. The script is then sent to the controller for testing job scheduling.

  Figure 5 illustrates a testing script example for the root GUI-state of HandyTimetable. The script generator converts the testing event sequences to the corresponding ADB (Android Debug Bridge) commands with the GUI-state IDs.
- **UI-state recorder** The UI-state recorder uses Android's UI-Automator to analyze the current screen layout by issuing the `Dump` instruction. The analysis result is converted to an XML-based UI-state which is passed to the event generator to issue events.
- **Logcat listener** Android provides a logcat mechanism for collecting and viewing system debugging information. Therefore, the logcat listener periodically check the logcat information. If it finds error messages, it generates the bug information to the controller.
- **State resumer** The state resumer is used to initialize the UI-state of the emulator based on the received UI-state. It then sends the testing script to trigger the event generator.

```
<GUI-state id="642aeee9b7d86fd59dd8eae7c8d84f75">
adb shell input touchscreen tap 716 94
</GUI-state>
<GUI-state id="883cf1d23117e048742a81a5f8c05ba1">
adb shell input touchscreen tap 620 94
</GUI-state>
<GUI-state id="ef1a91c5bd6194dc82ddb553ae07e36e">
adb shell input touchscreen tap 524 94
</GUI-state>
```

Fig. 5. A testing script example for the root GUI-state of HandyTimetable.

Figure 6 illustrates the details of the interactions between the coordinator and the slaves. If TC does nothave any unprocessed testing script, a nd it finds that all TS nodes are idle, the testing process for the current AUT is closed.

IV. Experiments

We have implemented a prototype consisting of a TC and two TSs. The TC node has an Intel Dual-Core E7300 CPU running at 2.66GHz with 4GB memory. The TS nodes have Intel Quad-Core Q8600 CPUs running at 2.66GHz with 4GB memory. All nodes run Ubuntu Linux 14.04 with Genymotion 2.3.1 emulators. We conducted testing experiments on four Android applications:

- **SceneNavigator**: This is a student programming work using simple GUI components to introduce five landscape scenes. This app has the fewest UI-states in the testing process.
- **HandyTimetable**: This is a timetable application for students. This app has been downloaded more than 1,000,000 times.

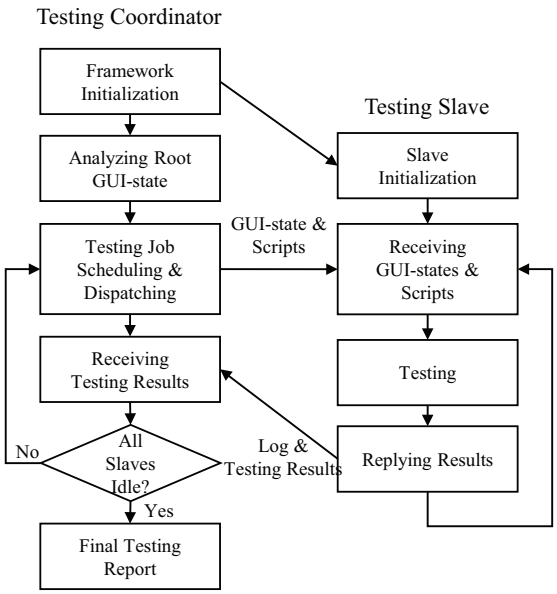

Fig. 6. The operation flow between the coordinator and the slaves.

- **CWA_app**[2]: This is an app for a non-profit organization[3]. It has more UI-states than HandyTimetable.
- **CHT_CustService**[4]: This is an app to provide customer services of CHT (Chunghwa Telecom), a Taiwan telecom operator. It has been downloaded more than 500,000 times. This app has the most UI-states in the testing process.

In the current prototype, PATS does not support complicated UI components which are common in many popular Android apps, such as games and Web applications. One major problem of handling such complicated UI components is that the UI operations of these components are ad hoc and content-dependent. In addition, the current PATS design does not consider how to test a group of collaborative applications. A model-based testing mechanism may help to tackle these testing problems. However, it is left to our future work plan.

In the experiments, we only perform simple tests by traversing every GUI components with their all possible actions for the sake of investigating the testing efficiency. These simple tests can be used to check crash faults. Other faults, such as event bugs listed in [12], are neglected in the experiments.

Figure 7 shows the experimental results of measuring the average testing time. In the experiment, each app is tested 10 times on a standalone node and the PATS prototype. In the standalone mode, all tests are executed on a single TS machine. The testing granularity in the experiments is a single GUI-state. A greedy dispatching algorithm is used to schedule the testing jobs in the experiments.

[2]https://play.google.com/store/apps/details?id=com.eightdsapp.sf00017
[3]http://cwa-roc.org.tw/
[4]https://play.google.com/store/apps/details?id=com.cht.custservice

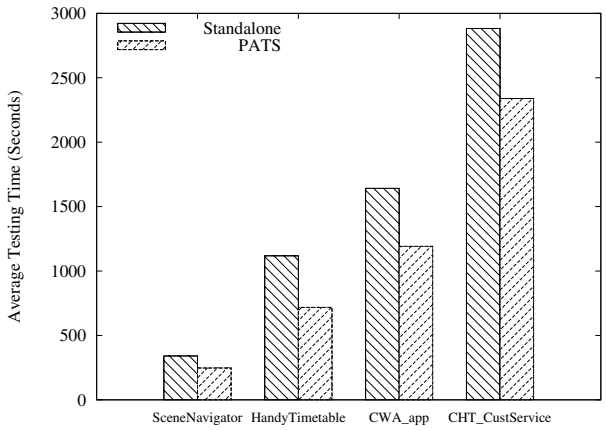

Fig. 7. Experimental results of the average testing time in seconds.

From the figure, we can notice that PATS can highly reduce the average testing time in SceneNavigator and achieve 27.35% improvement. HandyTimetable and CWA_app are two apps with 15-25 UI-states. In these two apps, PATS can achieve 35.78% and 27.41% improvements in average testing time. CHT_CustService is an app with more than 40 UI-states. For the testing experiments on this app, PATS can achieve 18.87% improvement in average testing time. For the total testing time on CHT_CustService, the standalone node takes 8 hours and 48 minutes. In contrast, PATS takes only 7 hours and 8 minutes. PAST outperforms the standalone testing mode in achieving high testing efficiency. There are two main reasons for the improvements: the GUI-trees of the applications have large degrees of parallelism; the UI-states can be evenly distributed to the TS nodes.

In the experiments, we also find a state inconsistency problem between the emulator and the state resumer in the current implementation. This inconsistency problem comes from the dismatched replaying speed with the slow reacting speed of the emulator. A synchronization mechanism for replaying operations will be discussed in the future implementation.

V. Conclusion and Future Work

In this paper, we present the design of a parallel GUI testing framework PATS. With a fine-grained testing model, PATS helps to improve the testing efficiency and mitigate the tedious testing process. We have implemented a prototype and conducted experiments with four Android applications. The preliminary results show that the PATS prototype can effectively reduce the testing time up to 35.78%. For testing a Android app with complicated GUIs or a large number of Android apps, testers can more quickly have the testing results.

Although the experimental results are preliminary, the PATS prototype demonstrates the effectiveness in reducing the testing time. However, our work is only the first step toward a parallel GUI testing framework. This framework derives many interesting issues which can be further investigated in the future implementation. For example, here are some crucial issues to the practical use of PATS. First, we have found the state inconsistency problem in the replaying process that

may influence the PATS performance significantly. A synchronization mechanism will be discussed to tackle this state inconsistency problem. Second, a proper scheduling algorithm will be also developed to decide the testing granularity and the dispatching process. The scheduling mechanism also helps the load balancing issue for slave nodes. In addition, the current prototype only considers a subset of GUI components. Other GUI components will be considered in the future implementation to enrich the functionality of PATS. Experiments on more complicated apps will be also conducted. Finally, the current PATS design does not consider the node failure issue. Node failure will interference the testing validity. This issue will be also addressed in our future work.

ACKNOWLEDGMENT

This work was supported in part by Ministry of Science and Tecnology, Taiwan under grants MOST 103-2815-C-155-023-E and MOST 103-2221-E-155-042. The authors would also like to express their sincere thanks to anonymous reviewers for their precious comments.

REFERENCES

[1] AppBrain, http://www.appbrain.com/stats/number-of-android-apps, last accessed on 12/26/2014.

[2] J. Andrus and J. Nieh, "Teaching Operating Systems using Android," in *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE '12)*, 2012, pp. 613–618.

[3] J. K. Muppala, "Teaching Embedded Software Concepts using Android," in *Proceedings of the 6th Workshop on Embedded Systems Education (WESE '11)*, 2011, pp. 32–37.

[4] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon, "GUITAR: An Innovative Tool for Automated Testing of GUI-driven Software," *Automated Software Engineering*, vol. 21, no. 1, pp. 65–105, Mar. 2014.

[5] P. A. Brooks and A. M. Memon, "Automated Gui Testing Guided by Usage Profiles," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*, 2007, pp. 333–342.

[6] B. Robinson, P. Francis, and F. Ekdahl, "A Defect-driven Process for Software Quality Improvement," in *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '08)*, 2008, pp. 333–335.

[7] "Android MonkeyRunner," http://developer.android.com/tools/help/monkeyrunner_concepts.html, last accessed on 01/05/2015.

[8] "JUnit," http://developer.android.com/tools/testing/testing_android.html, last accessed on 01/05/2015.

[9] "Robotium," https://code.google.com/p/robotium/, last accessed on 01/05/2015.

[10] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. D. Carmine, and G. Imparato, "A Toolset for GUI Testing of Android Applications," in *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM 2012)*, 2012, pp. 650–653.

[11] M. Bierma, E. Gustafson, J. Erickson, D. Fritz, and Y. R. Choe, "Andlantis: Large-scale Android Dynamic Analysis," in *Proceedings of the 3rd Workshop on Mobile Security Technologies (MoST 2014)*, 2014. [Online]. Available: http://mostconf.org/2014/papers/s3p2.pdf

[12] C. Hu and I. Neamtiu, "Automating GUI Testing for Android Applications," in *Proceedings of the 6th International Workshop on Automation of Software Test (AST '11)*, 2011, pp. 77–83.

[13] A. Jääskeläinen, M. Katara, A. Kervinen, M. Maunumaa, T. Pääkkönen, T. Takala, and H. Virtanen, "Automatic GUI Test Generation for Smartphone Applications - an Evaluation," in *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, 2009, pp. 112–122.

[14] T. Takala, M. Katara, and J. Harty, "Experiences of System-Level Model-Based GUI Testing of an Android Application," in *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation (ICST '11)*, 2011, pp. 377–386.

[15] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using GUI Ripping for Automated Testing of Android Applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*, 2012, pp. 258–261.

[16] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated Concolic Testing of Smartphone Apps," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*, 2012, pp. 59:1–59:11.

[17] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood, "Testing Android Apps Through Symbolic Execution," *SIGSOFT Software Engineering Notes*, vol. 37, no. 6, pp. 1–5, Nov. 2012.

[18] W. Choi, G. Necula, and K. Sen, "Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*, 2013, pp. 623–640.

[19] Y.-D. Lin, E. Chu, S.-C. Yu, and Y.-C. Lai, "Improving the Accuracy of Automated GUI Testing for Embedded Systems," *IEEE Software*, vol. 31, no. 1, pp. 39–45, Jan.-Feb. 2014.

[20] Y.-D. Lin, J. F. Rojas, E. T.-H. Chu, and Y.-C. Lai, "On the Accuracy, Efficiency, and Reusability of Automated Test Oracles for Android Devices," *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 957–970, Oct. 2014.

[21] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, and A. Stavrou, "A Whitebox Approach for Automated Security Testing of Android Applications on the Cloud," in *Proceedings of the 7th International Workshop on Automation of Software Test (AST '12)*, 2012, pp. 22–28.