2014 Fourth International Conference on Communication Systems and Network Technologies

Automatic Android GUI Traversal with High Coverage

Peng Wang, Bin Liang, Wei You, Jingzhe Li, Wenchang Shi
School of Information, Renmin University of China
Beijing, China
{pengwang, liangb, youwei, lijingzhe, wenchang} @ruc.edu.cn

***Abstract*—Android apps are usually rich in GUIs and users interact with the GUIs to use the functions provided by an app. To make Android apps reliable, GUI testing is an effective method. Automation and high GUI coverage is necessary in the testing for the sake of minimizing human effort and maximizing effectiveness. However, the existing work is insufficient to meet such requirements. In this paper, we identify several challenges for conducting GUI traversal on Android, such as component recognition, event injection and UI traversal. We present a tool named DroidCrawle to address the challenges for automatically exploring the GUIs of Android apps with high GUI coverage. The evaluation of DroidCrawler shows that it is efficient and effective to automatically capture the GUI tree of the target application with high GUI coverage.**

Keywords-automatic GUI Testing; high coverage; android

I. INTRODUCTION

Nowadays, people are increasingly relying on mobile devices for numerous computational needs. Among the mobile operation systems, Google's Android platform is continually dominating the mobile OS market worldwide. A recent survey showed that the Android platform contributed 68.8% of the smartphone market share in 2012, and up to 75.0% in the first quarter of 2013 [1]. A mobile device running on Android is typically equipped with a touch screen for user operations. As a result, Graphical user interfaces (GUIs) are becoming ubiquitous as means of interacting with Android apps.

To ensure the reliability of mobile apps running on Android, greater attention should be paid on GUI testing. GUI testing can be done either by manual testing or by automatic testing. Unfortunately, there are several limitations of manual testing the GUIs, such as it consumes human labor, and it is also time consuming. For such reasons, there is a strong demand for automatic GUI testing. One kind of the works is script-based. It needs the tester to write or record a testing script and then execute the testing script in a step-by-step manner [6, 8, 9, 19, 21]. While these techniques provide efficient means for automated testing, they require manual effort to write or generate the testing script. It is insufficient to provide completely automatic solutions by employing script-based method. Moreover, the coverage of GUI is another critical factor in GUI testing. In order to achieve better testing results, as many GUIs as possible should be tested. Fuzzing test approach can hardly provide an efficient way to get high test coverage. The GUI coverage in the script-based GUI testing approaches is also limited because of the testing script. In other words, we anticipate an approach that explores GUIs of the target application thoroughly.

In order to address the above challenges, we are motivated to implement a tool that automatically traverses the target application's GUIs with high GUI coverage on Android platform. The characteristic of Android OS makes it quite difficult to carry out GUI testing for Android apps. We have identified several major challenges for automatically exploring the GUIs of Android apps such as identifying GUI components for user interacting, generating user interactions to drive the GUI transitions, and implementing traversal algorithm to obtain a high GUI coverage. For addressing the mentioned difficulties, we present a tool named DroidCrawler, which runs on host PC. DroidCrawler communicates with the device to achieve the raw information of the current GUI for recognizing the GUI components. It manages an exploration procedure to obtain a thorough GUI tree for the sake of getting a high GUI testing coverage. The tool sends user events to the device to cause GUI transitions automatically which reduces human labor.

We perform an experimental evaluation using real-world apps to automatically obtain their GUI trees. The experiment results show that the tool is quite practical to get a high GUI coverage with a reasonable time overhead.

In a word, this paper makes contributions as followings:

- We identify several difficulties for performing automatic GUI testing, and leverage several techniques to automatically drive the application to execute without manual effort.
- We come up with a deliberated traversal algorithm to obtain the GUI tree of the target application with high GUI coverage. It is deliberately designed to make a thorough traversal.
- We implement DroidCrawler, a tool for automatic GUI traversal of Android apps. Our evaluation shows that the tool is effective to automatically get the GUIs with high coverage rate.

II. BACKGROUND

A. GUIs on Android

The important characteristics of GUIs include their graphical orientation, event-driven input, hierarchical structure of the interfaces, the components inside, and the properties of those components [20]. More specifically, a user will typically interact visually with an Android application through user interfaces. Through multiple GUIs, apps can provide users enhanced features and functionality. UI transitions are caused by user events as well as system

978-1-4799-3070-8/14 $31.00 © 2014 IEEE
DOI 10.1109/CSNT.2014.236

events, as Android application is event-driven. In general, Android user interfaces form in a tree structure with each UI represents a node of the tree and the user interaction indicates an edge between nodes. Inside a user interface, various UI components make up the interactive interface, such as a Button and a TextView. Different components in an interface are organized by layout configuration, which is defined in a XML file or defined programmatically. Each UI component has a set of properties, like component identity, text content, size and so forth.

B. Android Tools

1) adb

adb is short for Android Debug Bridge [3]. It is a powerful command line tool which lets one manages the emulator or device (device in short) from a PC. By using adb, one can execute the device shell commands, and manage port forwarding on device, etc. Generally speaking, adb tool is a client-server program that includes three components, adb client, adb server and adb daemon, respectively.

The adb client and the adb server runs on a PC. The adbd runs as a daemon on the device. The adb server binds to local TCP port 5037 and listens for commands sent from adb clients. The adb server also sets up connections to all running devices by scanning ports used by them. When the server finds an adb daemon, it sets up a connection to that port. After the connection, the adb server controls the communication with devices and deals with orders from several adb clients.

2) Monkey

Monkey [4] is a command-line based tool that runs on the device. The tool is primarily used in testing the stability of the target apps. It sends pseudo-random streams of user events into the system. These events include clicks, touches, or gestures, as well as several system-level events.

Besides of running random UI interaction commands, the Monkey includes a network interface that can be used through telnet to run specific individual events on the apps. However, for security reasons, Monkey can only communicate with the local port on device. The user should set up port forwarding for the sake of communicating with Monkey. Specifically, one can set up port forwarding of host port PORT1 to device port PORT2. After port forwarding, the commands sent to PORT1 will be forwarded to the PORT2, which is listened by Monkey.

3) Hierarchy Viewer

The Android SDK provides the Hierarchy Viewer tool [5] for developers to debug and optimize the user interfaces. It represents the UI's window lists. It also displays the UI's layout view hierarchy in a visual tree representation with detailed attributes information for each view node.

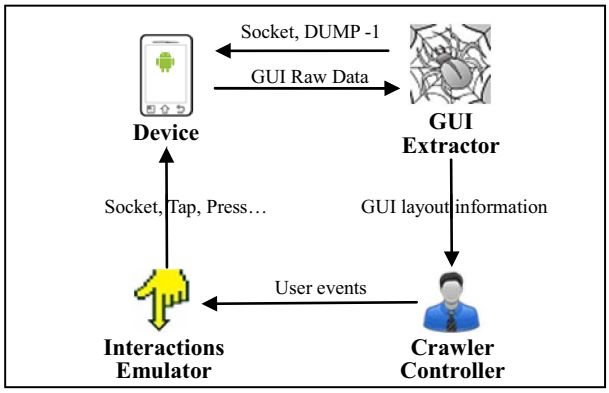

Figure 1. GUI Raw Data Format

Hierarchy Viewer can be considered as an adb client. It connects to the Window service running on the device, and communicates with the view server through port 4939 by using socket. The tool sends the server different commands to request for hierarchy view information, and displays the results. For example, when sending "DUMP -1" command to port 4939 on device, the view server will return the raw data dump of the on-screen GUI-objects such as layout hierarchy, UI components' ID, coordinates, text content and many other properties in details. The Hierarchy Viewer tool then parses the layout information and provides a visual representation to the user.

III. DESIGN AND IMPLEMENTATION

This section outlines a broad view and the implementation of our system for GUI testing. Before a further discussion of the system, we will put up several issues which should be figured out in our system:

- *I1. How does the crawler automatically identify the GUI components?* It is essential to identify the GUI under test for automatic testing. Image recognition may be a solution for component identification. It is, however, insufficient to identify all possible components because of the diversity of Android GUIs. In addition, Android framework allows the developers to custom their own GUI components, which adds the difficulty for recognition. Apart from that, as the core of Android architecture, the Android system runs each application in a separate process, and apps cannot interact with each other by default [3]. Because of these limitations, it is extremely challenge to programming identify the components as well as their properties, such as type, location, size, and whether can be triggered or not, directly from another application.
- *I2. How does the crawler generate user events to automatically explore the application?* For an automatic exploration, user interactions should also be performed automatically. Since most Android apps are GUI-based and event-driven, it is needed to simulate user interactions to make the application transit from one UI to another. To automatically

carry out the traversal process, the crawler must simulate different kinds of user interactions. Android provides various UI components and supports rich user events, like click, press and hold, swipe and so on. It is full of challenge to simulate such user events. In addition to that, in a similar line of reasoning as with the first issue, it is also hard to send user events to the target application directly from another application on device because of application sandboxes.

- *I3. How does the crawler perform the traversal procedure to get a high GUI coverage?* High GUI coverage is required in an efficient testing. We aim at traversing as many GUIs as possible. The constitution of an Android application is actually a GUI Tree, the nodes of which represent the UIs, while edges describe user interactions between them. A crawling algorithm is essential in order to conduct a thorough traversal. Despite of that, the crawling algorithm should take different kinds of GUI components into account, and inject the corresponding user events. For example, Tap event for a Button, and Type event for an EditText. The crawler should be able to recognize the state of a given GUI to avoid redundant exploration. As an example, given a GUI, a condition may tell whether the exploration of it must be continued or stopped.

To address these issues, we developed a GUI crawl tool to automatically explore the GUIs of the target application. The components composition of the tool is illustrated in Figure 1.The crawler runs on a host PC. The target application is first launched on a device or emulator. A GUI Extractor sets up the socket connection with the device via adb tool to request UI data and then retrieves the detailed attributes of each UI component, addressing I1. The traversal process is regulated by the Crawler Controller that generates user events depend on the type of UI components, and follow a deep-first traversal algorithm to determine the succession of user events to a thorough GUI traversal. This component addresses I3. In the last place, to address I2, there is an Interactions Emulator which is used to automatically inject user events to the target UI component to perform corresponding user interactions.

A. GUI Extraction

GUI components extraction is a critical aspect of this tool. This is the process to recognize the components on a GUI. Any component is characterized by a fixed set of properties. We aim at getting the components' types, such as a Button or a ListView, their absolute coordinates and their IDs in the target GUI, just as a human identify what components are in a GUI and where their positions are.

A socket connection is established between the GUI Extractor on PC and the Window Service on device. After sending "DUMP -1" command from GUI Extractor to Window Service, the Window Service returns the raw data of current GUI. In the raw data, each user interface consists of a group of view components, and each view component's detailed attributes are represented in a line. The relationship among views is organized in a hierarchy structure, which is reflected by the number of indents at the beginning of each line. The format of the obtained GUI raw data is illustrated in Figure 2.

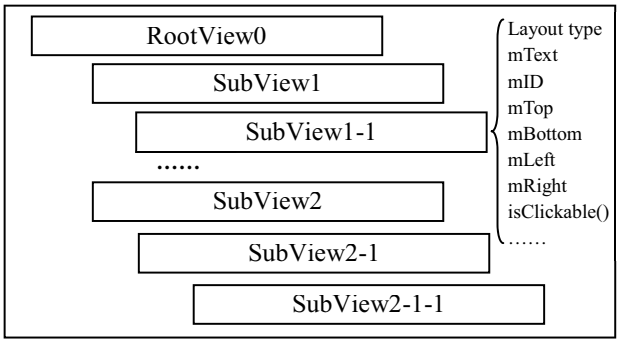

Figure 2. GUI Raw Data Format

From the raw data, we can read the various properties of all the view components in a GUI. However, it is quite obscure and inconvenient to use the raw data directly. The GUI Extractor handles the raw data and extracts the attributes of each view component, such as view type, ID, absolute coordinates, width, height, text content, whether can be clicked or not, and so on, into a particular data structure.

One important attribute of a view component is its absolute coordinate because it is essential to know the location for interacting with a view component. The absolute coordinates are measured from the top left corner of the device screen. It is worth noting that there are three aspects that should be taken into consideration when calculating the view's absolute coordinates. Firstly, the view's coordinates obtained from the raw data are relative coordinates to its parent view. The GUI Extractor needs to calculate the parent view's absolute coordinates start from the root view. Secondly, in some conditions, the GUI view does not fully occupy the screen and is shown in a proportional size. As a consequent, we should take the target GUI's dimensional proportion into account when conducting the calculation. At last, Android apps usually are not full screen, leaving the system status bar to be shown. The height of the status bar should be added for precise positions.

After GUI components extraction, we will get a component list of the target GUI with every component's attributes. This information will be used to generate user interactions and set up traversal procedure.

B. User Interactions Emulation

The running of Android apps is generally event-driven. In this paper we only consider user events triggered by users through UI interaction, despite of the fact that there are also system events that are triggered by other input sources, such as sensor receivers etc. GUI transition is driven by user events. There are two types of user event in Android, key event and motion event, respectively [8, 9]. Key event represents the event generated by pressing physical key, such as the HOME key. Motion event is generated when interacting with UI component, like button, image view.

Algorithm 1: The Traversal Algorithm

```
while (true) do
    send "DUMP -1" and get the results;
    if UI transition happen
        HierarchyList = hierarchy_new;
    else  HierarchyList = hierarchy _old;
    for each component in HierarchyList do
        if isVisible && isFireable && isNotFired
            send user event and record;
            set a isFired flag to component;
            break for;
        end if
        if current HierarchyList is handled over
            if all HierarchyLists are handled over
                break while;
            set a isOver flag to HierarchyList;
            execute Rollback algorithm
        end if
        continue while;
    end for
end while
```

Algorithm 2: The Rollback Algorithm

```
stop the application and restart;
figure out the last unfinished UI as target UI
for each event in Event List do
    re-exectue events;
    if reach target UI
        break for;
end for
```

Motion events describe movements in terms of an action code and a set of axis values. In our approach, an Interactions Emulator is used to send user events to the device for driving the application from one UI to another. It leverages the network interface of Monkeytool and takes advantage of the interaction commands supported by Monkey. The Interactions Emulator use tap, type, touch, swipe, key, press, wake and sleep during the crawl process by now. We extend the swipe command by wrapping a sequence of touch commands, for it is a quite common user interaction in an Android application. It is indeed that more complex gestures such as multi-touch gestures zoom-and pinch, are available in Android apps. We cannot simulate such user events by now. [9] can be regarded as a good solution to reply the low-level event stream.

C. Crawl Process Management

In this section, we will introduce the crawl strategy of the Crawler Controller. We consider the UI structure of Android application as a GUI tree, the nodes of which represent the GUIs, while edges describe user interaction between them. The crawl process can be taken as the process to build such a GUI tree. The Crawler Controller proposes a depth-first algorithm for constructing the GUI tree, which depends on the results of GUI Extractor and conducts the traversing procedure by exploiting Interactions Emulator. During the execution of the application, user events are generated based on the automatic dynamic analysis of the current GUI. The algorithm is revealed in Algorithm1.

In the traversal algorithm, we keep two lists, namely Hierarchy List and Event List. The Hierarchy list keeps the descriptions of GUIs, including its layout hierarchy and its UI components. We mark every different GUI with an identity. The Event List holds the events performed for each fireable component. The event is actually represented as a triples structure {FROM; EVENT; TO}, with FROM and TO indicate the identity of the GUI before and after conducting user event EVENT. Take triples {3; tap 100, 200; 4} for instance, this triple means a Tap event at the position (100, 200) causes the GUI transition from identity 3 to identity 4. To avoid redundant crawling, it is essential to make a GUI equivalence judgment. We adopt a similar approach as [8].

To get a high GUI coverage, several novelty design of the traversal algorithm will be discussed in the following paragraph. The Crawler Controller generates a corresponding user event to trigger the component based on the component's type, such as a tap corX, corY event for a Button. Currently, the Crawler Controller can tell standard Android GUI components. Before Android version 4.0, secondary menus or hidden options can't be obtained directly from the current GUI. The Crawler Controller simulates a Key MENU event on every GUI to reveal the hidden menus. Situations are common that the keyboard will be popped up automatically for the user's convenience when there is an EditText. This, however, will bring interference to our GUI recognition. We simulate a sequence of user events, first a Tap event and then a Key BACK event, to eliminate the impact of keyboard popup.

It's important to point out that we proposed a Rollback Algorithm. This algorithm is used to get to the next target GUI when the crawling of current GUI's Hierarchy List is completed. We define the next target GUI as the last unfinished GUI in the GUI tree build with depth-first algorithm. The Crawler Controller first stops the target application and then restarts it, then re-executes the events in the Event List until it reaches the next target GUI.

IV. EVALUATION

The evaluation were conducted on an emulator running Android OS v2.3.3[1] and a host PC running Ubuntu 10.10 with Intel Core 2 Duo CPU, 2.80GHz and 2G memory.

We tested several real-world apps with a clear GUI hierarchy. We recorded the testing time until the test was terminated. When calculating the GUI coverage, we manually examined the manifest file of the target apps to figure out the total number of user interfaces. Table 1 describes the evaluation result of DroidCrawler. The test time is mainly determined by the complexity of the GUIs.

[1] Note that it is not limited to the emulator and the specific version.

TABLE I. GUI TESTING RESULTS

App	Testing Time	GUIs obtained	GUI Coverage
SMSVoice	2'05'	6/6	100%
SMS Assistant	2'36''	6/8	66.7%
VoiceBox	3'12''	6/6	100%

Two GUIs of SMS Assistant were not explored because it required complexity user gestures, which will be further discussed in Section 5. We can tell that the tool achieves a quite high GUI coverage. The time consumption is reasonable when testing in a large scale.

Case Study

We present SMSVoice as a case study [22]. It provides the function of sending SMS via speech recognition. For a deep understanding of the performance of our system, we manual analyzed the app to find out that it defined 8 Activities in the manifest file. Three of them are not in our consideration: one is a splash interface, one is an ad window and one is invalid. Other three of them are triggered by receiving SMS, which is beyond our implementation as it is caused by system events. We should point it out that the number of Activities alone cannot fully indicate the number of GUIs. Dialogs can also provide UIs. After analyzing the application, there are 10 potential dialogs to be displayed. DroidCrawler is limited in crawling 6 of them, since 4 of them are related with SMS and 2 of them are shown for error warning.

That is to say there are 2 Activities and 4 Dialogs remaining which are valid for DroidCrawler. DroidCrawler traversed the app and obtained all the rest GUIs. The crawling process lasted 2 minute 5 seconds. GUI raw data extraction contributes most of the time. During crawling, 21 events were injected and 6 GUIs were obtained. In this case study, we can tell that, despite of the fact that some functions require voice service and system event which are not supported in our testing environment, the traverse result is a complete GUI tree under existing conditions. The result indicates that DroidCrawler is quite effective in traversing the app's GUIs to get a high GUI coverage. The time consumption is acceptable in a large scale when compared with manually writing testing scripts.

V. DISCUSSIONS

Limitations Though it can be employed for usage on vast apps with no manual operations, there are some limitations of DroidCrawler. (1) For GUI Extractor, when using the "DUMP -1" command to obtain raw data, the dump process is time-consuming essentially. However, this can be improved by refining the DUMP [24]. Apart from that, the GUI contents of a WebView cannot be obtained by using DUMP command. (2) The Crawler Controller can only tell the components which are standard Android components, and we do not give support to those developer-defined components by far. (3) As to Interactions Emulator, only the user events provided by Monkey is supported. Complexity gestures such as flip and zoom-in etc are unhandled by DroidCrawler. We only consider user events and do not take system events into account currently. In fact, system events generation is generally a difficult problem for all GUI testing work. (4) Currently, DroidCrawler is designed as a tool to automatic traverse the GUI tree of the target app without human interference. It can be easily implemented as a more intelligent testing tool via making additional configurations, such as the traverse algorithm, the maximum depth of the GUI tree, the end conditions and so on

VI. RELATED WORK

Android GUI Testing The major way for GUI testing in Android is the Android Instrumentation framework [16]. With Instrumentation, testing apps can be launched in the same process as the application under test. Nonetheless, accessing to the source code of the target application is required. The Android SDK provides the MonkeyRunner tool [6], which is keywords-driven, for testing. In comparison with Monkey, the MonkeyRunner tool provides an API for writing and replaying scripts to control devices and emulators from a workstation. Robotium [7], an open source project based on JUnit, offers automatic black-box testing for Android at function, system and acceptance level. Unfortunately, they all require manually generated test cases.

A mode-based approach for Android GUI testing has been proposed by Takala et al. [13]. The technique implements a keyword-based test automation tool. We have a similar technique implementation, while DroidCrawler extends user events and focuses on exhausting GUIs. Amalfitano et al. [8] present a tool A^2T^2 that based on a crawling algorithm for crash testing and regression testing. The tool exploits the Robotium and traverses the GUIs by simulating user events and reconstructs a GUI tree model. Gomez et al. [9] proposed a novelty tool named RERAN that permit record-and-reply for the Android platform. RERAN directly captures the low-level event stream on the phone and replays both GUI events and sensor events with microsecond accuracy. However, both tools require a script when implementing the testing, either written by the tester, or generated by recording the tester's operations. In our approach, the crawling procedure is totally automatic without any scripts or manual operations. AndroidRipper [14] is based on GUI ripping for automatic exploring the target app's GUI in a structured manner. It is implemented by using the Robotium Framework and the Android Instrumentation class. Hu et al. [15] presented a framework to detect GUI bugs by automatic generation of test cases. The source code of the target apps is required and instrumentation of the Dalvik VM is needed.

Driving Application Execution Automatic GUI traversal is always used as a way to driving application to run when doing dynamic program analysis. SmartDroid [11] used a UI Interaction Simulator to automatically reveal UI-based trigger conditions in Android apps. Rastogi et al. [12] proposed AppsPlayground that automate the analysis of Smartphone apps. They make an automatic exploration to explore the application code as much as possible. AppFence [17] implemented an automated execution, which is provided by the TEMA [18, 19], to measure the side effects caused by retrofitting Android apps. However, the test system requires

testing scripts with high-level commands. AppIntent [23] presented a controlled execution which automatically triggers event inputs with the support of Android InstrumentationRunner [25]. It requires modifying the original Android apk.

VII. CONCLUSION

GUI testing is necessary in order to ensure the reliability of apps on Android which are rich in GUIs. In this paper, our goal is to conduct automatic GUI testing on Android with high GUI coverage. Since Android apps are different from traditional ones, specialized challenges are presented when carrying out GUI crawling. We provided a tool called DroidCrawler to automatic traverse the GUIs with a well-designed traversal algorithm. The evaluation shows that DroidCrawler is effective and efficient for GUI testing which can be widely used to test apps automatically on a large scale.

ACKNOWLEDGMENT

This work was supported by the National Natural Science Foundation of China (61170240 and 61070192) and the National Science and Technology Major Project of China (2012ZX01039-004).

REFERENCES

[1] IDC Worldwide Quarterly Mobile Phone Tracker. May 2013.
http://www.idc.com/getdoc.jsp?containerId=prUS24108913

[2] Kapersky Lab. Today's Mobile Threatscape: Android-Centric, Booming, Espionage-friendly.
http://www.kaspersky.com/about/news/virus/2013/Todays_Mobile_Threatscape_Android_Centric_Booming_Espionage_friendly

[3] Android Debug Bridge
http://developer.android.com/tools/help/adb.html

[4] UI/Application Exerciser Monkey
http://developer.android.com/tools/help/monkey.html

[5] Hierarchy Viewer
http://developer.android.com/tools/help/hierarchy-viewer.html

[6] Android Developers, "MonkeyRunner", June 2013.
http://developer.android.com/tools/help/monkeyrunner_concepts.html

[7] Google Code, "Robotium", June 2013
http://code.google.com/p/robotium/

[8] D. Amalfitano, A. R. Fasolino, and P. Tramontana, "A gui crawling-based technique for android mobile application testing," IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2011), pp. 252-261, Mar. 2011.

[9] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "RERAN: timing- and touch-sensitive record and replay for Android," International Conference on Software Engineering (ICSE 2013), pp. 72-81, May 2013.

[10] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These aren't the droids you're looking for: retrofitting android to protect data from imperious apps," ACM conference on Computer and communications security (CCS 2011), pp. 639-652, October 2011.

[11] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications," ACM workshop on Security and privacy in smartphones and mobile devices (SPSM 2012), pp. 93-104, October 2012.

[12] V. Rastogi, Y. Chen, and W. Enck, "AppsPlayground: automatic security analysis of smartphone applications," Data and application security and privacy, pp. 209-220, February 2013.

[13] T. Takala, M. Katara, and J. Harty, "Experiences of system-level model-based GUI testing of an Android application," Software Testing, Verification and Validation (ICST 2011), pp. 377-386, March 2011.

[14] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using GUI ripping for automated testing of Android applications," ACM International Conference on Automated Software Engineering (ASE 2012), pp. 258-261, September 2012.

[15] C. Hu, and I. Neamtiu, "Automating gui testing for android applications," International Workshop on Automation of Software Test (ASE 2011), pp. 77-83, May 2011.

[16] Android Instrumentation
http://developer.android.com/reference/android/app/Instrumentation.html

[17] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These aren't the droids you're looking for: retrofitting android to protect data from imperious applications," ACM conference on Computer and communications security (CCS 2011), pp. 639-652, October 2011.

[18] A. Jääskeläinen, "Design, implementation and use of a test model library for GUI testing of smartphone applications," Doctoral dissertation, Tampere University of Technology, Finland, 2011.

[19] Tampere University of Technology. Introduction: Model-based testing and glossary.
http://tema.cs.tut.fi/intro.html.

[20] A. M. Memon, "An event‐flow model of GUI‐based applications for testing," Software Testing, Verification and Reliability, pp. 137-157, 2007.

[21] Ying-Dar, A. V. L, "Automatic Functionality and Stability Testing Through GUI of Handheld Devices," 2011.

[22] SMS Voice Suppeco.
www.appchina.com/app/it.suppeco.smsvoice.android

[23] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "Appintent: Analyzing sensitive data transmission in android for privacy leakage detection," ACM conference on Computer and communication security (CCS 2013), 2013.

[24] Request for help to improve automated testing for the Android platform.
https://groups.google.com/forum/#!msg/android-contrib/3ls0ige_E_k/iFboWDnM5vQ

[25] Android InstrumentationRunner.
http://developer.android.com/reference/android/test/InstrumentationTestRunner.html