2013 Seventh International Conference on Software Security and Reliability Companion

Behavioral Analysis of Android Applications Using Automated Instrumentation

Mohammad Karami, Mohamed Elsabagh, Parnian Najafiborazjani, and Angelos Stavrou
Computer Science Department, George Mason University, Fairfax, VA 22030
{ mkarami, melsabag, pnajafib, astavrou}@gmu.edu

***Abstract*—Google's Android operating system has become one the most popular operating system for hand-held devices. Due to its ubiquitous use, open source nature and wide-spread popularity, it has become the target of recent mobile malware.**

In this paper, we present our efforts on effective security inspection mechanisms for identification of malicious applications for Android mobile applications. To achieve that, we developed a comprehensive software inspection framework. Moreover, to identify potential software reliability flaws and to trigger malware, we develop a transparent instrumentation system for automating user interactions with an Android application that does not require source code. Additionally, for run-time behavior analysis of an application, we monitor the *I/O* system calls generated the by application under monitoring to the underlying Linux kernel. As a case study, we present two Android malware samples found in the wild to experimentally evaluate the applicability of our proposed system for uncovering potential malicious activities.

I. Introduction

The persistent Internet connectivity of smart devices coupled with their ubiquitous use and the desire of users to try new mobile applications make it a remarkable exploitation target [1]. Malware can easily pose as innocuous, must-have applications, while they can easily cost losses ranging from losing contact information to locking of the device. For instance, mobile malware has been used to steal personal contacts [2]. Moreover, current research indicates an increasing threat of malware for mobile platforms [3] even in the presence of anti-malware checks: Android Bouncer [4] has been employed in Google Play market to identify malicious applications. However, it seems that the attackers have found ways to evade detections [5]. To make matters worse, there are other, third-party markets that are available for downloading Android applications that can easily purvey malicious applications [6], [7].

Another challenge is that not all applications reveal their malicious behavior when they are installed or even run on the device. Instead, the malicious behavior can be triggered based on different conditions. For instance, a group of malware can stay dormant until they are triggered by an event [8]. Some events are independent of user interactions with the application (i.e. existence of network, etc), yet some others are based on user input [9], [10]. However, most software testing is performed towards checking the quality of software or apps but recently security testing has gained popularity and has proved its importance [11]. However, testing mobile application is not a straight forward task due to variety of inputs and heterogeneity of the technologies [12].

Two primary methods are being employed for mobile application analysis: white-box approach and black-box approach. In black-box testing only the inputs and outputs of the application are being exercised. On the other hand, for white box approach the source code need to be analyzed. Since the source code of the malicious applications that we get from Google Play is not available we cannot analyze the internal structure of the malicious applications to figure out what they exactly do, but we can utilize the black-box testing to define their functionality. In addition, current software testing tools employ randomly selected input originally introduced by the testing tool named Fuzz [13], one of the pioneers in software testing. Fuzz generates random input to the command line to leverage the security vulnerabilities. Fuzz-testing is usually a form of black-box testing that put the software being tested under the stress of unexpected inputs and monitor the respond of the system. A sample fuzzing tool is Android's Monkey [14], which due to its random inputs does not take into account all the functionality.

In contrast, Android applications are using activity driven graphical user interface heavily. Therefore, simply running the application for some time, may leave many application's functionality dormant, they might not be enough to gather information needed to figure out if the application is a threat to the user or Android device. There are different execution paths in an application and only a small number are covered by merely starting or running the application. Since dynamic analysis checks the executing code's behavior, to provide better, if not full, coverage the testing tool has to provide Graphical User Interface (GUI) input so that more paths can be covered.

Furthermore, Event Driven Software(EDS) testing techniques are based on GUI rippers [15] or finite state machines [16], however, adaptation is needed for Android application testing. GUI testing by utilizing state machines to represent GUI is discussed in [17]. The test cases generated in this model could be used by a test automation machine. Another approach [12] uses a tool to explore the application GUI and reconstructs a GUI tree model by simulating real user events. [12] builds the GUI model automatically and therefore it is a better fit for GUI testing automation. Yet, since the test cases derivation is based on a pre-defined exploration technique, it may not fulfill some exploration requirements.

978-0-7695-5030-5/13 $26.00 © 2013 IEEE
DOI 10.1109/SERE-C.2013.35

In this paper, we attempt to automate interactions with GUI in Android applications. We do so to expose hidden and regular GUI interactions or they will start without user interference. Since it will take a long time and much endeavor to try to test the applications GUI manually, we will utilize an automated approach to study the effect of user interaction on triggering malicious behavior. However, our project does not cover the context events since they cannot be controlled at the application framework layer.

The contributions of this study are two folds: an automated, intelligent fuzz testing framework that automates interaction with the application's user interface to study different functionality of the application. In addition, we introduce dynamic analysis using system calls that are gathered throughout application execution.

II. Instrumentation System

The instrumentation system developed in this study serves a major purpose: to expose the functionality of a target application, by automatically generating events that mimic user behavior. The main objectives of our system are:

- Fully supervise, monitor and control the target application
- Automatically detect and map GUI components in the target application
- Automatically generate and inject GUI events

The architecture of our instrumentation system is shown in Figure 1.

A. GUI Extraction and Mapping

To begin with, the *instrumentation engine* (based on Android InstrumentationTestRunner) loads the target application binary and coordinates the work of the different system components. Once the target process is loaded, the currently active GUI view is retrieved and passed on to the *GUI traversal* to be examined. The GUI traversal visits each component of the view, including options menu items, in preorder. When a component is visited, the GUI traversal attempts to map it to one of the default android GUI widgets by inspecting the inheritance tree of the component which is accessible through Java reflection. When the type is inferred, a set of input events that specifically fit the component type is generated. At the moment, components of unrecognized types are monkeyed with using the Android SDK Monkey event generator. This can happen if the GUI component is a direct sub-class of the basic building block, i.e., the android.view.View class.

B. Event Preparation and Scheduling

Generated events are queued into a multi-level scheduler, indexed by the view root, e.g., the parent activity. A *global event scheduler* selects the next event to dispatch and passes it back to the instrumentation engine. Each individual scheduler as well as the global scheduler can follow a set of different policies. Specifically, each scheduler can shuffle its items either randomly or based on some generated permutation. That gives control over the overall sequence of events to execute when the target process switches between different activities. In other words, it gives the view of a global sequence of events that span multiple activities. As this is all done at runtime, our system is also able to instrument dynamically loaded views, e.g., dialog boxes and menus.

The global event scheduler dispatches one event at a time. Whenever an editing event, i.e., an event that needs input from the user, is dispatched, the instrumentation engine queries the *input fuzzer* for input data. Basic fuzzing is currently implemented. The fuzzer generates sequences of characters based on the target component input type, e.g., numerical, alphanumeric, etc, that randomly vary in length. At the moment, the bounds of the sequence length can be tuned only before running the target process.

C. Event Injection

Now that the data is ready, the engine injects the event into the UI thread of the target process. Android permits the test runner to inject events only into the target process or else a security exception is thrown. That means events cannot be injected if the target process does not have focus. This is problematic as processes may request external services or may be sent to the background, e.g., due to loading another application or pressing the home key. The documented workaround is to possess the INJECT_EVENTS permission so that we can inject events into any window. One caveat though is that INJECT_EVENTS is a "signature" level permission, which would require baking our system into the Android ROM. To overcome that, we have implemented a *Hardware (HW) event injection service* that writes directly to the raw Linux input bus. Although it needs root access, writing to the Linux input bus goes below Android and so bypasses the Android permission model and eliminates the need for using the INJECT_EVENTS permission.

The main idea behind the HW event injection service is to map Android key names to their corresponding Linux scan codes, based on the available key layouts and input device handlers. Another advantage of this approach is that passing the input data becomes independent of any custom software input device that might be loaded by the target process. Generally, all input events are passed to the HW event injection service, while simple UI-wise events, e.g., gaining focus, scrolling, etc, are injected directly into the target UI thread.

D. Logging

During instrumentation, the target process is monitored and profiled. The *logger* monitors all network communication, file access and system calls, and logs them into trace files. We utilize a combination of Linux trace and Android debug facilities. Finally, after the target process consumes the input, the instrumentation engine fetches its active view, and the whole process is repeated till the test times out or is terminated by the user.

The following section provides more in depth discussion of how we monitor the target process and analyze the collected logs.

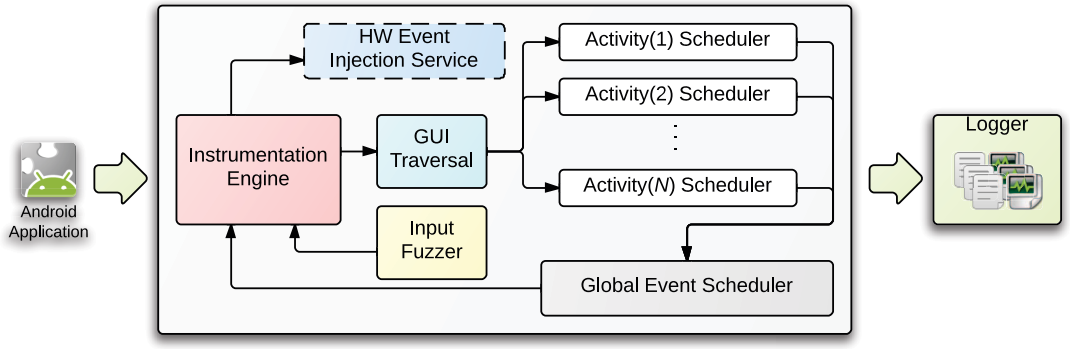

Fig. 1. Instrumentation system architecture.

III. Monitoring and Logging

In this section, we discuss in some depth how we monitor and profile a target process to characterize its run-time behavior. To this end, we first need to identify a set of events to be used as a basis for behavior characterization.

Application framework and Linux kernel constitute the two primary sources of events for behavior monitoring and profiling of Android applications [18]. Regardless of the events' source, the collected events must contain relevant information useful for revealing the potential malicious behavior of a monitored application. As the name suggests, profiling application framework events revolves around monitoring and recording application-level events such as calls made to APIs provided by the Android Software Development Kit (SDK). On the other hand, behavior monitoring and profiling at the Linux kernel-level is based on the requests made by the application under monitoring to the Linux kernel for an operating system service. System calls are the standard interface between Android applications running in Dalvik virtual machine and the services provided by the Linux kernel.

Interception of API calls is not readily achievable and platform modifications are required for gaining such capability. The Android SDK comes with a debugging framework called *logcat*. This framework can be used to partially observe the behavior of a running application. However, only limited events are dumped. For instance, we won't get any details on file access or network connections established by an application. Additionally, application developers have control over the contents to appear in *logcat's* output and therefore the logs may not be relied upon as a trusted source of information for our purpose. In short, the *logcat* framework is meant to be used as a debugging facility by application developers and it's not suitable for behavior analysis and detection of potential malicious activities.

In this study we focus on a small subset of *I/O* system calls for characterizing the run-time behavior of monitored applications. The Linux kernel constitutes the lowest level in the Android's architecture. Consequently, all user space requests for operating system services have to pass through the system call interface to get executed in the hardware. Analysis of *I/O* events as manifested in system call logs provides a reliable picture of run-time behavior of monitored applications. We rely on the *strace* tool to get visibility into system calls made by an application.

TABLE I
List of Monitored System Calls

OS Service Type	System Calls Names
File I/O	mkdir, rmdir, rename, unlink, open, read, write
Network Connection	connect, bind, listen

strace is a well-known Linux utility for monitoring system calls made by user space processes. We use a port of the *strace* utility compiled for ARM architecture to monitor a subset of *I/O* system calls.

The interception of system calls using *strace* is considered to be slow and it can incur a significant performance overhead on the execution time of the application under instrumentation. However, considering the usage of the proposed system as a testing framework, the associated overhead can be considered as negligible for the sake of improved security and reliability.

In fact, *strace* is a common tool in Android research and it has been used in recent works for malware detection [19]–[21].

One way to mitigate the performance overhead issue of *strace* is to limit the number of intercepted system calls. A modern Linux kernel has over 300 system calls and by default the *strace* tries to intercept and log all of them. However, most of these system calls are irrelevant for our purpose and do not contribute to characterization of potential malicious behavior. The operating system services related to network communications and File *I/O* operations are among the services used most heavily by different malware families [22]. Consequently, in our experiments we just focus on monitoring system calls related to these two service types. This also means no significant concern with storage requirement of generated logs. Table I lists the name of system calls that we are concerned about. In the following two subsections we provide more details on monitoring and logging of File *I/O* and network connection activities.

A. Monitoring of File I/O Activities

File *I/O* events can provide a valuable source of information for security inspection of android applications [21]. File *I/O* activities include creation, reading, writing, and deletion of file objects. Malicious applications make extensive use of file *I/O* for various malicious activities. For instance, many malware families include publicly available root exploits as file objects in their packages in either plain or obfuscated form [22]. As another example, consider a malicious application which masquerades as a legitimate app to steal users' credentials. The app might save the information as a file if a network connection is not currently available for sending back the credentials to its remote server. To avoid raising suspicion, a malicious application could delete its files once it has successfully used them for its malicious intents. To have an insight into file *I/O* activities of applications, we log relevant system calls made to the kernel. We also capture a full dump of the content for *read* and *write* system calls.

For each *read* or *write* system call, an integer file descriptor specifies the target file to be operated on. To be able to read from or write to a file , a process first needs to use the *open* system call to open the file in appropriate access mode. To have a more human readable log summary, we map the file descriptor argument of each *read/write* system call with the corresponding file object on the file-system. To do this, we record the mapping of file names passed as arguments to the *open* system call and the file descriptors assigned by the system call on a successful return. This way, we are able to identify the file associated with a file descriptor used as an argument in a subsequent *read/write* system call.

Note that, when the target of a *read* or *write* call is not a regular file on the file-system, for instance, when named pipes or Unix domain sockets are used for inter-process communication, the mapping can fail. However, If required, such file descriptors can be resolved by querying the *proc* file-system.

B. Monitoring of Network Activities

The authors of most modern malware samples are financially motivated and the monetization frequently relies on some form of communication over the Internet [23]. For instance, a spyware needs to communicate with its remote server to send back the information it has collected from the infected device or a botnet needs to contact its command and control server for downloading additional components, configuration data or operational instructions. This signifies the relevance of monitoring and profiling network activities of applications for detecting potential malicious behavior.

We monitor and log all the calls made by the applications under monitoring to *connect*, *bind*, and *listen* system calls. By inspecting the arguments of a *connect* system call, we can figure out the destination IP address and port number of a connection request. An invocation of the *listen* system call by an application indicates that the application has created a network socket and is willing to accept remote connection requests [24].

We record a summary of network connection information into a profile for each monitored application. At the same time, we run the *tcpdump* tool in background to capture and store all the network traffic in a *pcap* file. This way, if required we can use the information recorded in the network activity profile of an application to inspect the payload of its network traffic.

Typically the destination IP address and port number are sufficient for uniquely identifying the outgoing packets of an application. However, under very rare circumstances, we may have more than one application connecting to the same IP address and port number at the same time. In such case, the source port number would be necessary for distinguishing the outgoing packets of each application. The arguments of the *connect* system call as provided by the *strace* tool do not include the source port number. To address this issue, once a *connect* system call is observed in the *strace* log, the *lsof* utility is immediately invoked to identify the source port number used by the process under monitoring to establish a connection to the remote host as specified in the arguments of the *connect* system call. Please note that with this method, there is a small likelihood of missing the source port number if the established connection is extremely short-lived. This would be the case if a connection is instantaneously terminated after a *connect* system call and therefore causing an immediate invocation of *lsof* to fail to get the source port number for that connection. However, this is not a concern in our case, as in practice, a malicious connection would need to last for more than this short period of time in order to be able to accomplish its intended operation.

The source port number information of connections made by a monitored application is also recorded into the network activity profile.

IV. Experimental Evaluation

We have designed and conducted experiments to demonstrate the applicability of our methodology for detecting malicious behavior. The goal is to apply the proposed framework to identify malware samples that only expose their malicious behavior on user interaction. We evaluated our system on 20 recent malware samples collected from the wild. Not surprisingly, we found that for 17 of these samples, the malicious behavior does not depend on user interaction. The other three samples are all designed to capture and send out sensitive information entered by a user.

Here we use one malware sample from each category as use cases for the demonstration purpose. In this section, we briefly describe the functionality of the two malware samples, discuss the details of the experiment environment, and present the experimental results.

A. Malware Samples

Safe Virus Scan The first application used in our experiment is a Japanese malware called Safe Virus Scan in the native language. It was discovered and reported by the Symantec

corporation in late September 2012 [25]. The malware disguises itself as an antivirus application. However, stealing contact information stored on the infected device is the only functionality the malware is designed to perform.

Android.Fakeneflic This malware sample was also discovered and reported by the Symantec corporation in October 2011 [26]. It is basically an infostealer application targeting Netflix account information. The main component of the malware is a login screen where the account information is acquired and then uploaded to a remote server. Once the malicious application has fulfilled its purpose, it will show a dialog to the user indicating the incompatibility of the application with the current hardware. The user will have no choice but to proceed with application uninstallation.

B. Testbed Setup

We used one HTC Desire device running CyanogenMod Android 4.1 ROM for running our experiments. We equipped the device with *strace* and *tcpdump* binaries and connected it via USB to a Linux host. Our instrumentation system was deployed as a configurable test application, where only the target application package name and launcher activity need to be supplied. The Linux host was used to deploy and launch the instrumentation system and target applications, in addition to fetching the trace files generated by the logger.

Please note that both the test and target applications must be signed by the same keystore. Thus, we re-signed the malware packages with the default Android debug key before deploying them to the device.

C. Experimental Results

Some malware samples are activated based on events that are independent from user interaction while some others are only activated on user activity. The two malware samples that we have used in our experiments represent these two distinct activation conditions. For malware behavior analysis, we run each malware sample in two different scenarios. In the first run, the malware is launched and the kernel-level events generated by the application are collected as described in section III. In this scenario, the application is not instrumented. We refer to this scenario as the non-UI scenario. In the second run, we launch the instrumenting application of each malware sample to instrument the UI of the corresponding malware and simultaneously collect the same set of kernel-level logs generated by the application under instrumentation. We refer to this scenario as the UI scenario.

Based on the summary of file *I/O* and network activities that we recorded for the Safe Virus Scan malware, we observed that while the malware was pretending to be performing a genuine virus scan, all the email addresses available in the contact data were extracted, stored in a file on the SD card and the content of the file was uploaded to a back-end server at the end. At the time of this writing, the back-end server for this malware sample was still alive and therefore the scammers behind the malicious application could successfully harvest email addresses from infected devices. The logs collected for the Safe Virus Scan malware when running the application under both scenarios were qualitatively the same. This indicates that for this malware sample the UI events are irrelevant and the malicious activity will be triggered independent from user actions.

When running the experiment under the non-UI scenario for the Android.Fakeneflic malware sample, we recorded no evidence of malicious activities in our logs. However, when this malware sample was instrumented by our instrumentation system, we could detect that the malware attempted to establish a connection to a remote server when the email and password fields in the login screen were filled by the instrumentation system and then the Sign in button was clicked. The back-end server for this malware was offline at the time of this writing and therefore the malware failed to initiate the connection for posting the stolen credentials.

To uncover malicious activities we manually inspected the summary files generated under the two testing scenarios. Each summary file lists all the network connections attempted by an application and all the files being accessed on the internal memory or the external SD card. Although, the number of entries in a summary file is typically small enough for a manual inspection, this approach is obviously not scalable and the system can be improved by developing techniques for automated detection of malicious or suspicious activities.

V. Related Work

In [12] a crawler-based technique is used that stimulates real user events on GUI randomly. They modeled a GUI tree in which the nodes represent user interfaces and the edges signify transition between interfaces based on the events. Their proposed system uses the source code of the application for testing, which is not feasible in applications that are obtained from the Google Play. They experiment on one small size Android application. Our system, however, does not need to have access to the source code. Moreover, they used a testing scheme that just cover the user events produced by GUI and does not include the external events produced by sensors and network.

On the other hand, the authors in [27] has developed a blackbox, adaptive random GUI test-case generating technique that considers both user events and environmental events. Test cases consist of events sequence that are driven from randomly selected events. They used event sequence distance and Adaptive Random Testing that is not only limited to mobile application but can be used on other event-driven software. Our system does not use the distance factor and tries to generate all events that mimics user GUI input. Our system only works on android platform.

Authors in [14] have leveraged randomely generated GUI tests and used Monkey [14] platform to execute. Monkey [14] is another simple testing application that sends random event sequences, however the random testing may not be so effective in triggering of malware, that is why we did not follow the random method and tried to activate all the possible paths that are accessible by user input.

A white-box approach have been developed in [28] to analyze the program and generate test cases automatically, they used the cloud to provide scalable fuzzing. Robotium framework provided by Google Code website can be used to create powerful automatic black-box test cases for testing Android applications systems and functions .

VI. Conclusions & Future Work

We presented an automated dynamic analysis approach for security inspection of Android applications. Motivated by the observation that some malware samples are only activated on user actions, we have developed an intelligent instrumentation system to automatically interact with the UI of a target application. While an application is running, a subset of system calls are monitored and a behavioral profile is created. The profile can be investigated for uncovering potential malicious activities. To demonstrate the applicability of the proposed system, the results of applying it on two Android malware samples found in the wild were presented.

As a future work, we are exploring ways for improving our instrumentation system. These include: support for more complex interfaces such as OpenGl and gesture views and a solution for intelligently inferring the appropriate input format of input fields. Also to increase the likelihood of detecting a broader range of malicious activities we would need to consider monitoring and profiling more relevant events such as access to telephony and short messaging services.

References

[1] N. Leavitt, "Mobile phones: the next frontier for hackers?" *Computer*, vol. 38, no. 4, pp. 20 – 23, april 2005.

[2] S. Thurm and Y. I. Kane. (2010, Dec.) Apps are caught stealing private data stored on smartphones (Android and iOS). http://online.wsj.com/article/SB10001424052748704694004576020083703574602.html.

[3] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A survey of mobile malware in the wild," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, ser. SPSM '11. New York, NY, USA: ACM, 2011, pp. 3–14. [Online]. Available: http://doi.acm.org/10.1145/2046614.2046618

[4] D. Fisher, "Researchers find methods for bypassing googles bouncer android security," https://threatpost.com/en_us/blogs/researchers-find-methods-bypassing-googles-bouncer-android-security-060412, Jun. 2012.

[5] D. Halliday. (2013, Febuary) Security alert: Cleanedout. [Online]. Available: https://blog.lookout.com/blog/2013/02/07/security-alert-cleanedout

[6] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. (2012, Febuary) Hey, you, get off of my market:detecting malicious apps in official and alternative android markets. [Online]. Available: http://www.internetsociety.org/sites/default/files/07_5.pdf

[7] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Proceedings of the second ACM conference on Data and Application Security and Privacy*, ser. CODASPY '12. New York, NY, USA: ACM, 2012, pp. 317–326. [Online]. Available: http://doi.acm.org/10.1145/2133601.2133640

[8] S.-H. Seo, K. Yim, and I. You, "Mobile malware threats and defenses for homeland security," in *Multidisciplinary Research and Practice for Information Systems*, ser. Lecture Notes in Computer Science, G. Quirchmayr, J. Basl, I. You, L. Xu, and E. Weippl, Eds. Springer Berlin Heidelberg, 2012, vol. 7465, pp. 516–524. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-32498-7_39

[9] 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," in *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, ser. SPSM '12. New York, NY, USA: ACM, 2012, pp. 93–104. [Online]. Available: http://doi.acm.org/10.1145/2381934.2381950

[10] M. Szydlowski, M. Egele, C. Kruegel, and G. Vigna, "Challenges for Dynamic Analysis of iOS Applications," in *Proceedings of the Workshop on Open Research Problems in Network Security (iNetSec)*, Luzerne, Switzerland, June 2011.

[11] C. Wysopal, L. Nelson, D. D. Zovi, and E. Dustin, *The art of software security testing:identifying software security flaws*. Symantec Press, 2006.

[12] D. Amalfitano, A. R. Fasolino, and P. Tramontana, "A gui crawling-based technique for android mobile application testing," pp. 252–261, 2011. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5954416

[13] (2008, May) Fuzz. [Online]. Available: http://pages.cs.wisc.edu/~bart/fuzz/

[14] Android developers. ui application exerciser monkey. [Online]. Available: http://developer.android.com/guide/developing/tools/monkey.html

[15] A. Memon, I. Banerjee, and A. Nagarajan, "Gui ripping: Reverse engineering of graphical user interfaces for testing," in *Proceedings of the 10th Working Conference on Reverse Engineering*, ser. WCRE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 260–. [Online]. Available: http://dl.acm.org/citation.cfm?id=950792.951350

[16] A. M. Memon and Q. Xie, "Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software," *IEEE Trans. Softw. Eng.*, vol. 31, no. 10, pp. 884–896, 2005.

[17] T. Takala, M. Katara, and J. Harty, "Experiences of system-level model-based gui testing of an android application," in *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, march 2011, pp. 377 –386.

[18] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, ""andromaly": a behavioral malware detection framework for android devices," *J. Intell. Inf. Syst.*, vol. 38, no. 1, pp. 161–190, 2012.

[19] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "Profiledroid: multi-layer profiling of android applications," in *Proceedings of the 18th annual international conference on Mobile computing and networking*, ser. Mobicom '12. New York, NY, USA: ACM, 2012, pp. 137–148. [Online]. Available: http://doi.acm.org/10.1145/2348543.2348563

[20] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for android," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, ser. SPSM '11. New York, NY, USA: ACM, 2011, pp. 15–26. [Online]. Available: http://doi.acm.org/10.1145/2046614.2046619

[21] T. Isohara, K. Takemori, and A. Kubota, "Kernel-based behavior analysis for android malware detection," in *Proceedings of the 2011 Seventh International Conference on Computational Intelligence and Security*, ser. CIS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 1011–1015. [Online]. Available: http://dx.doi.org/10.1109/CIS.2011.226

[22] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 95–109. [Online]. Available: http://dx.doi.org/10.1109/SP.2012.16

[23] M. Chandramohan and H. B. K. Tan, "Detection of mobile malware in the wild," *Computer*, vol. 45, no. 9, pp. 65–71, 2012.

[24] M. Kerrisk, *The Linux programming interface: a Linux and UNIX system programming handbook*. San Francisco: No Starch Press, 2010.

[25] J. Hamada. (2012, Sep.) Fake antivirus app steals contact data on mobile devices. http://www.symantec.com/connect/blogs/fake-antivirus-app-steals-contact-data-mobile-devices.

[26] I. Asrar. (2011, Oct.) Will your next tv manual ask you to run a scan instead of adjusting the antenna? http://www.symantec.com/connect/blogs/will-your-next-tv-manual-ask-you-run-scan-instead-adjusting-antenna.

[27] Z. Liu, X. Gao, and X. Long, "Adaptive random testing of mobile application," in *Computer Engineering and Technology (ICCET), 2010 2nd International Conference on*, vol. 2, april 2010, pp. V2–297 –V2–301.

[28] 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," 2012.