# Toward Efficient Dynamic Analysis and Testing for Android Malware

Ying-Chih Shen[1], Roger Chien[1], and Shih-Hao Hung[1,2*]
[1]Academia Sinica, Taiwan
ycshen99@gmail.com
[2]National Taiwan University, Taiwan
roger.swchien@gmail.com

**Abstract**

Nowadays, Android-based mobile devices, such as smartphones and tablets, have become increasingly popular, and the number of Android applications is growing dramatically. To examine and validate such a high volume of applications, an automated testing and analysis environment is needed. Such an environment is particularly useful for the detection of malicious applications which steal the users' personal information and incur additional charges. In this paper, we present a testing and analysis framework for detecting such malicious applications. Our framework provides an automatic testing flow with minimal user interventions and is enhanced with heuristics to generate stimuli for speeding up the testing process. Compared to the built-in MonkeyRunner toolkit provided by Google, our framework delivered better efficiency in testing and detected more malicious applications with the added heuristics, according to our experimental results.

**Keywords**: Android, smartphone, information security, malware detection, automatic testing

## 1 Introduction

As the number of smart mobile devices quickly kept increasing in recent years, the *Google Android* operating system has become the most favorite target for cyberspace criminals, due to the openness of its ecosystem and the volume of its market share. Juniper Networks recently reported that the total amount of mobile malware has increased by 614 percent between March 2012 and March 2013, to a total of 276,259 malicious mobile applications [8]. It has been discovered that lots of severe security vulnerabilities may be exploited to bypass Android package digital signing mechanism [2] and can even be used to potentially infect future updates [14]. Valuable, sensitive and private contents have been leaked and have caused great damages due to such malware [7].

Unfortunately, it has been a challenging task for information technology (IT) staffs in enterprises and the operators of application markets (application stores) to determine whether an application is benign or malicious before deploying or selling the application. Since the number of new/updated applications created everyday is still increasing, an automated, efficient testing and analysis environment is needed for the community to cope with the challenge and screen malicious applications in time.

Traditional approaches can be mainly divided into two categories: static analysis and dynamic analysis. However, while many methodologies have been proposed for classifying mobile applications, many issues still exist. For example, it would take an enormous amount of time for the testing environment to generate proper stimuli (input) which make the application run through all possible execution paths. Furthermore, some malware would disguise its malicious behavior to avoid being detected by an automatic

*IT CoNvergence PRActice (INPRA)*, volume: 2, number: 3, pp. 14-23

*Corresponding Author: Dept. of Computer Science and Information Engineering, National Taiwan University, No. 1, Sec. 4, Roosevelt Rd., Taipei 10617, Taiwan, Tel : +886-2-33664888 x320, Email: hungsh@csie.ntu.edu.tw, Web: `http://www.csie.ntu.edu.tw/~hungsh/`

testing environment. This paper further discusses existing static analysis methods, dynamic analysis methods and their issues in Section 1.1, 1.2, and 1,3.

To address the issues, we proposed several techniques and integrated them into a framework to improve the efficiency of automatic dynamic testing and analysis. We present this framework and its implementation details in Section 2. To evaluate the performance of this framework, experimental results are provided and discussed in Section 3. Finally, we concludes this paper with suggestions on further research directions in Section 4.

## 1.1 Static Analysis

Static analysis utilizes the manifest of the Android applications and retrieves information such as permissions and Application Programming Interface (API) calls to detect malware. The common techniques used in static analysis methods involve but are not limited to: (1) disassemble of low-level code; (2) extraction of important properties, e.g. system calls, use of sensitive API's, access of storage devices, etc.; (3) application of pattern- or signature-based classifiers.

The implementation of static analysis methods needs to further consider a variety of issues which are specific to the Android platform. The following are recent examples:

- In SmartDroid [16], an analysis framework is developed to construct complete call graph since the linkage between two Android classes is not only via method invocation but also relies on the *intent*, especially for the view classes.
- ndroid applications which contain native codes and invoke them through the JNI interface may leads to increased complexity in call graph analysis, which was why DroidScope [15] amended this issue by extending the information flow tracking mechanism of SmartDroid from only covering the Dalvik VM layer to covering the JNI native code layer.
- While the Dalvik byte code adopted by the Android platform can be easily converted back to high-level Java source code for analysis, there are some obfuscation and evasion techniques which can be used by malware writers to prevent being analyzed [10].

Static analysis proved to be an effective way of screening malware. With a static analysis method, AndroidLeaks [3], evaluated 24,350 Android applications from several Android markets and found 57,299 potential privacy leaks in 7,414 of them. With manually verification discovered that 2,342 applications leak private data, including phone information, GPS locations, WiFi data, and audio recorded with the microphone.

The advantages of static analysis are small overhead, efficient and scalable, and the prediction methods may be based on simple, manually crafted detection patterns or learning-based models.

## 1.2 Dynamic Analysis

Unlike static analysis methods, a dynamic analysis method executes the application under test in a controlled environment, either via an emulator, sandbox or real machine with instrumentations to reveal the application's runtime behavior. Thus, dynamic analysis methods generally have an advantage over static methods since the additional data collected over the runtime can be used to improve the detection rate. Dynamically loaded library functions in Android applications can cause security issues. For example, it was found that the use of an improper loading technique exposed the application user to code injection attacks [9], and it is important for malware detection methods to examine and validate such codes during the runtime.

A complete procedure for dynamic analysis is usually divided into the following stages: (1) Instrumentation of the application under test or the testing environment with probes to collect data, (2) Execution of the application with proper stimuli, and (3) Collection and analysis of data to determine if the application is malicious.

The *MonkeyRunner* [4] utility provided by Google is capable of generating touchscreen events and limited system events in a random fashion for developers to test their Android applications. DroidBox [5], as an early implementation of Android malware detection framework, used MonkeyRunner toolkit to drive its dynamic analysis method by placing numerous API hooks in the Android system to detect potential information leaks and to generate a visualization map to classify the similar malware samples. In [11], an anomaly detection system was used for detecting meaningful derivations in the network traffic patterns generated by the applications. Semi-supervised machine learning was applied to this system to reveal two major observations: (1) Different types of applications could be categorized based on their traffic patterns. (2) Self-downloading malware or repackaged applications could also be classified with this mechanism.

Finally, it is possible to combine static and dynamic methods. *Mobile-Sandbox* [13] designed a system to automatically analyze Android applications by combing static and dynamic analysis. It was found that one of the fourth applications actually used native (non-Java) calls in their code. However, as we discuss in the following subsection, the MonkeyRunner toolkit used in Mobile-Sandbox and other works could be very inefficient in triggering malicious behavior.

### 1.3 Issues

There are two major issues which may hurt the performance of a dynamic analysis method. First, for some applications, it is difficult to generate proper stimulus to have a sufficient coverage on all possible execution paths with a reasonable amount of time. In an Android environment, such stimuli contain not only user inputs but also system broadcast events, intents, etc. Secondly, a modern malware may try to hide its malicious behavior in a stealth way in order to avoid being discovered easily. According to [17], to avoid dynamic detection, some malicious applications would use system broadcast events to hibernate and to wakeup itself.

The MonkeyRunner toolkit is quite inefficient for malware testing, since it naively and randomly generates touchscreen events and do not necessarily trigger the application under test to execute different paths. The MonkeyRunner toolkit does not pay attention to the structure and semantics of in the graphical user interface (GUI) provided by the application. In addition, the MonkeyRunner toolkit only supports a small subset of system events, which is not sufficient to expose malicious applications which may wake up on an event which is not in the subset. In [1], the structures of embedded GUI components were analyzed to trigger the transitions between different GUI views, but their work mainly focused on *clickable* objects and did not deliver the intelligence needed to handle all varieties of GUI components such as text inputs. Smartdroid [16] tackled this issue by analyzing the application in advance to guide its dynamic analysis method on the paths to reach sensitive API calls, but their efforts only focused on GUI-based events.

To expose the malicious applications which intentionally hide themselves and check if they were executed under a testing environment, a dynamic analysis method needs to be equipped with counter-measures. Multiple research reports show that some malware could detect whether it ran on an emulator and decided if it should hide its malicious behavior based on such knowledge [6] [13] [12].

## 2 Security Framework

Our detection framework, called *APE+*, is built on top of *TaintDroid*[13] and the *APE* GUI event triggering tool that we developed. APE+ is capable of real-time monitoring of privacy-sensitive operations on a smartphone by tracking the data flows inside the Dalvik VM. The framework is shown in Figure 1.

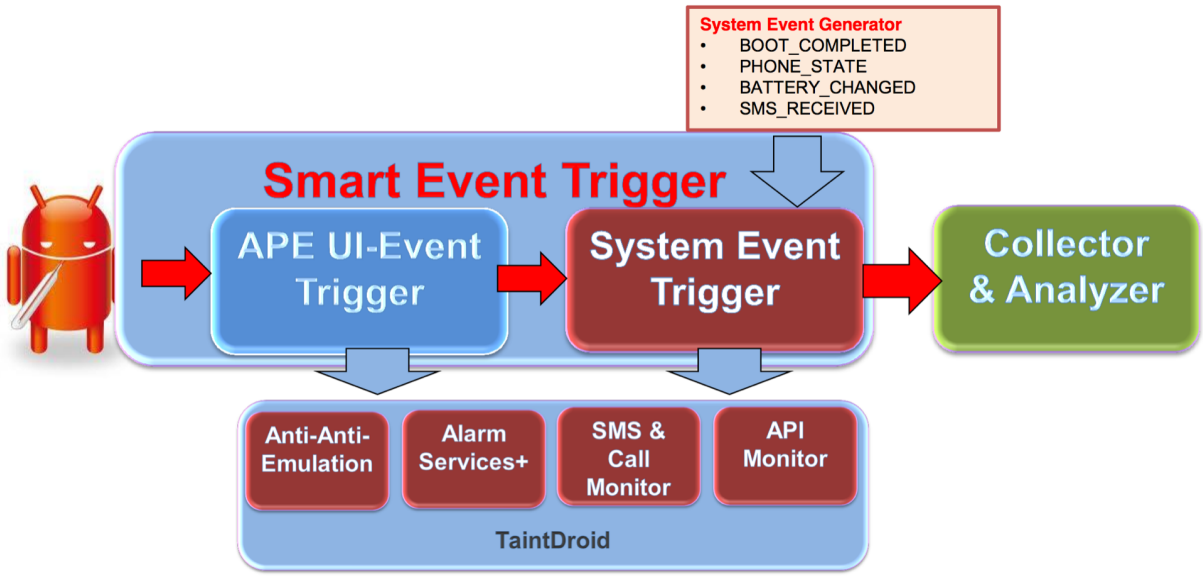


Figure 1: The APE+ framework

The *APE UI-Event Trigger* is basically a GUI components crawler but is further designed to stimulate more meaningful inputs to traverse various GUI views of the application under test with heuristics and information from examining GUI view structures in real time. With the knowledge of the application's user interface, this triggering tool is an evolution from the MonkeyRunner toolkit, which is why it is named APE. The life cycle of the view object is illustrated in Figure 2. The crawler logic is inserted in the *makeVisible()* call since all the view objects in a window are well prepared only when this call is invoked. For other GUI components which do not have the *makeVisible()* method, the proper place to insert the crawler logic is within *show()*.

The *System Event Trigger* serves to generate system events, such as battery status, connectivity status change, and SMS message receiving. The frequently triggered system events are listed partially in Figure 3. Malicious applications often register themselves as listeners for such events and start working when these events occur. For APE+ to generate different types of events on Android, the structure of an event content needs to be properly filled, and the events are triggered with a modified version of *Activity Manager*. The *APE UI-Event Trigger* and *System Event Trigger* are combined in APE+ as *Smart Event Trigger*.

Before Android version 3.1, some malware abused the *BOOT_COMPLETED* broadcast event to activate itself after the system had booted even that it had never been executed by the user. In such a scenario, malware could be installed and be activated automatically in next system boot without the consent from the user. After Android version 3.1, applications are in a stopped state by default and cannot receive any *BOOT_COMPLETED* broadcast event unless they are launched by the user at least once. Our framework is based on Android 4.1, and in order to detect more malware, we recently added the *FLAG_INCLUDE_STOPPED_PACKAGES* flag in our designed event to trigger applications in stopped state.

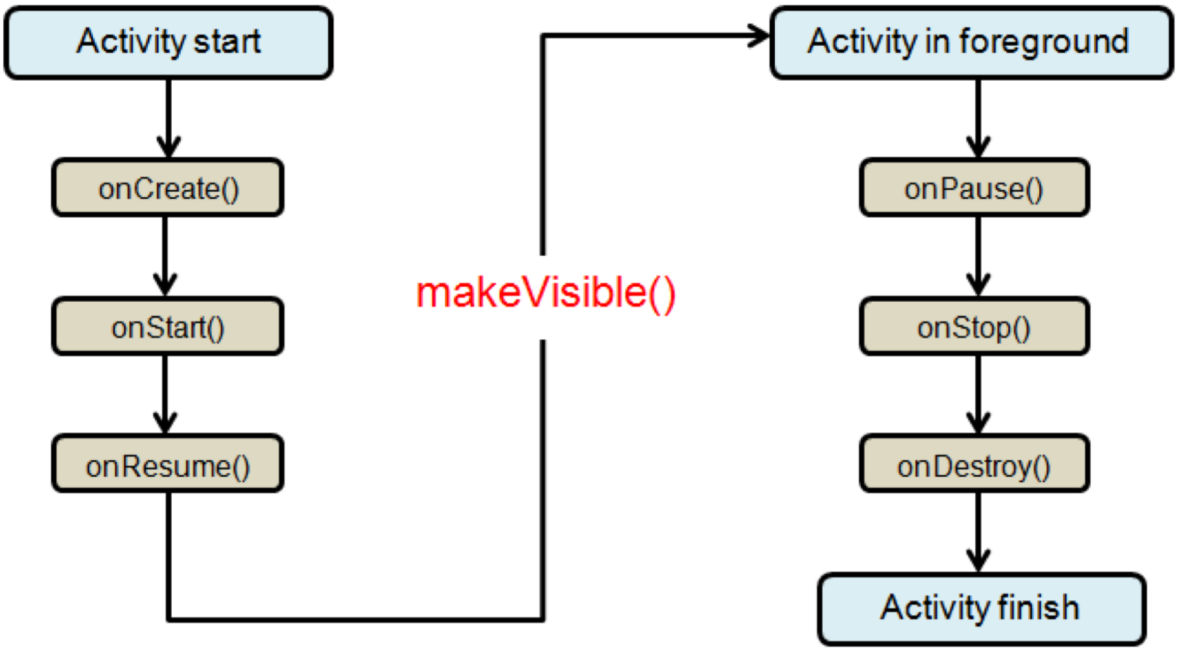


Figure 2: The life cycle of a view object

| Broadcast event actions | Description |
|---|---|
| *action.BOOT_COMPLETED* | ***System has finished booting*** |
| *Telephony.SMS_RECEIVED* | ***Received SMS message*** |
| *action.PHONE_STATE* | ***Phone state on the device has changed.*** |
| *action.NEW_OUTGOING_CALL* | ***Unlocks the screen.*** |
| *action.BATTERY_CHANGED* | ***Battery usage changed.*** |
| *CONNECTIVITY_CHANGE* | ***A change in network connectivity has occurred.*** |
| *action.USER_PRESENT* | ***Unlocks the screen.*** |
| *ACTION_POWER_CONNECTED* | ***Power got connected to the device.*** |
| *action.SIG_STR* | ***Data connection state has changed*** |
| *action.PACKAGE_ADDED* | ***A new application package has been installed*** |

Figure 3: The list of frequently triggered system events

The *Alarm Service+* is a modification of the original Android Alarm Service so that APE+ can monitor if the application under test registers itself to wake up in a later time. Once detected, APE+ will trigger the registered event and force the application of to wake up. Obviously, this allows APE+ to check if a malware intends to hide its malicious behavior from a testing environment using the alarm service. Of course, a malware writer can go further to check on the real time by referencing an external site or checking a global time service with the NTP protocol, but then the application exposes its intention by

doing so and can be caught by a static analysis method.

The *SMS/Call Monitor* is designed to check if the application under test makes a phone call or sends out an SMS message without the consent of the user. It is also extended to check if any call or SMS message is intercepted silently by malware since such kind of behavior is quite common for malware to steal the SMS token that is popularly used by Web services with two-factor authentication mechanisms. Similar to this, APE+ enables multiple security probes for sensitive operations in the emulated environment to collect detailed traces for analysis.

In addition to the aforementioned modules, APE+ has incorporates some *Anti-Anti-Emulation* techniques to avoid being detected by the malware when it intentionally checks if it is running within an emulated environment. The following modifications are applied: 1) Modification of *build.prop* to simulate specific builds of Android, 2) Modification of *PhoneSubInfo* to replace fixed values for IMSI, IMEI and phone numbers which are commonly used in the emulator, 3) Dynamic adjustment of signal strength and battery status to mimic real conditions, and 4) Installation of a *default* set of applications which are commonly found in most Android phones.

## 3 Experimental Results

In this section, we compare the performance of APE and MonkeyRunner, using 100 popular mobile applications downloaded from a third-party market. As shown in Figure 4, the left bar indicates the number applications which were found by the triggering method to have leaked any information (either with benign or malicious intention), and the right bar represents the average time (in seconds) used to test an application. In the case of MonkeyRunner, we configured the MonkeyRunner to generate 1000, 2000, and 3000 events. Obviously, with more events, it took a longer time for MonkeyRunner to finish its test for each application, but the chance to catch an information leak was increased.

The results show that APE used far less time to test an application and triggered more number of information leaks than MonkeyRunner did. Even when MonkeyRunner was configured to generate 3000 events, it could no longer increase its detection rate. We also noticed that 60% of the downloaded applications collected user's information and sent the information out to external sites, either under the user's permission or not. This represents serious risks to the user, since the user may have no idea what information has been sent.

Similar results are shown in Figure 5, where 100 malicious Android applications were used to conduct the test. The difference was even more significant since APE quickly detected 86 malware samples within approximately 20 seconds.

We also conducted another test for comparing APE and APE+ with 300 malicious applications. The results are shown in Table 1, which suggests that APE+ performed better in triggering the leaks but also needed to spend more time to handle the broadcast events. In Android, a broadcast event takes some time to complete since the event needs to be delivered to every receivers registered to receive the event.

We further break down the security risk discovered in the 226 detected malicious applications in Table 2 and outline the following observations: (1) Most malicious applications tried to collect privacy information, (2) More than a half of them steals privacy information, and (3) Some of them may lead to financial lost due to premium SMS or call.

For undetected malware samples, we also conducted manual analysis. Most of them were *root-kit exploits*, and a few of them were *remote control*, *phishing* and *SMS based bot*. The others were not identified as malware because the external sites which they tried to connect had been shut down.

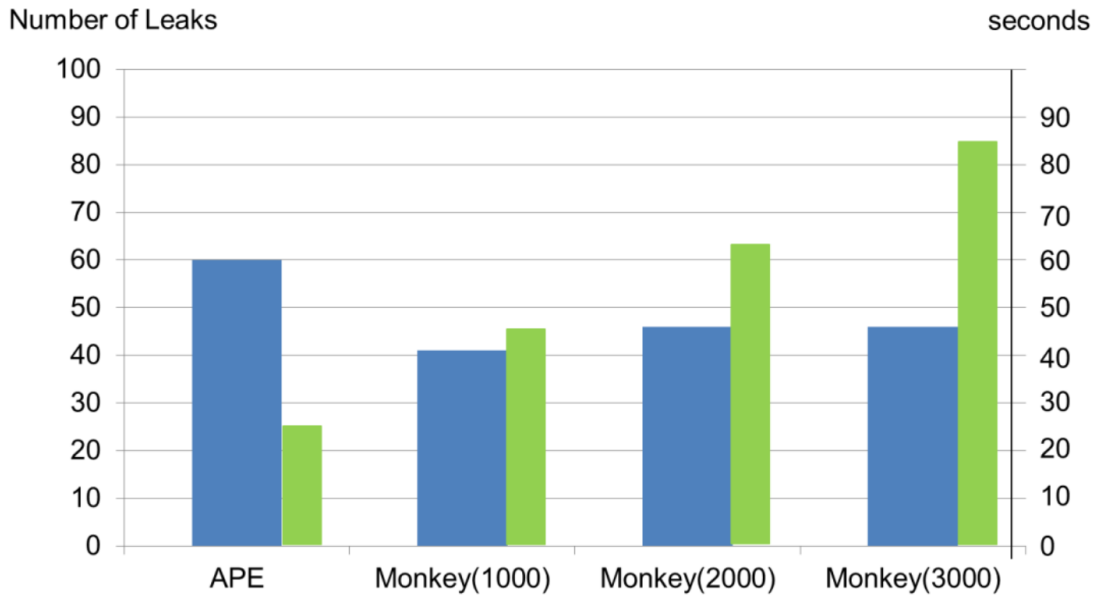


Figure 4: Performance comparison between APE and MonkeyRunner for 3rd-party applications

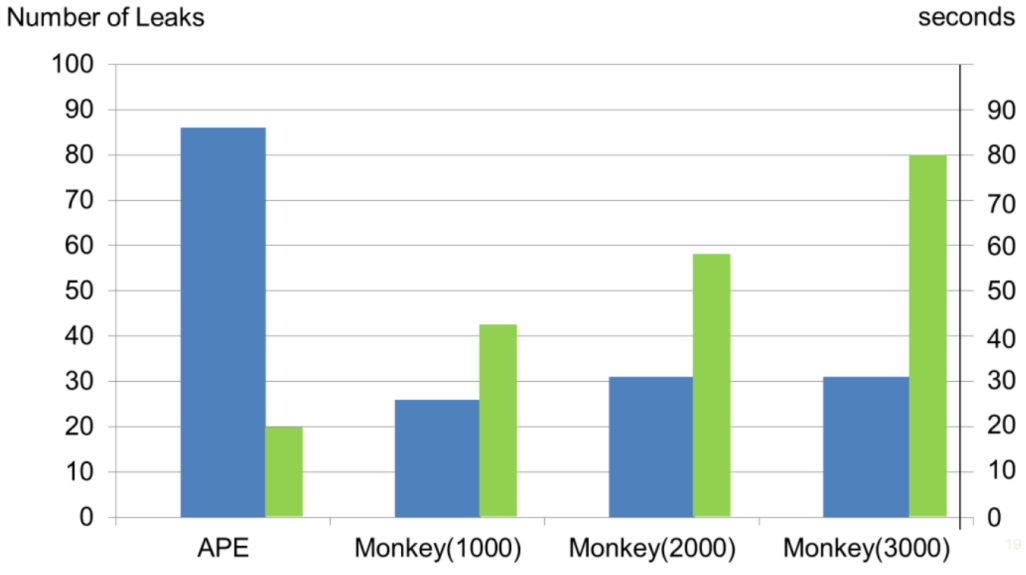


Figure 5: Performance comparison between APE and MonkeyRunner for malware

Table 1: Comparison with APE and APE+ with 300 malware applications

| | APE | APE+ |
|---|---|---|
| Number of Malware Detected(%) | 198(66%) | 226(75.33%) |
| Average Time Per Malware (seconds) | 25.7 | 58.3 |

Table 2: Detected malware behaviors

| Total Detected Malware | Information Stealing | Information Collecting | Financial Impact |
|---|---|---|---|
| 226 | 137(60.61%) | 216(95.57%) | 73(32.30%) |

# 4 Conclusion

In this paper, we have discussed analysis methods for detecting malware and addressed two major issues of black-box based dynamic testing. The efficiency and improved coverage of stimuli is important for a dynamic testing and analysis method to cope with today's high volume of malicious applications. Our proposed APE+ framework tackles these issues for automated dynamic security testing by (1) intelligently generating proper GUI events that leads to GUI view transitions, (2) injecting system-wide events to cover possible execution paths, (3) monitoring the registration of the alarm service and boosting the timer-based wakeup events, and (4) adding monitoring mechanisms for privacy related APIs. Our experimental results show improved performance with theses techniques.

In our future work, we would like to extend our framework to leverage the information gathered in static analysis stage that can have more advantages to provide best-fit stimuli and to deal with root-kit and botnet-controlled malware. We would also like to embrace big data and machine learning techniques to enhance the classification, since our APE+ framework has paved the road for data collection. Finally, we hope to reinforce our anti-anti-emulation mechanism so that APE+ may have stronger resilience against the malware which incorporate new anti-emulation capabilities.

# Acknowledgement

This work was supported in part by National Science Council in Taiwan under grants: 102-2221-E-002-087-MY3 and 103-2218-E-007-005.

# References

[1] D. Amalfitano, A. R. Fasolino, and P. Tramontana. A gui crawling-based technique for android mobile application testing. In *Proc. of 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW'11), Washington, USA*, pages 252–261. IEEE, March 2011.

[2] J. Forristal. Android: One root to own them all. BlackHat 2013 Archives, 2013. `https://media.blackhat.com/us-13/US-13-Forristal-Android-One-Root-to-Own-Them-All-Slides.pdf`.

[3] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *Proc. of the 5th International Conference on Trust and Trustworthy Computing (TRUST'12), Vienna, Austria, LNCS*, volume 7344, pages 291–307. Springer-Verlag, June 2012.

[4] Google. Ui/application exerciser monkey. Online. `http://developer.android.com/tools/help/monkey.html`.

[5] P. Lantz. An android application sandbox for dynamic analysis. Master's thesis, Electrical and Information Technology, Lund University, Sweden, 2011.

[6] J. Li, D. Gu, and Y. Luo. Android malware forensics: Reconstruction of malicious events. In *Proc. of the 2012 32nd International Conference on Distributed Computing Systems Workshops (ICDCSW'12), San Francisco, California*, pages 552–558. IEEE, June 2012.

[7] Q. Li and G. Clark. Mobile security: A look ahead. *IEEE Security & Privacy*, 11(1):78–81, Jan.–Feb 2013.

[8] J. Networks. Juniper networks third annual mobile threats report: March 2012 through march 2013. Annually online report, March 2013. `http://www.juniper.net/us/en/local/pdf/additional-resources/jnpr-2012-mobile-threats-report.pdf`.

[9] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *Proc. of the 19th Annual Network and Distributed System Security Symposium (NDSS'14), San diego, California*, pages 1–16, February 2014.

[10] V. Rastogi, Y. Chen, and X. Jiang. Droidchameleon: Evaluating android anti-malware against transformation attacks. In *Proc. of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIA CCS'13), Hangzhou, China*, pages 329–334. ACM, May 2013.

[11] A. Shabtai, L. Tenenboim-Chekina, D. Mimran, L. Rokach, B. Shapira, and Y. Elovici. Mobile malware detection through analysis of deviations in application network behavior. *Computers & Security*, 43:1–18, June 2014.

[12] M. Spreitzenbarth. *Dissecting the Droid:Forensic Analysis of Android and its malicious Applications*. PhD thesis, University of Erlangen-Nuremberg, 2013.

[13] M. Spreitzenbarth, F. Freiling, F. Echtler, T. Schreck, and J. Hoffmann. Mobile-sandbox: Having a deeper look into android applications. In *Proc. of the 28th Annual ACM Symposium on Applied Computing (SAC'13), Coimbra, Portugal*, pages 1808–1815. ACM, March 2013.

[14] L. Xing, X. Pan, R. Wang, K. Yuan, and X. Wang. Upgrading your android, elevating my malware: Privilege escalation through mobile os updating. In *Proc. of 35th IEEE Symposium on Security and Privacy (IEEES&P'14), San Jose, California*, pages 1–16. IEEE, May 2014.

[15] L. K. Yan and H. Yin. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proc. of the 21st USENIX conference on Security symposium (Security'12), Washington, USA*, pages 29–29. USENIX, August 2012.

[16] 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 *Proc. of the 2nd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM'12), Raleigh, USA*, pages 93–104. ACM, October 2012.

[17] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proc. of the 33rd IEEE Symposium on Security and Privacy (SSP'12), San Francisco, California*, pages 95–109. IEEE, May 2012.

## Author Biography

**Ying-Chih Shen** is currently a staff in the Computer Center of the Institute of Statistical Science, Academia Sinica. He is an expert in the area of information security and system management. He obtained a M.S. degree in Computer Science from National Taiwan University in 2013.

**Roger Chien** is currently a PhD student in the Department of Computer Science and Information Engineering at National Taiwan University. His research interests include high speed networking, mobile system security and network security. He has been worked in the network security industry for years, at Broadweb Corp.(2000–2003) and at Lionic Corp. (2003–2013). He received his M.S. degree in Computer Science and Information Engineering at National Tsing-Hwa University in 2002 and graduated from National Taiwan Normal University with a BS degree inInformation and Computer Education in 1997.

**Shih-Hao Hung** is currently an associate professor in the Department of Computer Science and Information Engineering at National Taiwan University. He is also a joint faculty in Research Center for Information Technology Innovation of Academia Sinica. His research interests include mobile-cloud computing, parallel processing, computer system design, and information security. He worked for Sun Microsystem Inc. (2000–2005) after completing his post doctoral work (1998–2000), Ph.D. training (1994–1998) and M.S. program (1992–1994) at the University of Michigan, Ann Arbor. He graduated from National Taiwan University with a BS degree in electrical engineering in 1989.