Hindawi Publishing Corporation
Mobile Information Systems
Volume 2015, Article ID 369489, 12 pages
http://dx.doi.org/10.1155/2015/369489

Research Article

Analyzing User Awareness of Privacy Data Leak in Mobile Applications

Youngho Kim,[1] Tae Oh,[2] and Jeongnyeo Kim[1]

[1]*Electronics and Telecommunications Research Institute (ETRI), 161 Gaejong-Dong, Yuseong-Gu, Daejeon 34129, Republic of Korea*
[2]*Rochester Institute of Technology (RIT), 1 Lomb Memorial Drive, Rochester, NY 14623-5603, USA*

Correspondence should be addressed to Youngho Kim; wtowto@etri.re.kr

Received 7 August 2015; Revised 7 November 2015; Accepted 16 November 2015

Academic Editor: Kamal Deep Singh

To overcome the resource and computing power limitation of mobile devices in Internet of Things (IoT) era, a cloud computing provides an effective platform without human intervention to build a resource-oriented security solution. However, existing malware detection methods are constrained by a vague situation of information leaks. The main goal of this paper is to measure a degree of hiding intention for the mobile application (app) to keep its leaking activity invisible to the user. For real-world application test, we target Android applications, which unleash user privacy data. With the TaintDroid-ported emulator, we make experiments about the timing distance between user events and privacy leaks. Our experiments with Android apps downloaded from the Google Play show that most of leak cases are driven by user explicit events or implicit user involvement which make the user aware of the leakage. Those findings can assist a malware detection system in reducing the rate of false positive by considering malicious intentions. From the experiment, we understand better about app's internal operations as well. As a case study, we also presents a cloud-based dynamic analysis framework to perform a traffic monitor.

1. Introduction

Malicious code in the form of computer viruses and another malware is known to wreak havoc on IoT infrastructure as well as edge devices including mobile devices. Since mobile devices are in much wider use, the devices are more likely to be exposed to malicious code and environments similar to those devices that target enterprise systems. Antivirus and antimalware tools built for enterprise systems do not transfer well with mobile devices. A part of reasons is to the limited ability of mobile devices to efficiently run antivirus tools. Mobile device limitations include power issues due to reliance on batteries, fewer CPU cycles to dedicate to running protective software, and a smaller memory footprint to run the tools. Few literatures propose an architecture to perform mobile malware analysis in the cloud. The main purpose of the architecture is to identify the malware prior to activation on the mobile device. This can preempt the malicious code and mitigate the threat before it causes any harm to the user or the device.

However, it is quite challenging to detect an application as malware when information leakage really happens. In many cases, users are willing to send their private or sensitive data to a remote server in exchange for useful service(s) such as location-aware search services. Therefore, making a decision by a seemingly data exfiltration within a certain period of time can lead to false positives in identifying a malicious information leakage. A best way of getting the original intention of the outgoing data flow would be to ask the user if the data flow would be permissible with him/her. However, getting the intention from the user is not allowed in an automated cloud system. Instead, we choose to analyze the intention of the application causing outgoing data flow in terms of its hiding efforts in preventing the user from getting aware of the application's leaking activity.

The contributions of this paper are in three folds. First, we differentiate the data by a knowledgeable user's request and data exfiltration by malware. According to a recent study [1], Android malware tends to transmit private data without a user consent. Therefore, most of the malware samples listen

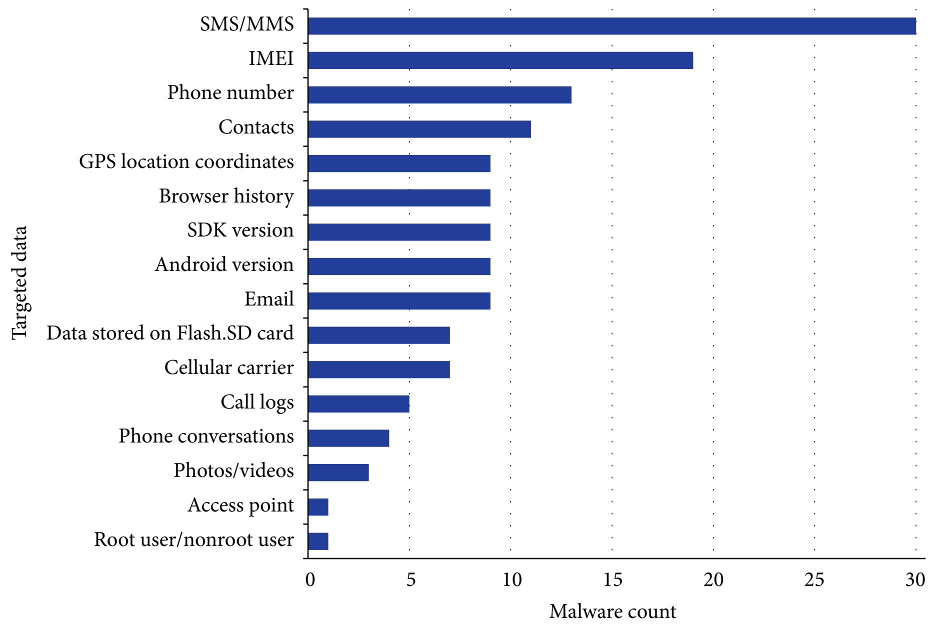


Figure 1: Private data targeted by Android malware.

to unnoticeable system events to start off their undercover services. To quantify this, the paper presents a methodology on how to measure user awareness on the data leaks. We produce this quantitative value in terms of the timing distance between a user event and data leaks activated by the event.

Second, we analyze a timing distance of real-world Android apps downloaded from the Google Play, based on the TaintDroid-ported emulator. The TaintDroid [2] traces internal data flow in order to detect a privacy leak. This instrumentation allows us to spot the leaked information and record a time stamp in milliseconds for each event. Combined with the Android's logging system, logcat [3], we can build a complete record table, featuring leaked information type, invoking interface, destination IP address, and time for the event. Our experiments show that the IMEI, accounting for 50 percent of privacy leaks, is frequently transmitted during the device identification process. In addition, the portion of IMEI leak, occurring within 30 seconds after the user's first input like a button, stands at 65.5%. The remaining cases take place during the automatic location update under a user's consciousness.

Third, to expand the concept of user awareness on the data leak, we build a cloud-based analysis system for detecting mobile malware [4, 5]. Our proposed system involves an amalgamation of both a dynamic application analysis and a network traffic analysis while the application is running. The finding that some leaked data were captured in a plain form while data are being transmitted calls for the need to secure the sensitive data with end-to-end encryption mechanism. The user awareness analysis and network-traffic monitoring can help to detect more sophisticated malware like botnet [6].

The rest of this paper is organized as follows: Section 2 gives the problem we are targeting and explanation of preliminary analysis result. Section 3 describes a design of the proposed measurement methodology and architecture. In Section 4 we present experiment results. In Section 5, we explain a cloud-based dynamic analysis as a case study. And in Section 6, we discuss challenging issues and problems. We present previous work in Section 7 and, lastly, the conclusion is shared in Section 8.

2. Background

Limited resources available to software tools deployed on mobile platforms indicate the value of a cloud-based solution. A cloud computing provides multiple instances of mobile platform with more powerful CPU and less memory constraints. In addition, the analysis and processing of potentially malicious code occur in a virtualized phone outside of the real mobile device [4, 7–9]. Aside from the limited resources, the vulnerabilities [10] of mobile devices and the hacking capabilities from the malware would make the problem more complex than enterprise computing systems.

In an effort to understand categories and behavioral characteristics of malware in obtaining the private data targeted by them, we performed a static analysis on 55 collected Android malware samples from different families by reverse engineering [11] and analyzing the codes. The analysis result of Figure 1 suggests that sensitive data including Short Message Service (hereinafter SMS) and contacts are mainly targeted by Android malware. Most malware usually has excessive access permissions to SMS and IP networking. Another aspect to note is that many malware samples imitate popular applications to deceive users into installing the malware apps without any skepticism. So we compare the permissions requested by the original application with the imitations. The chart (Figure 2) of sample applications shows three different types of characteristics. First, being the obvious one, is where

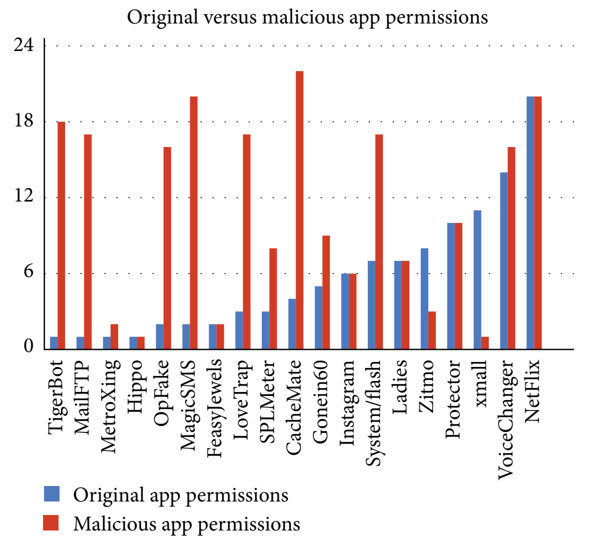


Figure 2: Permission analysis of original app versus imitation app (fake and malicious).

the malicious application requests permissions above and beyond the ones requested by the original application. For the second type, distinguishing malicious applications from the original applications based on the same number of permissions is challenging. One of the reasons is that permissions are broad in nature and encompass many features. For example, the permission called "INTERNET" allows the application complete network access. Finally the most interesting type of malicious application requires fewer number of permissions in comparison to the original application. This is the case where malware simply looks like the original application but has completely different code.

It is very challenging to draw a clear line between the malicious activity and a normal operation of mobile applications. In general, a large number of malware samples harvest the private data stored in the mobile device and send the data to remote servers with malicious intent. This form of information leak attack could have more impact on the users when financial credential is targeted. However, to make our goals more evident, this paper defines the malware as private data leaks without user's consciousness. A recent study [1, 12] shows that existing Android malware is mainly activated by BOOT, SMS, and NET related events. In particular, BOOT_COMPLETED event among system events is mostly targeted by the malware. Another interesting feature is that most malware is commanded through network and SMS messages from a remote Command and Control (hereinafter C&C) server. This trend shows a stealthy feature of malicious apps and makes them unnoticeable to the user. Also, this is a critical challenge for existing malware mitigation approaches [2, 9, 13]. Among the approaches, TaintDroid provides runtime analysis by tracking the sensitive data flow to identify the misuse of user's private information. In addition to Java-written portion of application, the DroidScope allows its internal analyzer to track the information flow of native library portion where malicious elements could reside [14].

However, automatic malware detection without human intervention tends to be erroneous due to lack of understanding of user intention. Many popular mobile applications demand user's geographic information to offer accurate locality-based services without any explicit input from the user. For example, Foursquare [15] helps people find places of interest nearby using their location. Therefore, it is really hard to tell whether the data exfiltration of the GPS location is malicious or not.

Therefore, a result of data transmission from a mobile device does not necessarily mean a data leak situation. Even benign activity could be considered as a data leakage from a simple decision based solely on information flow. As a result, a malware analysis model with rough threat definition could lead to a high false positive even in a situation when users purposely allow the internal information to leave their devices. In addition to the internal data flow itself, we should consider external contexts of applications (such as user intent) in making a decision on whether the leak activity is malicious or not.

Instead of incurring immediate information leak on the stored data, some malware aims to control the mobile device for future exfiltration. For instance, the Android.Stels [16] is one of infamous Trojans recently reported in Android platform. Once installed on the infected device, the Trojan opens a back door for handling requests from a C&C server. And later the Trojan initiates phone calls to premium numbers obtained through the C&C server. Making phone calls and sending SMS messages to premium numbers could result in a financial damage to the user. However, the Trojan never moves sensitive data out of the device or exfiltrates private user information to the remote server. Instead, this special malware activates the malicious operations without user consent. Developing a method to measure user awareness of the application's activity can assist in identifying the unintended sensitive API calls: sending SMS messages and making phone calls.

The architecture and design principles presented in this paper do not limit their applications to any specific mobile platform. The predominance of the Android platform in the mobile device market and continually expanding growth (approximately 78 percent of the mobile market in the first quarter of 2015 [17]) convince the authors to choose the Android platform as a reference model.

3. Design

3.1. Measurement Methodology. A recent user-friendly mobile application comprises multiple instances of view objects and layers. This type of application is running in an event-driven way, rather than a sequential execution flow from a program's entry point. For example, an Android app consists of several activities, representing independent execution element with corresponding user-interactive object. Each activity responds to user's inputs and displays results for visible information. For example, the Android app of Figure 3 consists of six independent elements, expressed as a set of $\{E_0, E_1, E_2, E_3, E_4, E_5\}$. Among them, the two elements, E_2 and E_5, have user-interactive operations in the blue

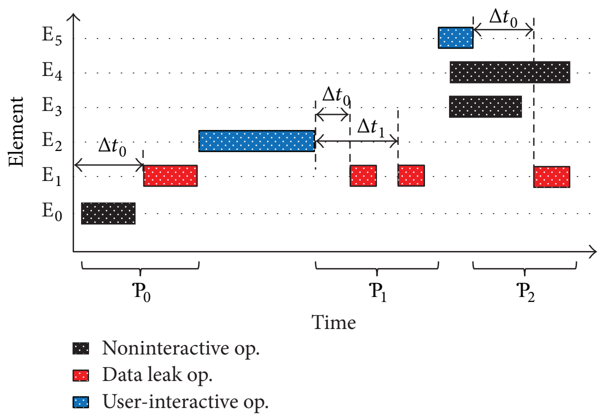


FIGURE 3: Measurement factors for data leak intention.

rectangles, responding to the user inputs. The element E_1, running as a background service without user interaction, includes data leak operations at the four distinct time slices in the red rectangles. The distance variable (Δt) designates the elapsed time between the data leak operation and its preceding user-interactive operation. The period variable, noted as $\mathcal{P}$, specifies the time span during which all corresponding data leaks are assumed to be triggered by the identical user-interactive operation. The time distances Δt_0 and Δt_1 in period $\mathcal{P}_1$ are assumed to be triggered by the same user-interactive operation.

Generally, app can be launched by clicking the app's icon. So, we assume that there is an explicit user interaction involved. Then, by our definition, the first period $\mathcal{P}_0$ begins right after the app gets started. Among all execution elements in Figure 3, the elements E_0, E_3, and E_4 in the black rectangles are running without any explicit user interaction during the app's lifetime. They do not involve any data leak either, so we exclude them from the measurement procedure. Considering the malicious applications' behavior, malware manages to hide its internal activities from the user's consciousness. In other words, the device user hardly notices the activities and events which are happening behind the user-interactive operations. As a basic unit of measurement, a time distance (Δt) of an event represents a quantitative value of a user awareness on the event. By combining all timing distances, we aim to measure a degree of intention with which the application tries to hide its internal activities from the user consciousness. As the application hides more, the distance values would be higher.

3.2. Overall Architecture. To conduct our measurement on an application in question, the runtime information inside the application should be monitored while it is running. Hence, the crucial function in our approach is to track and record all activities that take place inside the application and let the analyzer examine the gathered information afterwards. With three major design principles, we propose a platform architecture that collects the runtime information at two different checkpoints. First, tracking the data flow inside the application is an initial step to trace the flow of sensitive information. When the data under monitoring leaves the application through network or interapplication communication (IPC), this potential leak should be reported with supplemental information. To accomplish this task, application's operational information needs to be provided from the base platform on which the application is running. Second, in addition to the local data flow, external data flow such as network traffic should be taken into account. Our preliminary analysis showed that a good amount of malware communicates with an external server acting as a C&C server. Moreover, analyzing network traffic of the application allows getting a better understanding of tactics used by malware. Finally, the measurement architecture can be deployed in cloud computing environment, so the entire processes should be done automatically without human intervention.

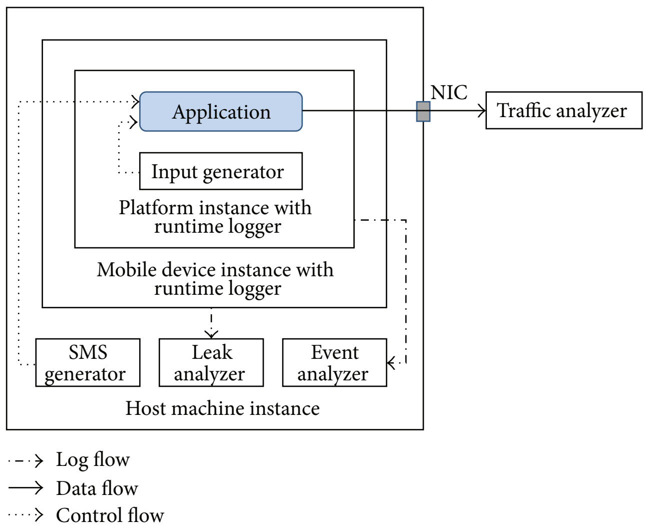


FIGURE 4: Core building blocks and operating platform.

The proposed architecture shown in Figure 4 integrates core building blocks and surrounding environment. The required functions from the design principles above are incorporated into core building blocks. And the operating platform is projected on the basis of cloud computing's virtual environment. Overall, the architecture contains three instance layers of virtual execution environment: host machine, mobile device, and software execution platform. To obtain operational information inside and outside of the application, two runtime loggers are positioned in the device and software platform instance, respectively. If we take Android as a reference model, a mobile device along with the Android software platform can be emulated on a host machine with x86 CPU. Even further, the host machine can be virtualized into multiple instances in the cloud. In order to examine an Android app, a virtualized host machine creates an instance of a mobile device with the Android platform installed. The runtime loggers instrumented in the Android platform and mobile device allows the analyzers to retrieve app's operational information. The app feels the same environment as in the real Android device so that running the app would be straightforward.

3.2.1. Analyzers. The system shown in Figure 4 has three main building blocks for analyzing application's runtime behavior while the application is running. First, the leak analyzer collects all logs of internal data traces and the outgoing data flow. Once logs are generated, the data leak analyzer inspects the leakage and then the event analyzer looks into the related event causing the data leak. Based on the gathered logs, the leak analyzer measures the time distances described in the previous section to gauge the degree of user consciousness of the data leak. For every seemingly data leak, the leak analyzer needs to locate unconscious data leak using the information provided by the event analyzer. Aside from direct data flow of the application, supplementary information such as system resources and configuration is helpful to understand application's context. Therefore, the runtime loggers located in the base platforms and mobile device are responsible for providing information to the two analyzers. Two loggers located in the instance layers of platform and mobile device generate logs which include system events, emulated user events, and their time information. In this paper, we use the logcat as a ground logging system with which the Google's Android Software Development Kit (SDK) [3] comes. Along with these internal analyzers, there is an external analyzer, traffic analyzer. When an application is transmitting data outside, the outgoing packets would be generated. Then the traffic analyzer can examine the data flow passing through the network interface card (NIC).

As for the data leak monitoring, many outstanding approaches [2, 5, 18–20] have been suggested using dynamic analysis. Generally, the dynamic analysis examines the behavior of an application in execution within a controlled environment. Among those available in public domain, TaintDroid was adopted as a runtime logger to trace data flow and identify any data leak. The TaintDroid, a type of dynamic taint tracking technique, traces local data and variables through the instrumented Dalvik virtual machine. This Android-based approach aims to identify the sensitive data leakage in real phone.

3.2.2. Generators. Interactive mobile applications are usually triggered by user inputs or events from system services. To deploy our architecture in the cloud, we build core blocks on the virtual instances. For the same reason, one of most important issues in automating the analysis process is to emulate user interaction without actual human involvement. The input generator of Figure 4 controls the application under examination by emulating user input events to trigger all activities inside. In addition to the input generator, the SMS generator makes the application believe that it runs on a real phone by emulating SMS messages. When testing an application without human intervention, the performance of the input generator determines the accuracy of the entire dynamic analysis system.

Our preliminary static analysis in the previous section showed that most malware has excessive permissions on the SMS and IP networking access. Similarly, the previous studies [1, 12] suggest that a good number of malware samples utilize SMS messages to activate themselves. Therefore, emulating SMS is critical for activating malicious applications as well as normal applications. In practice, the SMS generator is tightly coupled with a mobile device emulator. The generator used in our experiment utilizes the Google's Android Debug Bridge (hereinafter ADB). The ADB provides a command to simulate an incoming message with a bogus phone number.

The proposed input generator scans all objects on the view layout [21] of the application and traverses each view object one by one using emulated valid inputs. This process can be achieved in a depth-first search data structure which keeps track of the activities in a stack and visits them thoroughly, thereby traversing through all the views of the application. Consider an application with three different visual activities shown in Figure 5. The proposed algorithm works by generating appropriate input to its text field and an event for clicking the button. And then clicking the button gives rise to new activity, Activity 2 for button 1. If the next activity gets started, then the same procedure applies recursively to the activity to explore further. The search flow path of our depth-first search described so far is represented as a tree in solid line.

Once the recursive search routine reaches the right-most leaf node, we can get the search flow back to the previous activity via an explicit or implicit back button. Before getting started at every activity's entry, a loop-check routine is in place to see if there is a cycle in the already traversed nodes. To avoid a loop situation, the traversed activity nodes are maintained in the list and we compare them with the current node before getting started.

With the help of these two generators, any application can be tested without any knowledge about the application's source code. However, recent mobile applications are implemented with more sophisticated view objects so that the input generator needs to understand the object's detailed information to generate valid input events. This often results in considerable overhead when writing robust, automatic black box test cases. Moreover, as privacy sensitive apps tend to require user authentication, generating valid text for one-time passcode remains as a challenging problem to emulate correct user inputs.

4. Experiments

In this section, experimental data sets and results are presented. With representative real-world Android apps, user's awareness of data leaks is measured based on our methodology introduced in Section 3. For the underlying virtual instance layers of operating platform, VirtualBox [22], QEMU [23], and Dalvik virtual machine are used. The QEMU-based device emulator and the TaintDroid-ported Dalvik virtual machine have equipped the runtime logger, providing application's operational information. The host machine for our experiments is based on 64-bit Linux with 3.5 GHz 8 cores and 32 GB memory. The mobile software platform is Android 4.3 Jelly Bean which is the latest applicable version for the TaintDroid integration. There are several ways to get Android apps for experiment but downloading from Google Play, known as a formal Android application market place, is selected. There are several criteria for selecting apps to analyze the experiment. First, apps need to be installed and

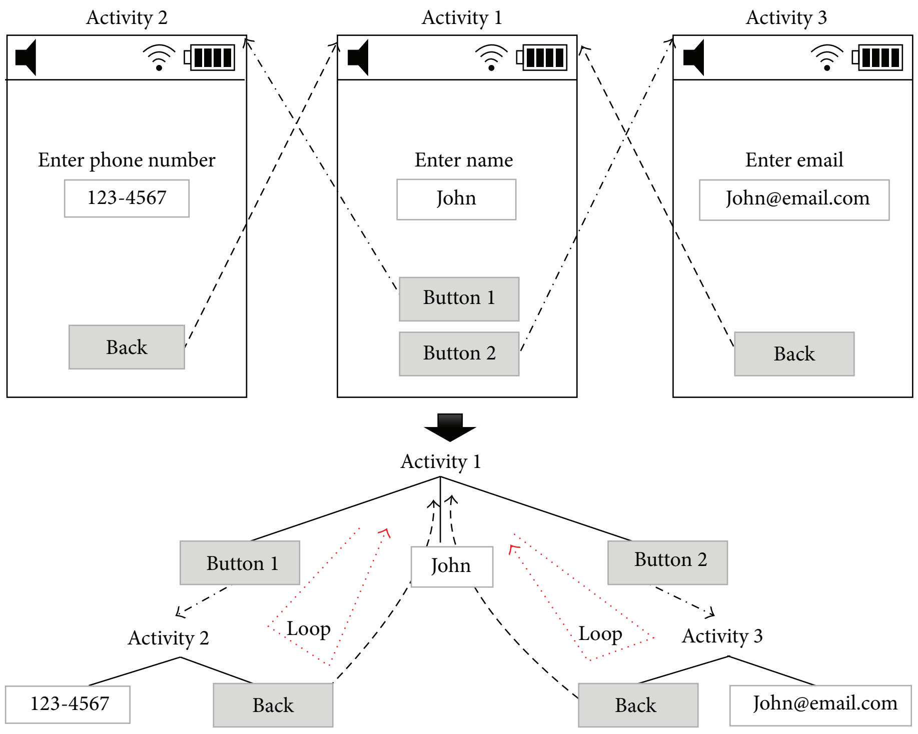


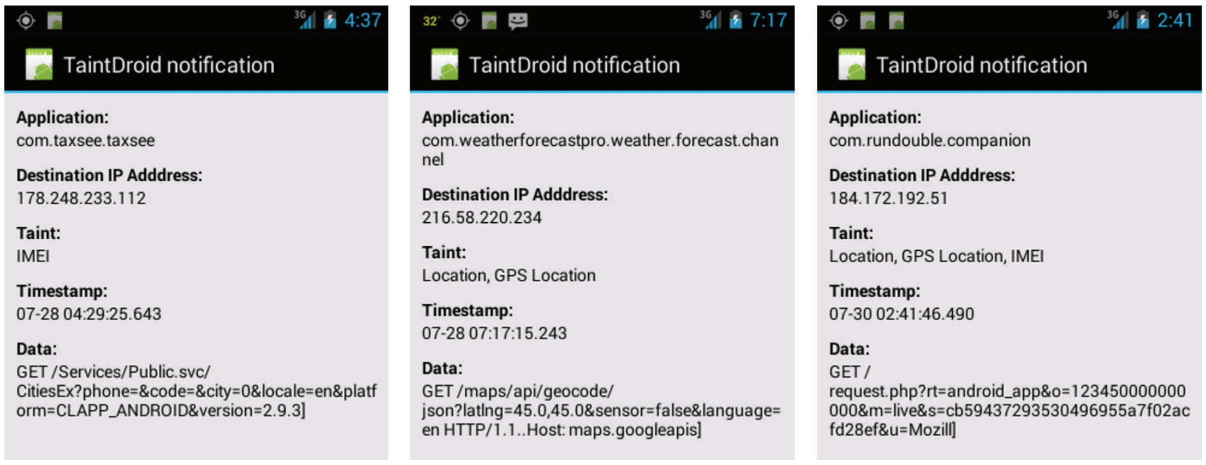
Figure 5: Depth-first search tree in user inputs emulation.

run successfully on our emulated platform. Some apps check whether a Subscriber Identification Module (SIM) is inserted and ready for use. The emulator-based software platform fails to install those apps due to the difficulty in emulating the physical SIM card. Second, to examine the relationship between data leak and user awareness, the app under test should transmit sensitive data to an external server or a group of servers. This requires the app to gain the permissions of full network access and location. Before downloading the candidate app, a full list of permissions should be checked to be granted and make sure that the app is likely causing data leaks. After the candidate meets this screening requirements, then a measurement sample is selected. Third, data set needs to represent the same type of services. We mainly focus on the personal service apps which utilize sensitive data. The coverage of the finally selected apps ranges from health, weather, and traffic to chatting messenger services.

Before starting our measurement experiment, an emulator is set up with bogus private data such as IMEI, GPS location coordinate, and contacts. This will let the app under test run as normally as in a real phone. The Google's ADB tool provides a set of commands for developers to configure the phone's system information. For example, the emulated device's geographic location can be set by the geolocation command. The power status of the phone can be manipulated to trigger the phone into a power save mode. Other than editable system information at the ADB tool, the IMEI number can be directly inserted into the platform's source code. And the contacts and photos are manually filled in and saved for every experiment. Other delicate issues related with the emulator testing environment will be addressed later in the discussion section.

Once an Android app is downloaded from the Google Play and installed, our analysis process is ready to start off. With the help of the TaintDroid notification function, a data leak can be easily spotted. The pictures of Figure 6 present some screen captures of the notifications informed by the TaintDroid. When the TaintDroid icon shows up on the status bar, the icon can be pulled down and show the detailed information about the corresponding data leak. The information includes the name of app causing the data leak, destination IP address, leaked data type, time of this event, and data itself. Among them, the data type, marked as tainted, and the timestamp are measured. Eight sample apps were tested, representing different service families. The preliminary analysis result shows that the 8 samples are sending sensitive data to remote server(s) more than one time. In practice, location-aware apps try to locate the phone in order to provide neighboring services or location-dependent information.

Given this fact, it is expected that the Glimpse and Weather Forecast Pro, location-related services, transmit the location data to the server. Also the personal workout assistant app, RunDouble, transmits the phone's IMEI, location, and GPS location multiple times during the test. The personalized apps usually transmit devices' IMEI information as well as location data to the remote server. Likewise, sending IMEI out is an essential part to personalized or device-oriented services. As with these 3 samples, other 5 apps provide their unique information such as device's IMEI in exchange for device-identifiable services.


4:37
TaintDroid notification

Application:
com.taxsee.taxsee

Destination IP Address:
178.248.233.112

Taint:
IMEI

Timestamp:
07-28 04:29:25.643

Data:
GET /Services/Public.svc/CitiesEx?phone=&code=&city=0&locale=en&platform=CLAPP_ANDROID&version=2.9.3]

7:17
TaintDroid notification

Application:
com.weatherforecastpro.weather.forecast.channel

Destination IP Adddress:
216.58.220.234

Taint:
Location, GPS Location

Timestamp:
07-28 07:17:15.243

Data:
GET /maps/api/geocode/json?latlng=45.0,45.0&sensor=false&language=en HTTP/1.1..Host: maps.googleapis]

2:41
TaintDroid notification

Application:
com.rundouble.companion

Destination IP Adddress:
184.172.192.51

Taint:
Location, GPS Location, IMEI

Timestamp:
07-30 02:41:46.490

Data:
GET /request.php?rt=android_app&o=123450000000000&m=live&s=cb59437293530496955a7f02acfd28ef&u=Mozill]

FIGURE 6: Data leak notification screenshots.

There are several findings to notice in our experiment result of Table 1. First, the case of IMEI information leaks accounts for about 50 percent of all the data leaks. It means that the counterpart service server requests the phone's ID for every service transaction. So, we expect more frequent IMEI transmissions as an app provides more various services through communications with its server. Interestingly, the traffic information app Bey2ollak, a cross platform mobile application, transmits device's IMEI 16 times to several servers as the app starts up. This can show an exemplary behavior of cross platform app, putting pieces together from crowd-source. Second, apps with periodic data updates show higher number of information transmissions. The RunDouble tracks the distance of the running course and allows the phone user to calculate the calories. This kind of app should read location data at a certain time interval while it is in active mode. This is also shown in the Glimpse app when the function of sharing location is activated. However, most privacy leaks occur at the early time of app's execution and discontinue without explicit inputs.

Among the leaked data types, the location and GPS location type seem similar but actually have a delicate difference in getting the location information. There could be several ways to find a location of the phone. Among them, the location means to get the location information through network instead of the GPS. In particular, when the user is inside the building or in downtown surrounded by high-rise buildings, known as GPS Dead Zone, the location should be resolved by network. In our experiment, the location and GPS location are both tainted whenever the location data leaks happen.

For the timing analysis of our experiment result, leaked information types which are IMEI and location were plotted. In the measurement methodology, each data leak has a corresponding input event triggering the leak. The x-axis represents a time distance from the event to the data leak, denoted as Δt in Section 3. Apps may have multiple triggering input events and corresponding data leaks and so there exists a set of periods $\{\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_n\}$ in one app. But others like the Weather Forecast Pro and My Backup have only one input

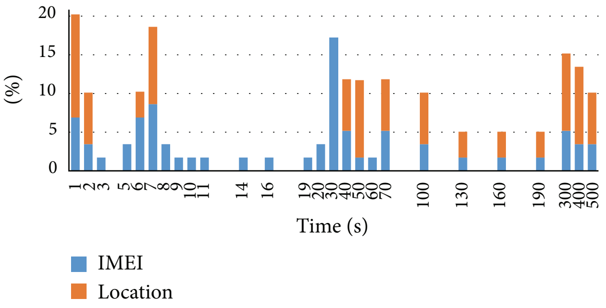


FIGURE 7: Time distance graph for all data leaks.

event, triggering actual data leaks. And then they have one period, $\mathcal{P}_0$, and a set of time distances, $\{\Delta t_0, \Delta t_1, \ldots, \Delta t_n\}$, in $\mathcal{P}_0$. The graph in Figure 7 depicts the aggregated time distances in the periods of all the apps tested.

As for the activation mechanism, clicking the app's icon is a common trigger for the data leak to happen. Therefore, many privacy leaks have been detected during the app's launch time. The next common triggering element is button object. In general, user interaction with mobile application works by clicking GUI view objects like button. Many Android apps utilize the button object to get a consent from its user. In this sense, monitoring the button object in the screen is crucial to identifying the triggering event correctly. The last device to activate the privacy leak is timeout events to get updated information from the server. As explained before, some mobile applications track location or get updated as time goes by. Such applications send out privacy data whenever they keep the local data synchronized with that of the server. And this update continues until the applications get deactivated. In all, our data set includes all privacy leaks activated by these three types of mechanism. According to the timing graph, we notice that a big group of data leaks comes within 11 seconds.

In particular, the data leak within 1 second stands at 21 percent. Considering these two facts, we assume that these apps respond to user events immediately by sending out

Table 1: Privacy leak and information type.

App name	App type	Transmitted information			API calls
		IMEI	Location	GPS location	
RunDouble	Health	●	●	●	57
Maxim	Taxi order	●			4
Trucaller	Call ID and block	●			8
Bey2ollak	Traffic	●			26
Glimpse	Location share		●	●	12
Weather Forcast Pro	Weather		●	●	6
My Bakcup	Backup	●			1
Hi	Communication	●			4
Total		**58**	**30**	**30**	**118**

necessary privacy data to the server. The next noticeable group comes around 30 seconds. Looking into this group further, most of the data leaks come mainly from the Glimpse, Bey2ollak, and RunDouble. Those apps have common characteristics, keeping the local information updated regularly. The remaining sporadic data leaks after the second group are also from those three apps. Therefore, two distinct data leak patterns, which are triggered by user's input events and recurrent timeout events, were discovered.

Getting a closer look at the first group, two separate subgroups were also discovered. The front subgroup within 3 seconds looks normal in that the data leaks in that group stem from the immediate responses from the user input events. But, the rear subgroup forming around 7 seconds needs more explanation. When an app get started or triggered into activation, it usually changes the screen for next activities. This screen change is a time-consuming job, compared to arithmetic operations or network communications, and leads to delays. Therefore, the seeming delayed date leak in the rear subgroup is attributed to the screen change and network connection latency. Overall, the time distance of data leak can depend on the application's programming design as well as activating event types.

Getting back to our design principal, we manage to correlate the privacy leak and user inputs events. That is considered to be a way to measure how much the user is aware of what the application is doing, especially leaking sensitive data. Our experiment shows that about 40% of the IMEI leaks and 33% of the location transmission happen within around 10 seconds right after user input events. All these data leaks respond to user willingness to get services by providing the personal data to external server. Meanwhile, the remaining 60% of the IMEI and 67% of the location leaks seem to take place far away from user's inputs. However, all these data leaks occur while the apps under test are running foreground with visual activities. In other words, the user assumes to be interacting with the apps without explicit user inputs. In this situation, the visual presentation shown on the screen is considered as a user's implicit intention. To be more quantitative argument, we need to devise a method to gauge the data leaks implicitly knowledgeable to the user. This will be a challenging topic and we leave it as future work.

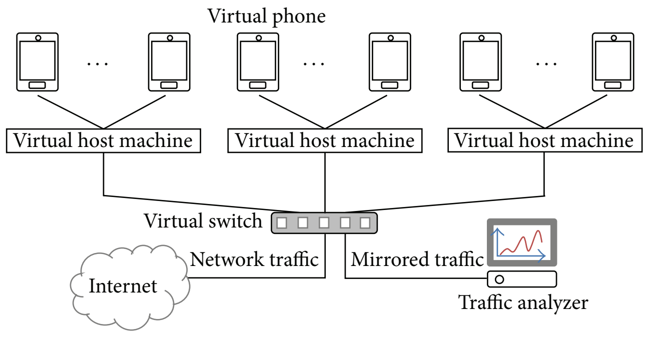


Figure 8: Cloud-based dynamic traffic analysis framework.

5. Case Study: Cloud-Based Analysis

To put the whole system into the cloud, the host machine as well as the mobile device under test should be virtualized. When an application is installed and activated by the input generator shown in Figure 4, the application will try to communicate with a remote server to send sensitive information. In that case, the traffic between the application running on the mobile device and the remote counterpart server can be analyzed with the traffic analyzer. For the network traffic analysis [24], the traffic analyzer can receive aggregated traffic from multiple instances of emulated mobile device. The overall traffic allows us to detect various traffic behavior which can only be shown in a group of mobile devices combined. For the traffic analyzer to exercise an extensive analysis over multiple devices, the traffic analyzer residing on a physical machine could be separated. With the isolated traffic analyzer, a scalable framework consisting of multiple virtual host machines is shown in Figure 8. A virtual host machine represents a host machine instance of Figure 4. The instance of host machine can connect to other virtual host machines, joining a huge virtual network which encompasses all virtual phones.

The overall framework of Figure 8 shows the traffic analyzer residing on the switch where all traffic from virtual phones merges. With this the analysis model, traffic coming from all virtual phones can be examined by a single traffic analyzer. This architecture has the following advantages against the traffic analysis running on each host machine.

(i) *Analysis on the Logical Network*. The traffic analyzer inspects the aggregated traffic pattern as well as a single stream of traffic from a device. For example, virtual devices of the same affiliation can construct a logical network. This model allows us to apply the traffic analysis to a logical user group without regard to their actual physical location.

(ii) *Efficient Policy Management*. Single check point model at gateway is efficient to deploy consistent policy over all virtual devices. This obviates the need for replicating identical policy to each host machine whenever policy changes.

(iii) *Flexible Traffic Analyzer Implementation*. No design constraints on the traffic analyzer allow using a dedicated hardware with special accelerator(s) in analyzing packets. Or we can deploy software-based traffic analyzer on general-purpose host machine.

Through the network analysis in the experiment, personally sensitive data are transmitted through the network in a plain text form. Only 16 out of 182 data leaks have been secured by the SSL encryption. IMEI and location information might be less sensitive than personal messages and contacts. And sending data through mobile data communications is relatively secure due to the closed cellular network. However, they can have a harmful impact on the user when compromised with criminal purposes. Likewise, network-level analysis helps to understand the security vulnerabilities and come up with the measures mitigating them.

6. Discussion

In this paper, the privacy data leaks transmitted through IP network are traced. However, SMS and MMS message may also be used for a mobile app to send user's sensitive data to external server. The purpose of our analysis methodology is to measure user awareness of the privacy leak and discern useful apps against malicious apps leaking personal data by stealth. While the outgoing SMS and MMS without user's explicit consent are likely to be malicious and harmful, it is obvious to make a line between normal messages transmission and anomalous behavior with bad intention. Also, our methodology can facilitate detecting malware which targets premium-rate messages. Another applicable use case is lost phone services. These applications have built-in functions for the user to make a remote control over the phone. When the user lost the phone, the GPS location information is transmitted to the management server so that the user can locate the phone. In this case, the location-related privacy data leaks take place without user's input onto phone's view object. However, we make an assumption that the phone users are already conscious about the data transmission from the lost phone. For this reason, we exclude this case from the experimental data set.

When planning our experiment in the beginning, certain types of apps were expected to cause privacy leak. Particularly, map location services are expected to yield personal data leaks frequently. However, they have rarely produced privacy leak during the test. Downloading map information before starting the navigation services explains this situation. Map and the geographic information obtained by GPS module can provide the finding-location services without incurring any data leak. As a side effect, our experiment result helps to understand mobile application's internal operation.

To test real-world applications, the emulator should be set up with experimental configuration for an application under test to feel like a real phone. Even though we put some bogus personal data into the emulator, there are still more issues to be addressed. For example, whenever Android SDK makes an instance of mobile device emulator, the MAC and IP address of the instance are allocated all the same. Malicious apps use this information to circumvent the emulated system featuring our measurement module. Fortunately, normal apps do not check their emulator information to see if they are running on the emulator or real phone.

More apps tend to require robust user authentication or device authentication instead of simply requesting email address or IMEI number. As a common user authentication method, the sever checks if the email address given by the user is effective or not by confirming reply email from the user. For device authentication, pass code validation is used in several apps. This strict authentication process is the most challenging issue for our automatic analysis approach.

7. Related Work

With the increase in sales of smartphones, there also has been a steep rise in the number of malicious applications entering the online market. Given the enormous growth of the malware, security researchers and vendors must analyze more and more applications in a given period of time to understand the purpose of the software and to develop countermeasures. Until recently, analysis is done by using tools like decompilers and runtime debuggers. This process can be very time-consuming and error-prone depending on the skill set of the analyst. On the other hand, an automatic analysis [25] investigates the downloaded application without human intervention. Main technique that the automatic analysis applies is binary forensics including decompilation, decryption, pattern matching, static system call analysis, and control flow analysis. However, malware developers also put their efforts in finding new ways for the malware to circumvent the detection mechanism [1, 26].

As one of best well-known analysis approaches, TaintDroid [2] tracks runtime data flow through variables and intercomponent communication to detect privacy data leaks. However, this fine-grained tracking technique only covers the Dalvik's Java byte codes. So, it bears limitation in identifying the data leaks occurring in the native library written in C and C++. The Mobile-Sandbox [19] integrates the TaintDroid into a virtualized phone and extends the coverage of code into native libraries written in programming languages other than Java. However, the limitation of the taint tracking lies in the lack of understating of surrounding circumstances including the user's intent. To improve the runtime taint tracking, the VetDroid [27] analyzes permission use behavior to detect malicious information leaks. As an integrated framework model, AppsPlayground [28] combines several analysis approaches such as dynamic taint tracking, API monitoring, and kernel level monitoring to supplement individual approach. Pegasus [29] uses formal method in specifying permissions and APIs property. By enforcing a permission event graph constructed from model checking, the formal method detects malicious behavior. However, the permission misuse and abnormal API call patterns only focus on the application's internal operations, not considering the user's interaction with the application. Even when a data leak is detected, determining whether the leak is maliciousness or not requires additional information.

To make all tests happen without human intervention, the input generator should control the application under examination by generating events which trigger all activities inside. Therefore, the accuracy of the input generator determines the overall performance of an automatic analysis system. The widely used Monkey tool [30] with which Android SDK comes simulates user inputs without the knowledge about the application's source code. However, this tool generates random events automatically regardless of the application's actual view layouts. Therefore, incorrect inputs sometimes lead to a crash due to input format mismatch or invalid values. Similarly, the MonkeyRunner [31] is an automatic input generator with more valid input creation. However, this tool is made to test a device at the functional or framework level and so it tends to be more application-specific in nature. SmartDroid [32] manages to find effective user inputs that trigger sensitive behavior. This approach uses both static analysis [33] constructed from function call graph and dynamic analysis exploring the UI elements to reach the sensitive APIs. Like our approach, AppIntent [20] manages to discern user intended data leak from unintended one by providing efficient sequence of GUI interactions to result in privacy data leaks. The event-space constraints model of AppIntent reduces the search space with the similar code coverage to other approaches. Unlike the above tools, Brahmastra [34] and the TriggerMetric of [35] use static analysis to construct execution paths to invoke sensitive APIs. Rather than focusing on GUI elements, Brahmastra rewrites the application to trigger the callback functions that reach privacy-sensitive APIs. Instead of instrumentation profiling, A^3E [36] improves coverage by using a static, taint-style, dataflow analysis on the bytecode.

In addition to the data leakage analysis, traffic analysis for mobile platform has been proposed in some literatures. A hacked cellular station provides a chance of identifying traffic with the malware signature [16]. That is a fundamental concept of wired network-based intrusion detection system (NIDS). Routing all the traffic from the mobile device to VPN server allows monitoring the packets the same way as the NIDS [37]. By emulating the virtualized network environment, we can apply a network-level detection to screen all traffic coming from or to the mobile device. In addition, the virtual network constructed from individual devices can provide an overall traffic behavior from the logical network perspective. This will let the traffic analyzer identify the potential threats from the network perspective as well as a single device perspective. Similar to the dynamic analysis framework proposed in the paper, Andlantis [5] provides a good scalability in a clustered environment, being capable of processing 3000 Android applications per hour. But Andlantis uses MonkeyRunner to generate user input events.

8. Conclusion

Given the pervasiveness of mobile device in modern life, proactive prevention measures against malicious applications should be put in place along with existing security solutions. In this paper, we have presented a methodology and an architecture for measuring user awareness of sensitive data leakage, which features runtime application analysis over timing distance between the user input event and actual privacy data leak. Mobile apps may request privacy data in exchange for useful services. And this seemingly voluntary data leak leads to difficulty making a clear line over whether the intention of the data leaks is malicious or not. From our experiment on real-world Android apps, we discover that the IMEI and location information are used for device identification and location-aware services. For normal apps, most data leaks stem from user's direct input events or implicit interaction with visual presentation on the screen. Moreover, the proposed methodology helps understand the mobile application's internal operations. Combined with existing malware detection systems, we expected the user awareness measurement can assist in reducing the false positive in a delicate situation by measuring the app's malicious intent.

To overcome the limited resource and computing power of mobile device, cloud computing is a great platform upon which we can build a solution free of resource constraints. Another main contribution of the paper is to employ the network-based monitoring in mobile traffic analysis. The virtual network constructed from individual phone emulators can provide a more complete network landscape the same as in physical network. From the network-perspective analysis in our experiment, we observed the vulnerable practices of transmitting the IMEI and location information in a plain text form.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgment

This work was supported by Institute for Information & Communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (no. R-20150518-001267, Development of Operating System Security Core Technology for the Smart Lightweight IoT Devices).

References

[1] Y. Zhou and X. Jiang, "Dissecting Android malware: characterization and evolution," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, pp. 95–109, San Francisco, Calif, USA, May 2012.

[2] W. Enck, P. Gilbert, S. Han et al., "TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones," *ACM Transactions on Computer Systems*, vol. 32, no. 2, article 5, 2014.

[3] Android SDK, http://developer.android.com/sdk/index.html.

[4] J. Oberheide, K. Veeraraghavan, E. Cooke, J. Flinn, and F. Jahanian, "Virtualized in-cloud security services for mobile devices," in *Proceedings of the 1st Workshop on Virtualization in Mobile Computing*, pp. 31–35, ACM, Breckenridge, Colo, USA, June 2008.

[5] M. Bierma, E. Gustafson, J. Erickson, D. Fritz, and Y. Choe, "Andlantis: large-scale Android dynamic analysis," in *Proceedings of the 3rd Workshop on Mobile Security Technologies (MoST '14)*, San Jose, Calif, USA, May 2014.

[6] C. Xiang, F. Binxing, Y. Lihua, L. Xiaoyi, and Z. Tianning, "Andbot: towards advanced mobile botnets," in *Proceedings of the 4th USENIX Conference on Large-scale Exploits and Emergent Threats*, p. 11, Boston, Mass, USA, March 2011.

[7] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos, "Paranoid Android: versatile protection for smartphones," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC '10)*, pp. 347–356, Austin, Tex, USA, December 2010.

[8] C. Jarabek, D. Barrera, and J. Aycock, "ThinAV: truly lightweight mobile cloud-based anti-malware," in *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC '12)*, pp. 209–218, ACM, Los Angeles, Calif, USA, December 2012.

[9] P. Gilbert, B.-G. Chun, L. P. Cox, and J. Jung, "Vision: automated security validation of mobile apps at app markets," in *Proceedings of the 2nd International Workshop on Mobile Cloud Computing and Services (MCS '11)*, pp. 21–26, ACM, Bethesda, Md, USA, June-July 2011.

[10] A. Nazar, M. M. Seeger, and H. Baier, "Rooting Android—extending the ADB by an auto-connecting WiFi-accessible service," in *Information Security Technology for Applications*, P. Laud, Ed., vol. 7161 of *Lecture Notes in Computer Science*, pp. 189–204, Springer, Berlin, Germany, 2012.

[11] Androguard, http://code.google.com/p/androguard.

[12] W. Enck, P. Traynor, P. McDaniel, and T. La Porta, "Exploiting open functionality in SMS-capable cellular networks," in *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS '05)*, pp. 393–404, November 2005.

[13] T. Bläsing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, and S. Albayrak, "An Android Application Sandbox System for suspicious software detection," in *Proceedings of the 5th International Conference on Malicious and Unwanted Software (MALWARE '10)*, pp. 55–62, IEEE, Lorraine, France, October 2010.

[14] L. K. Yan and H. Yin, "DroidScope: seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis," in *Proceedings of the 21st USENIX Conference on Security Symposium*, p. 29, Bellevue, Wash, USA, August 2012.

[15] Foursquare, https://foursquare.com/.

[16] S. Davido, D. Harrison, R. Price, and S. Fretheim, "Do-it-yourself cellular intrusion detection system," LMG Security Whitepaper, 2013.

[17] IDC: Smartphone Market Share, http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

[18] 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 (SPSM '11)*, pp. 15–26, Chicago, Ill, USA, October 2011.

[19] M. Spreitzenbarth, T. Schreck, F. Echtler, D. Arp, and J. Hoffmann, "Mobile-Sandbox: combining static and dynamic analysis with machine-learning techniques," *International Journal of Information Security*, vol. 14, no. 2, pp. 141–153, 2015.

[20] 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," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS '13)*, pp. 1043–1054, Berlin, Germany, November 2013.

[21] Robotium, https://code.google.com/p/robotium.

[22] Oracle VirtualBox, https://www.virtualbox.org.

[23] QEMU, http://www.qemu.org.

[24] B.-H. Chang and C. Y. Jeong, "An efficient network attack visualization using security quad and cube," *ETRI Journal*, vol. 33, no. 5, pp. 770–779, 2011.

[25] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Computing Surveys*, vol. 44, no. 2, article 6, 2012.

[26] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: systems, languages, and applications," *ACM Transactions on Information and System Security*, vol. 15, no. 1, article 2, 2012.

[27] Y. Zhang, M. Yang, B. Xu et al., "Vetting undesirable behaviors in Android apps with permission use analysis," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS '13)*, pp. 611–622, ACM, Berlin, Germany, November 2013.

[28] V. Rastogi, Y. Chen, and W. Enck, "AppsPlayground: automatic security analysis of smartphone applications," in *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy (CODASPY '13)*, pp. 209–220, New Orleans, La, USA, February 2013.

[29] K. Chen, H. Johnson, V. D'Silva et al., "Contextual policy enforcement in android Applications with permission event graphs," in *Proceedings of the 20th Annual Network and Distributed System Security Symposium, (NDSS '13)*, San Diego, Calif, USA, February 2013.

[30] Monkey tool, http://developer.android.com/tools/help/monkey.html.

[31] Monkeyrunner, http://developer.android.com/tools/help/monkeyrunner_concepts.html.

[32] C. Zheng, S. Zhu, S. Dai et al., "Smartdroid: an automatic system for revealing UI-based trigger conditions in Android applications," in *Proceedings of the 2nd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM '12)*, pp. 93–104, ACM, October 2012.

[33] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of Android application security," in *Proceedings of the 20th USENIX Conference on Security (SEC '11)*, p. 21, San Francisco, Calif, USA, August 2011.

[34] R. Bhoraskar, S. Han, J. Jeon et al., "Brahmastra: driving apps to test the security of third-party components," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, pp. 1021–1036, San Diego, Calif, USA, August 2014.

[35] K. O. Elish, X. Shu, D. Yao, B. G. Ryder, and X. Jiang, "Profiling user-trigger dependence for Android malware detection," *Computers & Security*, vol. 49, pp. 255–273, 2015.

[36] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of Android apps," in *Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming Systems Languages & Applications*, pp. 641–660, Indianapolis, Ind, USA, October 2013.

[37] A. Parrizas and D. Adrianto, *Monitoring Network Traffic for Android Devices*, SANS Institute InfoSec Reading Room, 2013.