AppTrace: Dynamic Trace on Android Devices

Lingzhi Qiu, Zixiong Zhang, Ziyi Shen, Guozi Sun
College of Computer
Nanjing University of Posts and Telecommunications, Nanjing, China

***Abstract*—Mass vulnerabilities involved in the Android alternative applications could threaten the security of the launched device or users data. To analyze the alternative applications, generally, researchers would like to observe applications' runtime features first. Then they need to decompile the target application and read the complicated code to figure out what the application really does. Traditional dynamic analysis methodology, for instance, the TaintDroid, uses dynamic taint tracking technique to mark information at source APIs. However, TaintDroid is limited to constraint on requiring target application to run in custom sandbox that might be not compatible with all the Android versions. For solving this problem and helping analysts to have insight into the runtime behavior, this paper presents AppTrace, a novel dynamic analysis system that uses dynamic instrumentation technique to trace member methods of target application that could be deployed in any version above Android 4.0. The paper presents an evaluation of AppTrace with 8 apps from Google Play as well as 50 open source apps from F-Droid. The results show that AppTrace could trace methods of target applications successfully and notify users effectively when some sensitive APIs are invoked.**

I. Introduction

Traditional dynamic analysis methods such as TaintDroid uses dynamic taint tracking technique to mark information at source APIs to identify when applications send privacy sensitive information to network servers. TaintDroids custom sandbox environment differs from the official running environment that Google provides, hence, it is not compatible with all the Android versions [5]. This motivates us to design a stable dynamic analysis system which could be compatible with as many different Android versions as possible.

Researchers begin with dynamic analysis because application source code is usually unavailable. By dynamic analysis, Researchers can observe applications runtime behavior. The next step in studying applications is trying to decompile the application to take the smali code or Java code. Another motivation is to help reserchers having insight into Android applications runtime features and closing the distance between static analysis and dynamic analysis. The above motivates our work on the AppTrace framework. For example, AppTrace logs all Java method called by a callback method when a researcher clicks a Button. And if one of these Java methods belongs to the sensitive API set which are predefined, our system will notify the researcher. Consequently, the researcher will locate the corresponding Java code as soon as possible.

In this paper, we implement the prototype of AppTrace system and run it on various Android platforms. The result shows that the AppTrace system could execute stably on any of these platforms. Besides, we have chosen 8 Apps in Google Play and 50 open source Apps from F-Droid randomly to evaluate our system. The result indicates that AppTrace could effectively trace all Java methods and feedback sensitive APIs.

Our proposed solution in this paper makes the following contributions:

a) In contrast to traditional dynamic analysis platform such as TaintDroid which is a custom sandbox environment, we propose a dynamic analysis system that could run stably in any version of Android OS.

b) We propose a novel method to trace all Java methods called by a declared method in a class and feedback the sensitive API that we predefined before. As a result, we have succeeded in closing the gap between dynamic analysis and static analysis.

c) We implement the prototype of AppTrace system and evaluate it by several real-world applications. The results show that our system could effectively trace the Java methods.

The rest of this paper is organized as follows: Section II introduces Android background. Section III describes our system design. Section IV presents the implementation of our system. Section V evaluates our system and analyzes the results. Section VI describes related work. Section VII summarizes our conclusions and future work.

II. Background

From the perspective of developers, Android applications could be classified into two categories: system applications and alternative applications. System applications are preinstalled in the device while alternative applications are downloaded from App markets or other source by users. Android applications are usually written by Java. But sometimes, certain functions are realized by native code such as C/C++ because of execution efficiency or library reusability.

Fig. 1illustrates the launch process of an Android application from the user's click to the interface being visible. The Launcher is responsible for the application startup, which is also an application. After a user clicks the application icon, the Launcher tells Activity Manager Service who needs to create a new process of the Activity via the Android particular IPC mechanism called Binder. Zygote is the parent process of all applications and system services in Android. It is created by the first process of Linux called init. Hence when the Activity Manager Service receives the request from the Launcher it will ask the Zygote process to fork out a new Linux process where the application will launch the first Activity.

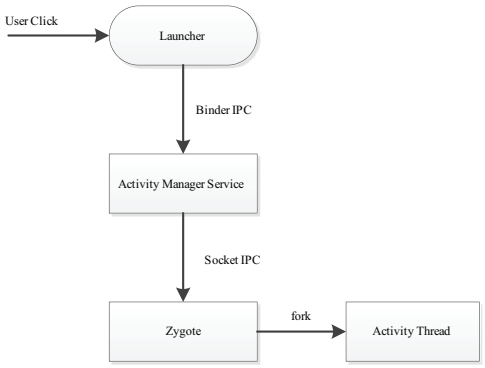

Fig. 1. The launch process of an Android App.

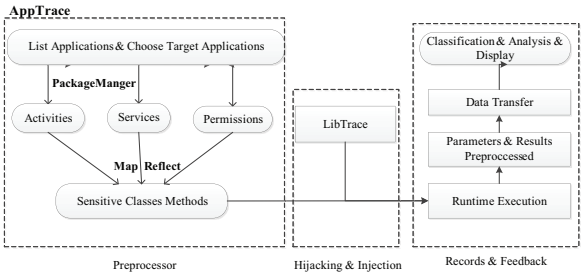

Fig. 2. AppTrace architecture overview.

TABLE I
LOADING TIME

AppPackageName	Size (MB)	Original (sec)	After Monitoring (sec)
com.xunxin	1.6	<1	<1
com.moji.mjweather	8.7	<1	>30
com.tencent.mm	22.6	<1	>55

III. DESIGN AND CHALLENGES

In this section, we discuss the challenges faced with our prototype and the outline of the prototype implementations.

A. The Challenges

1) Without Apks or Source Code: Assume that we have no access to either installation package of target applications or source codes. Static analysis of applications tends to depend on reverse engineering with apktools, dex2jar or other similar tools. However, considering that we do not accurately acknowledge the version of target applications, and the manufacturer might have changed the original source code to add or modify functionality. Therefore, we prefer not to use the apks or source code.

2) Running on Android Devices: When it comes to method trace or functions called records, some sandbox used to analyze application can achieve this goals, such as Droidbox [4]. However, to the best of our knowledge, most of such sandbox are deployed on Windows or Linux and may be troublesome for users to set up the necessary runtime environment. Furthermore, apks or source code are required for these tools, which reduces their availability.

3) Scalability: AppTrace can be used to trace sensitive APIs invoked by applications and classification of sensitive APIs should be defined in our process context. However, the diversity between normal APIs and sensitive APIs varies in different users. Besides, along with fast development of Android, new APIs are bound to be put forward or some old APIs may be modified. Under that circumstances, scalability becomes an tough issue. In addition to scaling availability of AppTrace, we must provide users with interface to improve the definition of sensitive APIs.

B. The Design

An overview of our AppTrace architecture is illustrated in Fig. 2. Referring to this figure, we divide our system into three parts or procedures by time sequence that each part is in action: Preprocessor, Hijacking and Injection, Records and Feedback.

1) Preprocessor: During the procedure of preprocessor, AppTrace determines which methods should be traced and make preparation for query from LibTrace. Android applications are composed of multiple packages and classes and there are plenty of methods invoked during runtime. In original, AppTrace intends to hook and monitor all methods in certain applications. This scheme is full-scale and exhaustive, but inevitably makes performance of target application worse. One primary principle of dynamic analysis is influencing the target application as less as possible. While being monitored by AppTrace, either system overhead or coefficient of performance should not be affected apparently. We demonstrates it with some experiments, and the results are showed in Tab. I.

In this experiment, we choose different applications with various sizes, and record time consumption between clicking an application icon and the first activity appearing. After comparing time overhead before and after monitoring applications, we find that the larger the application size, the more the performance overhead. So we apply another scheme that hook some sensitive APIs as an alternative. For example, while handling applications with permissions of Internet, AppTrace may only focus on APIs correlated to Internet operations.

2) Hijacking and Injection: Application Instrumentation is often used to insert some customized codes into target process under the circumstances that maintaining the integrity of the process logic. Analyzing feedbacks by these customized codes, we can obtain control flow or data flow information. In view of Android applications, traditional static instrumentation relies on smali codes de-compiled from Android's dex file. This technique finds position possible to be instrumented in smali files and inject customized codes. Finally, these tampered smali files need to be repackaged and installed into Android devices. The procedure is not only troublesome but also unstable. Errors of robustness or integrity often take place during its runtime.

Comparatively, dynamic instrumentation are considered as

a better option. With dynamic instrumentation, customized codes are injected dynamically while the application is being launched, which avoids complex procedure of static instrumentation. What's more, it does not require to modify the source codes, which may bring up integrity issues. Generally speaking, our AppTrace relies on the dynamic instrumentation and executes customized codes before or after methods called.

3) Records and Feedback: After the operations of approaches mentioned above, data about methods runtime can be obtained in the context of target applications. Next, AppTrace needs to acquire these data and interact with users. In order to interact with users in a friendly way, AppTrace will send a notification or make a toast to the forehead activity so as to notice that AppTrace is running. At the same time, LibTrace is busy in collecting information and transferring data to AppTrace.

IV. Implementation

A. Information Gathering

1) Activities and Services: A major goal of our AppTrace is to establish the relationship between user's behavior and executed codes. So AppTrace lists activities and services declared in Android-Manifest. Meanwhile, users can select a target activity or a service they are interested in. Traditional static instrumentation obtains activities or services belonging to certain application via decompiling the Android Manifest file. AppTrace approaches the application information with utilization of PackageManager, which retrieves various kinds of information related to the application packages that are currently installed on the device. Once one activity or service selected, AppTrace writes its class name, a part of Android Storage Options, into SharedPreferences, and provides a general framework that allows you to save and retrieve persistent key-value pairs of primitive data types. To a certain extent, SharedPreferences only transfers data in single application, but LibTrace can approach the xml file which deposit key-value pairs and obtain activity or service you selected.

2) Sensitive APIs: As third-party applications get access to system resources, Android system will check their permissions which are declared while install-time are unchangeable. Unlike all methods in activities or services that will be hooked and traced, only several or even none sensitive APIs will be hooked by LibTrace. Reasons are as followed,.Firstly, Android defines hundreds of permissions categorized into three threat levels and each permission may be associated with a number of methods. So if all methods have been hooked, performance overhead of the target application is sure to be lowered drastically and even the system shut down. Secondly, from a developer's or user's perspective, permissions or related methods are just strings. Definition of sensitive APIs are in terms of different applications or different usage situation.

AppTrace contains a built-in mapping table from permissions to correlated application programming interfaces. Additionally, AppTrace also provides users with interface to modify this mapping table. Due to the limitation of pages, here we only take some common permissions for example, such as SMS, device id etc. Details are showed in Tab. II.In our future work, the predefined mapping table will also be optimized completely.

TABLE II
Mapping Table

Resource	Permission	Class	API
SMS	send_sms	android.telephony.SmsManager	sendDataMessage sendTextMessage sendMultipartTextMessage
unique device ID	read_phone_state	android.telephony.TelephonyManager	getDeviceId
phone number			getLine1Number
serial number of SIM			getSimSerialNumber
unique subscriber ID			getSubscriberId
contacts	read_contacts	android.content.ContentResolver	query
internet	internet	org.apache.http.client.HttpClient	execute

B. LibTrace Injection

As stated in section II, Zygote is the parent of all App process and it preloads all necessary Java classes and resources before the first Dalvik VM being created. All applications can be considered as a copy of it originally, benefited from the fork mechanism in Linux. The startup of Zygote is triggered from the app_process by Init.rc after Service Manager and others. On account of the mechanism and function of Zygote, we would like to replace the app_process, which is located in /system/bin/app_process, so as to injecting our LibTrace into target applications.

Xposed Framework [2] takes advantage of this mechanism and it will replace the app_process file with a modified one. While new Dalvik VM is being created, some external jar packages are loaded, including our LibTrace, And these sensitive APIs will be replaced with certain native methods, which are finally redirected to our customized codes. The superiority of Xposed Framework lies in that we only need to focus on the application framework layer and our AppTrace is based on this framework. The overall course of hooking methods in target application is illustrated in Fig. 3.

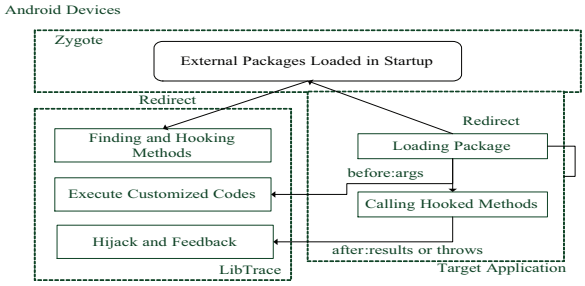

Fig. 3. Injection Procedure.

C. Calling LibTrace

Before or after the hooked methods are invoked in target application, LibTrace would be woke up and execute our customized codes. Prime missions of LibTrace here are two steps: collecting runtime information and pass them back to AppTrace. Generally speaking, LibTrace will record parameters and results of the hooked methods. It will find out classes of parameters and dispose of them roughly.

Disposal of activities or services may be more complex. Because AppTrace is designed to associate users' behavior with codes, all methods in activities or services should be hooked. However, there are so many methods called throughout activities' life-cycle and it is difficult to hook these methods one by one. Therefore, AppTrace invoke the Debug class provided by Android SDK to generate log files for certain methods. Debug can create log files that give details about an application, such as a call stack and start/stop times for any running methods. It can provide various debugging functions for Android applications, including tracing and allocation counts. LibTrace calls the startNativeTracing function before a hooked method running and stops Debug after the method being over.

At last, LibTrace will transfer information collected in the past to AppTrace via the ContentProvider initialized by AppTrace. In order to avoid blocking target application, LibTrace prefer to start new threads. Some time consuming tasks are implemented in these threads, such as reading data from log files created by Debug. Due to Android permissions' mechanism, these log files can only be approached in the context of target applications.

D. Analysis and Display

While receiving data passed from LibTrace, AppTrace will send flags back and restore these data into SQLite. How to analyze these data with great redundancy and obtain important information is another problem. AppTrace are concentrated on two aspects: traversing data from log files to discover whether there has any hidden APIs, and handle sensitive APIs' invoked sequence to match any harmful behavior. Single call of sensitive APIs is normal, but series call of sensitive APIs that has been invoked may be suspicious. Finally, AppTrace offers end-users several interface to retrieve and check trace information.

V. Evaluation

In this section, we present the evaluations of our AppTrace prototype. The goals of the evaluations are threefold: *1) AppTrace must be demonstrated to be stable and usable for end users. 2) AppTrace must be capable of tracing most of sensitive methods selected by users. 3) Performance loss incurred by AppTrace overhead must be measured.*

A. Case Study

In this part, we take an application as a case study. WeChat is a popular social networking software among released by Tencent and its package name is com.tencent.mm [8]. At first we aim to LauncherUIActivity class so all its methods declared in this activity would be hooked in LibTrace. In addition, all sensitive APIs related to its required permissions would be hooked by default.

While the target application running and LibTrace been triggered, LibTrace will provide users with notification, as illustrated in Fig. 4.

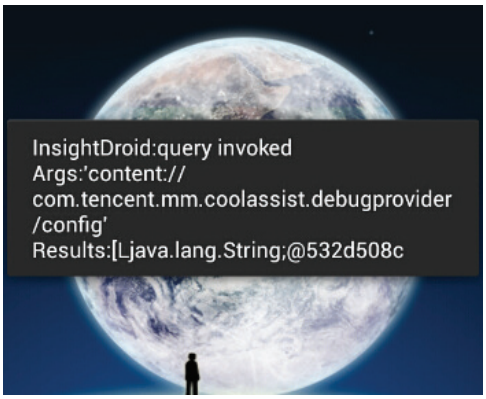

Fig. 4. LibTrace, running in the target application and notifies users.

TABLE III
Stability Test on Different Playforms

Version	API Level	Stable
Ice Cream Sandwich	14	√
	15	√
Jelly Bean	16	√
	17	√
	18	√
KitKat	19	√

After LibTrace fulfills its mission in the target application and transfers data back to AppTrace, it also sends a feedback and users can read the method trace log in AppTrace. For each activity or service, AppTrace lists sensitive methods belonging to it and hooked by LibTrace. If certain sensitive method in list is clicked, AppTrace will display package name, method name, method parameters and trace logs in a table. Details are illustrated in Fig. 5.

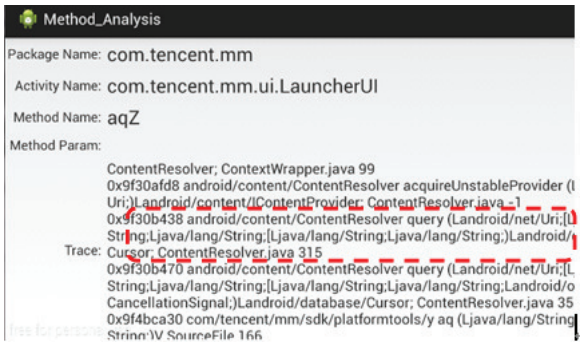

Fig. 5. Results of runtime information for a certain method.

B. Stability

Referring to official statistics from Google [1], more than 80 percent of Android devices have been updated beyond Android 4.0. Therefore we launch AppTrace on several devices including simulators and real ones. The results in Tab. III demonstrate that AppTrace is compatible with Android versions beyond 4.0.

TABLE IV
TRACE INFORMATION OF SENSITIVE APIS

ApplicationName	Trace log	send_sms	read_phone_state	read_contacts	internet
WeChat	√	none	none	query	execute
PPTV	√	none	getDeviceId	none	execute
XunFengZhongZi	√	none	none	none	execute
I'go Reader	√	none	getDeviceId	query	execute
QuanXianGuanLi	√	none	none	none	execute
Pico TTS	√	none	none	none	execute
NetFix	√	none	none	none	execute
Youni	√	sendTextMessage	getDeviceId	query	execute

C. Functional Evaluation

First, we randomly select 8 real Apps from Google Play to evaluate AppTrace. Our methodology for this evaluation was to: (a) decompile the target App by apktools and search all sensitive APIs in samli codes; (b) install AppTrace; (c) run the App and randomly select certain component to observe whether AppTrace generates trace log; (d) record the sensitive APIs AppTrace warned and compare with the results in step one. Experimental results are illustrated in Tab. IV.

In the table above, none means this application does not declare the permission. The table shows that AppTrace could trace all the target methods successfully and trigger the call-back function efficiently when certain sensitive API has been called. However, we also find some sensitive APIs failing to be captured. After debugging with logcat tools, we infer that there are several possible reasons:

1) Overloading Methods: This means that methods within a class can have the same name if they have different parameter lists. Because methods are hooked according to their names, it is likely to hook incorrect method when we resolve its parameters referring to the other overloading method.

2) Encapsulation: Some sensitive APIs encapsulated for convenient development are not the real execution function. For example, developers tend to use the 'org.apache.http.client. HttpClient.execute' function to initiate web request. But in experiments we find that this function cannot be hooked, while the function: org.apache.http.impl.client.AbstractHttp Client. execute can be hooked, which is invoked by org.apache.http.client.HttpClient.execute. Fortunately, we could look up the trace log and find out the real invoked methods.

3) Sub threads: If some sensitive APIs are invoked in certain sub thread started by the main thread, AppTrace have the possibility to fail in hooking. This is due to the exception introduced by Xposed framwork.

Finally, we pick 50 open source Apps in the F-Droid for further test. The results show that AppTrace could trace all target methods in 50 Apps, i.e., the success rate is 100%. It also triggers 317 times sensitive API calls in a total of 382. The accuracy rate is 83%.

D. Performance Overhead

To compute the overhead that is introduced by AppTrace, we use two ways: CPU utilization rate and Java Mircobench-mark.

1) CPU utilization rate: First, we choose three popular Apps in Xiaomi market [9] which is a Chinese alternative application market. Then we test the CPU utilization rate by Linux top command and compare the performance differences between the original system and that loaded with AppTrace. To decrease the influence introduced by random events, each app should be tested ten times and we compute the average value of them. Fig. 6 displays the test results. We can find that the CPU utilization rate has been increased 20.7% averagely. We believe this overhead is acceptable in consideration of the temporality during CPU run and the resource abundance with the development of Android devices.

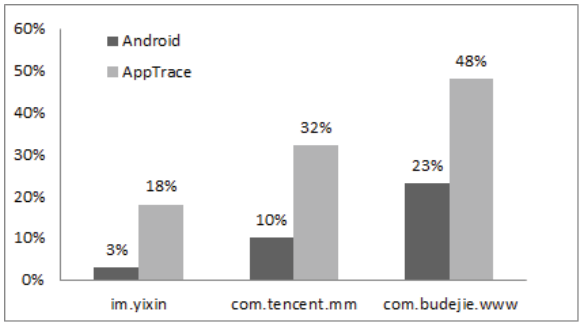

Fig. 6. CPU utilization rate.

2) Java Mircobenchmark: The CaffeineMark [10] is a series of tests that measure the speed of Java programs running and scores correlate with the number of Java instructions executed per second. We used an Android port of the standard CaffeineMark 3.0. Results for the benchmark tests are depicted in Fig. 7. Studying the results, we conclude that the method benchmark experiences the greatest overhead while that is the string in TaintDroid. This is under control because our targets are member methods. The overall results indicate cumulative score across individual benchmarks. Here, the original Android system had an average score of 12990, and AppTrace measured 11784. AppTrace has a 9% overhead with respect to the original system.

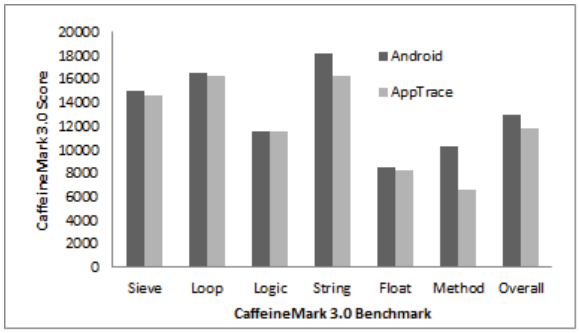

Fig. 7. Java microbenchmark.

VI. RELATED WORK

Android application analysis has been a hot field in academic research. Previous works usually look at permission, code, and runtime behavior.

*Permission.*The Kirin install-time certification system, proposed by Enck et al. [11], was the first security policy extension for Android. Pandita et al. [12] proposed WHYPER framework on USENIX Security 2013. They used Natural Language Processing (NLP) techniques to identify sentences that describe the need for a given permission in an application description.

*Code.*Chin et al. [13] proposed ComDroid, which operates on use disassembled DEX bytecode. Grace et al. [14] developed Woodpecker which does control flow analysis based on the smali code. However, above techniques belong to static analysis which not sufficient in analyzing the GUI components such as button, and they do not have runtime information. Our AppTrace makes use of the dynamic analysis to trace Java methods and could help the static analysis.

*Runtime behavior.*Enck et al. [15] proposed TaintDroid to identify when applications send privacy sensitive information to network servers by dynamic taint tracking analysis. Based on TaintDroid, researchers proposed several improvements such as AppFence [16], AppsPlayground [17]. But the sandbox environment is different from the official running environment and not compatible with all Android versions. In contrast, our AppTrace system could run stably on any version above Android 4.0.

VII. Conclusion and Future Work

In this paper, we present AppTrace, a prototype system based on dynamic instrumentation, to help analyzing and tracing methods invoked during applications' runtime. Any normal or sensitive action can be mapped into a series of methods. So we can have insight into the target applications runtime features. The evaluation results have shown that AppTrace succeed to report sensitive APIs invoked in various applications.

Currently, the predefined mapping table between permissions and sensitive APIs are incomplete. In addition, the dynamic instrumentation framework which AppTrace relies on may be imperfect. In the further work, we intend to learn the method introduced by Felt et al. [18] to match API calls and permissions for completing the predefined mapping table. Besides, we could modify the Xposed dynamic instrumentation framework or adopt other framework such as Dynamic Dalvik Instrumentation Toolkit [3] to improve the efficiency of LibTrace and decrease performance overhead. Moreover, in this paper, we havent aim for the problem whether the call of sensitive APIs results in privacy leakage or not yet. We notice that the AppIntent [19] has done many works on it which is worthy of our further study. At last, from the point of privacy protection, we could use some data to replace original sensitive data in case of privacy leakage.

ACKNOWLEDGEMENTS

The authors would like to thank the reviewers for their detailed reviews and constructive comments, which have helped improve the quality of this paper. Thanks Daoyuan Wu, who is now studying in the Hong Kong Polytechnic University (PolyU). Daoyuan has given us so many good suggestion about this work. This paper is supported by the National Natural Science Foundation of China (No. 61373006), the Foundation of Nanjing University of Posts and Telecommunications (No. NY213160).

References

[1] Google, "Dashboards,"http://developer.android.com/about/dashboards/index.html, March 20, 2014.
[2] Rovo89, "[Framework only!] Xposed - ROM modding without modifying APKs (2.6.1)," http://forum.xda-developers.com/xposed/framework-xposed-rom-modding-modifying-t1574401, May 20, 2014.
[3] Crmulliner, "DDI - dynamic dalvik instrumentation toolkit," https://github.com/crmulliner/ddi, January 6, 2014.
[4] Google Project Hosting, "DroidBox, Android application sandbox," http://code.google.com/p/droidbox, March 20, 2014.
[5] Realtime Privacy Monitoring on Smartphones, http://www.appanalysis.org, April 14, 2014.
[6] Google play, https://play.google.com/store, March 14, 2014.
[7] Free and Open Source App Repository, https://f-droid.org/,March 14, 2014.
[8] Google play, https://play.google.com/store/apps/details?id=com.tencent.mm&hl=zh_CN May 20, 2014.
[9] Xiaomi market. http://app.mi.com, May 24, 2014.
[10] CaffeineMark. http://www.benchmarkhq.ru/cm30/index.html. May 24, 2014.
[11] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," *in Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009, pp. 235-245.
[12] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "WHYPER: towards automating risk assessment of mobile applications," *in Proceedings of the 22nd USENIX Security Symposium, Washington DC, USA*, 2013, pp. 14-16.
[13] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," *in Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, 2011, pp. 239-252.
[14] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang, Systematic Detection of Capability Leaks in Stock Android Smartphones., *in NDSS*, 2012.
[15] W. Enck, P. Gilbert, B. G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *in Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, 2010, pp. 1-6.
[16] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These aren't the droids you're looking for: retrofitting android to protect data from imperious applications," *in Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011, pp. 639-652.
[17] V. Rastogi, Y. Chen, and W. Enck, "AppsPlayground: automatic security analysis of smartphone applications," *in Proceedings of the third ACM Conference on Data and Application Security and Privacy*, 2013, pp. 209-220.
[18] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," *in Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011, pp. 627-638.
[19] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. Wang, "Appintent: Analyzing sensitive data transmission in android for privacy leakage detection," *in Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 1043C1054.