uses, we modify Android system to monitor API invocations and permission uses that are related to privacy information exposure and stealthy charges. Then we develop an automated testing platform to load the customized Android system in order to automatically install and start up apps, simulate user operations, inject SMS and phone-call events, and collect app behavior records.

A. *API invocations inspection*

According to TaintDroid [4], we implement the taint tracking technique on the 4.4.2_r2 branch of AOSP to check whether the data sent out contains taint in order to monitor privacy relevant API calls. The taint tracking technique uses a 4-byte unsigned integer to describe the taint field. Each bit represents a type of privacy data. If more than one bit are set to 1, it indicates that this data is relevant to multiple types of privacy information.

As Fig. 1 shows, apps may invoke some APIs to access certain privacy information during execution. We modify the JAVA primary data types of Boolean, Double, Float, and Integer as well as the encapsulated data type of String, in order to add the taint field for each type. When the data is accessed by an API, taints are added and the corresponding bit of the privacy data type is set to 1 in the invoked API. Then, any further operations on the data including numerical calculation, truncation, concatenation, type conversion and encryption, are composed of the basic instructions in Dalvik VM. Therefore, we modify the basic instructions in Dalvik VM to maintain the taints. When the data is sent, taints can be detected in the Socket I/O functions to determine which types of privacy data are sent by checking the bits of the taint field.

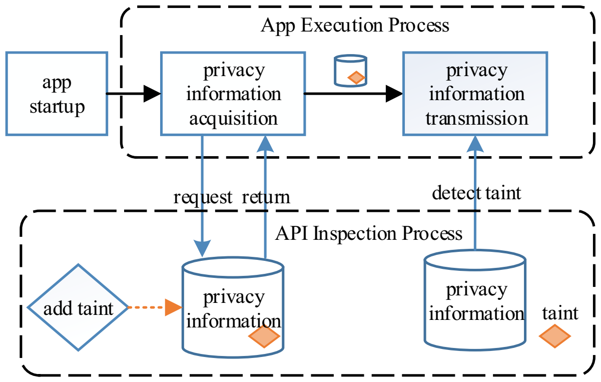

Fig. 1. API Invocations Inspection

We add log functions in 3 positions to fully monitor API invocations: (1) add a taint to the data in the procedure of data requests; (2) check the taint in the procedure of data transmission over network; (3) invoke charges-relevant APIs, including sending SMS messages and making phone calls.

Since the API name may be changed in different system versions, we should translate the recorded API invocations into permission checks as [5] explains. Because all the APIs related to charges and privacy information correspond to certain permissions [6], it is easy to perform the translation.

B. *Permission use inspection*

Since the taint tracking technique cannot cover all types of malicious behaviors such as privacy leakage in native code, we monitor permission checks to completely record the behaviors of apps. If the sensitive API is invoked, the Android system will check whether the caller app obtains the corresponding permission in the installation process. As shown in Figure 2, permission check in Android system is executed in 2 different system layers, i.e., the framework layer and the kernel layer. We record the permission uses of apps by monitoring the permission checks in these two system layers.

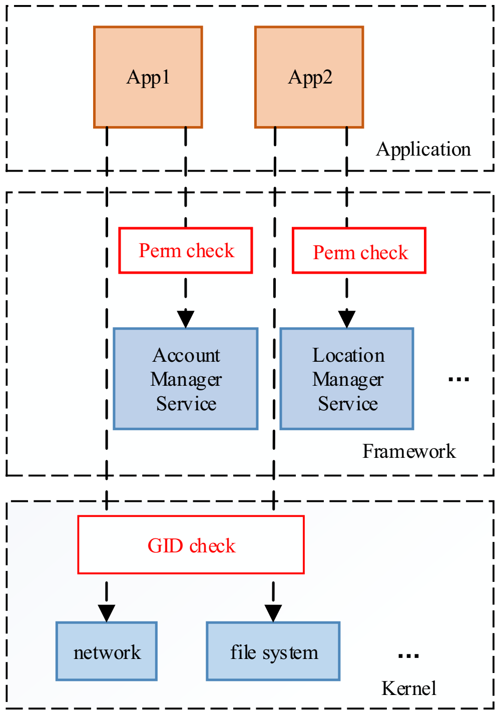

Fig. 2. Permission Use Inspection

Most of the system permissions are checked in the framework layer. In particular, the checkUidPermission() method in PackageManagerService is called to check if the caller has the permission to use the current API. Therefore, we add a log function in the checkUidPermission() method in PackageManagerServcie in the framework layer to record the caller's UID (User ID), timestamp and permission name.

The rest system permissions, including network, file system, Bluetooth, system log, etc., are checked in the kernel layer using the original file access control mechanism in the Linux system to check API caller's permission. Specifically, the mapping relationships between these system permissions and kernel GIDs (Group ID) are described in the system file, i.e., /system/etc/permissions/platform.xml, and are checked in two functions, i.e., in_group_p(gid_t grp) and in_egroup_p(gid_t grp), in the source code, i.e., kernel/groups.c. Consequently, we add a log function in these two functions, i.e., in_group_p() and in_egroup_p(), to record the caller's UID, timestamp and GID in the kernel log, and then obtain the corresponding system permission names by using the mapping relationship between permission name and GIDs. In this way, we can monitor the permission uses in both framework layer and kernel layer.

C. Automated Testing Platform

To automatically collect behavior records from large amounts of apps, we develop an automated testing platform to run these app samples on our customized Android system. Our platform can install, start up apps, and collect behavior records automatically. To make the apps reveal more behavioral characteristics on the platform, we simulate user operations using the application exerciser tool Monkey [7] and inject SMS and phone-call events through telnet commands.

For a given app, the automated testing platform works as the following steps.

Step 1 Extract the package name and the launchable activity name from the manifest file, i.e., AndroidManifest.xml, in current app for analysis purpose.

Step 2 Make a replica of the customized Android system and run it on an Android emulator.

Step 3 Redirect the user-space and kernel-space system logs onto the local disk using the adb tool.

Step 4 After the emulator is launched, the app is installed into the emulator.

Step 5 Launch the app and run the Monkey tool to exercise 300 random operations on the app with an interval of 3 seconds between two continuous operations.

Step 6 After the monkey completes its operations, log on the emulator with Tc0. mand send two events of receiving a phone call and two events of receiving a message.

Step 7 Shut down the emulator and delete the replica of Android system image.

Step 8 Repeat steps 1 to 7 until all the apps have been tested.

Each app runs for about 18 minutes following The above steps. As several errors may occur during the test,1.693the output information of step 3 to step 5 is parsed to detect the errors occurred in these steps. If the errors are identified, our platform terminates the emulator and excludes the app.

III. Malicious Behavior Detection

Malicious behavior detection includes3feature selection and malicious behavior recognition.

A. Feature Selection

We extract user-relevant and dynamic behavioral features to restore apps' run-time characteristics for classification. The user-relevant features indicate whether a sensitive permission or API use is caused by user operations. The dynamic behavioral features include whether the used permissions match a certain sensitive permission combination, and whether the overall sequence of API invocations and permission uses contains some sensitive sequences.

1) user-relevant feature

If charge-related activity or privacy information acquisition and transmission is caused by user operations, e.g., screen touch and swipe, behavior cannot be classified as malicious. Therefore, we detect user operations in order to reduce false alarms of our malicious behavior detection method.

Since the user operations on our platform are simulated by using Monkey, we explore the source code of both Android system and Monkey in order to record the simulated user operations. In particular, we find that Monkey tool obtains an instance of WindowManager class in Android framework, and then injects user operation events by calling the injectKeyEvent() method in the WindowManager class. The WindowManagerService in the Android framework receives the events and dispatches them to corresponding app user interface windows in the foreground. As the InputEventReceiver used by WindowManagerService invokes dispatchInputEvent() method in this procedure, we add a log function to record the user operation events sent from the Monkey tool.

2) dynamic behavioral features

Dynamic behavioral features consist of sensitive permission use combinations and sensitive behavior sequences obtained at the app run time.

Sensitive permission use combination: Commonly, to steal privacy information, a malware sample requires a series of permissions [8]. Thus, we analyze extensive permission uses of both malicious and benign apps and extract the sensitive permission combinations by Association Rules Mining in order to use them to detect malware. Table I shows the permission use combination. However, these sensitive permissions can be used by benign app as well. Take the two permissions READ_CONTACTS and INTERNET requested by an instant messenger app, e.g., QQ, as an example. QQ needs the INTERNET permission for communication. Besides, QQ can use user contacts for friend recommendation, which requires the READ_CONTACTS permission. As a result, the combination of these two permissions for malware detection can cause false alarm. To address this issue, we adopt sensitive behavior sequences to increase the detection accuracy and reduce the false alarm.

TABLE I SENSITIVE PERMISSION COMBINATIONS EXTRACTED BY ASSOCIATION RULES MINING

No.	Sensitive permission combinations
1	INTERNET, WRITE_SMS
2	READ_PHONE_STATE, WRITE_SMS
3	INTERNET, READ_PHONE_STATE, READ_SMS
4	ACCESS_WIFI_STATE, READ_SMS, WRITE_SMS
5	WRITE_SMS, RECEIVE_SMS, RESTART_PACKAGES
6	READ_PHONE_STATE, WRITE_SMS, WRITE_CONTACTS
7	WAKE_LOCK, WRITE_CONTACTS, RESTART_PACKAGES
8	READ_SMS, SEND_SMS, RECEIVE_SMS
9	READ_SMS, RECEIVE_SMS, RESTART_PACKAGES
10	WRITE_EXTERNAL_STORAGE, READ_SMS, WRITE_SMS

Sensitive behavior sequences: API invocations and permission uses are both the invocation of functions outside the app sandbox, so the combination of their inspection results can relatively completely restore the behavior of an app. The transferred data has been added taint to indicate the type of privacy information, so that the acquisition and transmission operations can be mapped into the corresponding permissions. As a result, we merge the mapped permissions from API invocations with the permission check records according to timestamps so as to get a syncretic sequence S_{all}.

We run the malware dataset in [9] on the automated testing platform to obtain behavior records. Since the malware samples are categorized by their malicious behavior, we *a priori* know what malicious behavior the samples will perform. In total, 24 malicious sequence patterns $S_{malicious}=\{s1, s2, \ldots, s24\}$ are manually extracted from the behavior data of malware, each corresponding to one malware category.

If we perform a complete matching with each malicious sequence pattern as a classification feature, we will only get 24 features to describe whether the behavior sequence contains malicious behavior sequence patterns. However, complete matching might ignore apps' performing part of the malicious behavior, causing high false negative rate. Moreover, we need to detect potential malicious behavior as sooner as possible. Hence, we use every partial sequence pattern of a maximum length N ($2 \le N \le 5$) as a feature, denoted by *PSP* for short. The number of features denoted by M varies if N is assigned with different values. We will discuss about the selection of value N in section IV.

B. Malicious Behavior Recognition

In order to detect malicious behavior of an app, the behavior sequence needs to be compared with the malicious sequence patterns. We present an Online Multi-mode Sequence Matching algorithm to match the malicious sequence patterns.

Note that when a malware sample is performing malicious activities, there will also be normal operations which should be filtered out during the matching of malicious sequence patterns.

First, we design a data structure used by the algorithm. Fig. 3 shows the data structure that is based on trie tree, with two modifications made.

1) A field representing the node number is added to every tree node in addition to the original pointers pointing to child nodes and a flag indicating whether it forms a *PSP*.

2) A hash map is added which maps the node number to the memory address of each node in order to jump between nodes freely.

Similar to trie tree, a path from the root down to a certain node indicates whether the current sequence fragment forms a *PSP*, and the whole tree is made up of all the M *PSP*s. It is easy to see that the tree has M nodes that own flags indicating a *PSP*. With this tree, we can match the malicious sequence patterns with the behavior sequence for analysis. TABLE II shows the Online Multi-mode Sequence Matching algorithm.

The output of the algorithm is a list of values, each of which indicates whether the corresponding *PSP* is satisfied when matching with the current behavior sequence. If the current behavior sequence satisfies some of the *PSP*s, the corresponding features' values are 1, and other features' values are 0. If none of the nodes is the end of a *PSP*

IV. Evaluation

A. *Experimental Data Acquisition*

The automatic app testing platform runs on a PC that installs the Ubuntu 14.04 system and has an i7-4700 CPU and 8GB memory. We run 7 emulators simultaneously to make full use of the hardware resources.

The malware dataset used in this paper is obtained from Android Malware Genome Project[9], which has 1243 malware samples in 34 categories. Other apps are downloaded from Google Play and a third-party alternative market Anruan. 12582 apps from Google Play are treated as benign apps which are all top 500 on the most popular apps lists in all categories. 14733 apps from Anruan Market are also on the top lists. After running them on the testing platform and preprocessing the data, we finally obtained behavior data from 504 malware samples, 5193 apps from Google Play and 3917 apps from Anruan Market. 24 malicious sequence patterns are extracted from 24 categories of malware which contain 390 samples in total.

The training set is as follows. The behavior records from malware samples that containing *PSP*s are marked as malicious, while all the other records are marked as benign. In addition, we randomly select behavior records of equivalent numbers from Google Play apps and marked them all as benign. There are 120641 records in total, 117741 benign and 2900 malicious. Behavior records of apps from Anruan Market are used for open-world analysis.

to 5 and reach an accuracy of 99.0% with a 1.0% false positive rate and a 2.3% false negative rate.

We mark an app as malicious if there is at least one of its behavior records that is classified as malicious. 280 malware samples out of the 390 total samples are detected, accounting for 71.8%. By manually checking the behavior records from malware samples that are classified as benign, no obvious evidences of malicious behavior are found. It results from that the missed malware samples have not performed malicious behavior during our test. This is probably because these malware samples are designed not to perform malicious activity without meeting some triggering conditions, such as adequate running time, certain operations on the apps, connection to a remote server, etc.

We also use the classification model to detect malware from Anruan Market. As a result, 952 apps are marked as malicious out of 3917 ones, accounting for 24.3%. We randomly select 120 apps that are marked as malicious, manually check the behavior records, and find that most of them performed certain malicious behavior as TABLE III shows. Most of them fetch IMEI or IMSI and then send it to a remote server through the Internet. We can see that quite a proportion of apps utilize the IMEI and IMSI as the identifiers of devices, which can be used to track users. There are still 24 apps that don't perform malicious behavior but are marked as malicious, causing false alarm. These 24 apps generated 2167 entries of records, 75 of which are classified as malicious, accounting for 3.5%. So the false positive rate is still very low at the behavioral level.

TABLE III MANUAL ANALYSIS OF ALARMED APPS IN ANRUAN MARKET

Malicious behavior	Number of apps with this behavior
Steal location	20
Steal phone number	9
Steal messages	2
Steal IMEI	77
Steal IMSI	51
Steal ICCID	3
Steal browse history	1
None	24

The sum of the numbers in the table is more than 120 because one app may have several types of malicious behavior

V. RELATED WORK

Techniques for detecting Android malware can be categorized into three types, namely static analysis techniques, dynamic analysis techniques and hybrid analysis techniques.

A. *Static Analysis*

Static analysis is commonly used in analyzing the security of software. Static analysis of Android apps can be classified into installation package analysis, bytecode analysis and source code analysis according to the targets of analysis.

Installation packages analysis is first proposed by Enck *et al.* [10]. They analyze the requested permissions for possible malicious functions by breaking malicious functions up into corresponding permissions. Zhou *et al.* [11] collected a large set of malware samples and analyzed the permission use in depth. Many subsequent researches used this dataset for malware detection. MAST[12] collects information of permission requests, whether having native code, self-startup behavior, etc., to rank the risks of apps and decide which apps require deeper scan. Similarly, DREBIN[13] collects more information available in the installation package, and it is suitable for installation-time analysis concerning its performance. Binary code or byte code analysis is to scan compiled code in the installation files for malicious function features. For example, ComDroid[14] analyzes inter-application communication to find malicious behavior such as broadcast interception, service hijacking. However, analyzing binary code or byte code is tough work, so more researches decompile the byte code before scanning. Aafer *et al.*[15] research into the API call features after decompiling, including API name and API parameters, while this method is probably be deliberately evaded by malware developers. DroidSIFT[16] constructs calling dependency graphs based on semantics to avoid detection evasion, which results in lower false negative rate and false alarm rate.

In addition to malware analysis, static analysis can also be used to detect vulnerabilities in apps which can be utilized by malware, such as inter-application communication vulnerabilities (ComDroid[14]), component hijacking[17], capability leaks[18]. Static analysis is generally suitable for offline analysis, and can be used for market-scale apps analysis, but is ignorant of run-time context and can be easily evaded by malware developers. Thus, some researchers propose dynamic analysis techniques for detecting anomaly at run time. Dynamic analysis is usually more suitable for being deployed on user devices and can intercept malicious behavior timely.

B. *Dynamic Analysis*

The most typical work of dynamic analysis is TaintDroid[4] proposed by Enck *et al.* which adds taint to privacy data for propagation and tracks the taint flow to detect privacy information leakage. It can find out privacy leakage effectively at run time, but fails to track leakage occurred in native code. Furthermore, when deployed on a smartphone, it demands for non-negligible CPU usage and energy consumption. VetDroid[19] is a privacy leakage detection system based on taint tracking that not only monitors explicit permission use but multiple code exploration techniques to analyze privacy leakage and malicious functions durin app's run time.

C. Hybrid Analysis

Some researchers combine the static and dynamic analysis techniques to gain the efficiency and flexibility of static analysis and accuracy of dynamic analysis simultaneously. DroidRanger[25] filters suspected malicious apps by analyzing permission requests and byte code, then further explores malicious behavior using dynamic system call inspection. It is also effective in finding out zero-day malware from market-scale app samples.

VI. Conclusion and Future Work

In this paper, we propose an analysis framework for monitoring, recording and analyzing app behavior at run time, in order to detect malicious behavior of Android apps. Our scheme has a high detection rate for detecting malicious behavior in known malware families, and finds out a relatively high proportion of malware samples. For those missed malware samples, we will further do research to provide a runtime environment in the emulator that is more close to a real smartphone to trigger more behavior of the apps in order to raise the detection rate of malware.

Acknowledgment

This work is supported by National Natural Science Foundation of China under Grants No. 61272054, 61572130, 61320106007, 61502100, 61532013, and 61402104, Jiangsu Provincial Natural Science Foundation under Grants BK20140648 and BK20150637, the Fundamental Research Funds for the Central Universities under Grant 2242014R30010, Jiangsu Provincial Key Technology R&D Program under Grant BE2014603, Qing Lan Project of Jiangsu Province, Jiangsu Provincial Key Laboratory of Network and Information Security under Grant BM2003201, and Key Laboratory of Computer Network and Information Integration of Ministry of Education of China under Grant 93K-9.

References

[1] Worldwide Smartphone Market Will See the First Single-Digit Growth Year on Record, According to IDC [EB/OL]. http://www.idc.com/getdoc.jsp?containerId=prUS40664915.

[2] LBE Tech [EB/OL]. http://www.lbesec.com/#/products/2.

[3] 360 Security [EB/OL]. http://www.360securityapps.com/en-us.

[4] Enck W, Gilbert P, Chun B-G, et. al. TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones [C]. In Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI), Berkeley, CA, USA, 2010, pp. 1-6.

[5] Manifest.permission [EB/OL]. https://developer.android.com/reference/android/Manifest.permission.html

[6] Au K, Zhou Y, Huang Z, Lie D, Gong X, Han X, and Zhou W. Pscout: Analyzing the android permission specification. In Proceedings of the 19th ACM conference on Computer and communications security (CCS), 2012: 217-228.

[7] UI/Application Exerciser Monkey [EB/OL]. https://developer.android.com/studio/test/monkey.html

[8] Enck W, Ongtang M, McDaniel P. On lightweight mobile phone application certification [C]. In Proceedings of the 16th ACM conference on Computer and communications security. ACM, 2009: 235-245.

[9] Android Malware Genome Project [EB/OL]. www.malgenomeproject.org.

[10] Enck W, Ongtang M, and Mcdaniel P. Mitigating Android software misuse before it happens [R]. Tech. Rep. NAS-TR-0094-2008, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA, 2008.

[11] Zhou Y, and Jiang X. Dissecting Android Malware: Characterization and Evolution [C]. In Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P), 20-23 May 2012.

[12] Chakradeo S, Reaves B, Traynor P, et. al. MAST: triage for market-scale mobile malware analysis [C]. In Proceedings of the 6th ACM conference on Security and privacy in wireless and mobile networks (WiSec), Budapest, Hungary, 2013, pp. 13-24.

[13] Arp D, Spreitzenbarth M, Hübner M, et. al. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket [C]. In Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS), February 2014.

[14] Chin E, Felt A P, Greenwood K, and Wagner D. Analyzing inter-application communication in Android [C]. In Proceedings of the 9th international conference on Mobile systems, applications, and services (MobiSys). ACM, 2011.

[15] Aafer Y, Du W, and Yin H. DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android [C]. In Proceedings of the 9th international conference on Security and Privacy in Communication Networks (SecureComm), Sydney, Australia, Springer International Publishing, Sept. 25-27, 2013, pp. 86-103.

[16] Zhang M, Duan Y, Yin H, et. al. Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs [C]. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS), Scottsdale, Arizona, USA, 2014, pp. 1105-1116.

[17] Lu L, Li Z, Wu Z, et. al. Chex: Statically vetting android apps for component hijacking vulnerabilities [C]. In Proceedings of the 19th ACM conference on Computer and communications security (CCS), 2012.

[18] Chan P P, Hui L C, and Yiu S M. Droidchecker: analyzing android applications for capability leak [C]. In Proceedings of the 5th ACM conference on Security and Privacy in Wireless and Mobile Networks (WiSec), 2012.

[19] Zhang Y, Yang M, Xu B, et. al. Vetting undesirable behaviors in android apps with permission use analysis [C]. In Proceedings of the 2013 ACM SIGSAC conference on Computer and Communications Security (CCS), Berlin, Germany, 2013, pp. 611-622.

[20] Reina A, Fattori A, and Cavallaro L. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors [C]. In Proceedings of the 6th European Workshop on Systems Security (EuroSec). Prague, Czech Republic, April, 2013.

[21] Tam K, Khan S J, Fattori A, Cavallaro L. CopperDroid: Automatic Reconstruction of Android Malware Behaviors [C]. In Proceedings of the Network and Distributed System Security Symposium (NDSS), 2015.

[22] Yan L K, Yin H. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis [C]. In Proceedings of the 21st USENIX Security Symposium (USENIX Security). 2012: 569-584.

[23] Shabtai A, Tenenboim-Chekina L, Mimran D, et. al, Mobile malware detection through analysis of deviations in application network behavior [J]. Computers & Security, vol. 43, pp. 1-18, 2014.

[24] Rastogi V, Chen Y, and Enck W. AppsPlayground: automatic security analysis of smartphone applications [C]. In Proceedings of the 3rd ACM conference on Data and application security and privacy (CODASPY), San Antonio, Texas, USA, 2013, pp. 209-220.

[25] Zhou Y, Wang Z, Zhou W, et. al. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets [C]. In Proceedings of the 19th Network & Distributed System Security Symposium (NDSS), Hilton San Diego Resort & Spa, 2012.