

SPECIAL ISSUE PAPER

Detection of malicious behavior in android apps through API calls and permission uses analysis

Ming Yang  | Shan Wang | Zhen Ling | Yaowen Liu | Zhenyu NiSchool of Computer Science and Engineering,
Southeast University, Nanjing, China**Correspondence**Ming Yang, School of Computer Science and
Engineering, Southeast University, Nanjing,
China.

Email: yangming2002@seu.edu.cn

Funding informationNational Natural Science Foundation of China,
Grant/Award Number: 61572130,
61320106007, 61502100, 61532013, and
61402104; Jiangsu Provincial Natural Science
Foundation, Grant/Award Number:
BK20140648 and BK20150637; Jiangsu Pro-
vincial Key Technology R&D Program, Grant/
Award Number: BE2014603; Qing Lan Project
of Jiangsu Province, Jiangsu Provincial Key
Laboratory of Network and Information Secu-
rity, Grant/Award Number: BM2003201; Key
Laboratory of Computer Network and Infor-
mation Integration of Ministry of Education of
China, Grant/Award Number: 93K-9**Summary**

In recent years, with the prevalence of smartphones, the number of Android malware shows explosive growth. As malicious apps may steal users' sensitive data and even money from mobile and bank accounts, it is important to detect potential malicious behaviors so as to block them. To achieve this goal, we propose a dynamic behavior inspection and analysis framework for malicious behavior detection. A customized Android system is built to record apps' API calls, permission uses, and some other runtime features. We also develop an automated app behavior inspection platform to install and inspect massive samples so as to collect apps' dynamic behavior records. Then these records are exploited to train a string subsequence kernel-based Support Vector Machine (SVM) model, which can be used to classify benign and malicious behaviors offline. To realize online detection, we further extract apps' runtime features including sensitive permission combination uses, sensitive behavior sequences, and user interactions for behavior classification. The classification results can reach an accuracy of 84.9% in offline phase and 99.0% in online phase. Besides, we verify our scheme for identifying malicious apps, and the results show that 71.8% instances of malware samples are identified by running each app for only 18 minutes.

KEYWORDS

Android malware, API calls analysis, dynamic behavior analysis, permission use analysis, string subsequence kernel

1 | INTRODUCTION

In recent years, smartphone shipments have experienced a fast growth. It was predicted¹ that the global smartphone shipments will grow to 1.54 billion units by 2019 and the Android market share will increase to 82.6%. The openness of the Android platform not only leads to its rapid growth but also provides a convenient way for malware development. While smartphone apps facilitate our life and work by meeting our demands of various social activities, diverse malwares pose a severe threat on our privacy information stored on the devices. The most common malicious activities include stealing user privacy information, sending premium-rate SMS messages, etc. Furthermore, malware may even steal money from smartphone users' mobile and bank accounts.

The security issues of Android platform have recently attracted substantial attention in both industry and academia. Products from smartphone security software companies, such as LBE² and 360 Security,³ allow users to accept or deny some sensitive permission uses. But there are 2 disadvantages of this type of solution. Firstly,

most users do not have enough professional knowledge and security awareness to determine whether the current permission use is harmless or not. Secondly, these security apps require to gain root privileges; however, the root privileges can also be exposed to malware on a rooted Android system, and thus, it brings more security risks. In addition, the security apps scan installed apps for malware detection according to malware signature database. This approach demands frequent network communication and maintenance of up-to-date malware database. It requires extra users' data traffic and may fail to defend against zero-day malware. Researches on Android malware detection generally fall into 2 categories, ie, static analysis techniques and dynamic analysis techniques. Static analysis tries to cover all codes of an app to thoroughly explore possible malicious behaviors but fails to detect malware samples that can dynamically load malicious code at run time. Dynamic analysis can handle this type of samples; however, most of existing schemes inspect API calls and permission uses in an isolated way. As a matter of the fact, the combination of the sequence information of API calls and permission uses can represent app dynamic behavior attributes more accurately.

In this paper, we propose a dynamic behavior inspection and analysis framework for malicious behavior detection of Android apps. Our main work includes the following. (1) We customize an Android system to record API calls, permission uses, and some other runtime features. On this basis, we develop an automated app behavior inspection platform, and 13 825 malicious and benign apps are loaded and run automatically on the platform. (2) We model the recognition of malicious apps as a text classification problem and build a string subsequence kernel (SSK)-based Support Vector Machine (SVM) model to classify benign and malicious behaviors. This model can be used to perform offline malicious app recognition. (3) We extract apps' runtime features and further implement online malicious behavior detection using machine learning techniques. Of 390 malware samples, 71.8% instances are identified by running each app for only 18 minutes.

An early version of our work is presented in Ni et al.⁴ Based on the conference version, we introduce a new offline malicious behavior detection method, new experimental evaluation, and others in this extended version. The rest of this paper is organized as follows. In Section 2, we describe how to customize an Android system to record the behaviors of apps and automatically run plenty of apps on the customized system. In Section 3, we describe the basic idea of our approach and present in details the offline and online detection algorithm of malicious behavior respectively. In Section 4, we evaluate the effectiveness of our malicious behavior detection schemes. Finally, we discuss related work and conclude our work in Section 5 and Section 6, respectively.

2 | DYNAMIC BEHAVIOR INSPECTION OF ANDROID APPS

Since apps' dynamic behavior characteristics can be represented by the sequence of API calls and permission uses, we modify an Android system to monitor the API invocations and permission uses that are related to privacy information exposure and stealthy charges. Then we develop an automated app behavior inspection platform to load

the customized Android system to automatically install and start up apps, simulate user operations, inject SMS and phone-call events, and collect app behavior records.

2.1 | API call inspection

To monitor privacy-relevant API calls, we implement the taint tracking technique on the 4.4.2_r2 branch of AOSP according to TaintDroid.⁵ The taint tracking technique uses a 4-byte unsigned integer to describe the taint field. Each bit represents a type of privacy data. If multiple bits are set to 1, it indicates that this data is relevant to multiple types of privacy information.

As Figure 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, 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. We also modify the basic instructions in Dalvik VM to maintain the taints, since any further operations on the data including numerical calculation, truncation, concatenation, type conversion, and encryption are achieved by using basic instructions in Dalvik VM. 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.

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

Since API names may be changed in different system versions, we translate the recorded API invocations into permission checks as presented in one study.⁶ Because all the API calls for charges and retrieving privacy information can be mapped to corresponding permissions,⁷ it is easy to perform the translation as shown in Table A. 1.

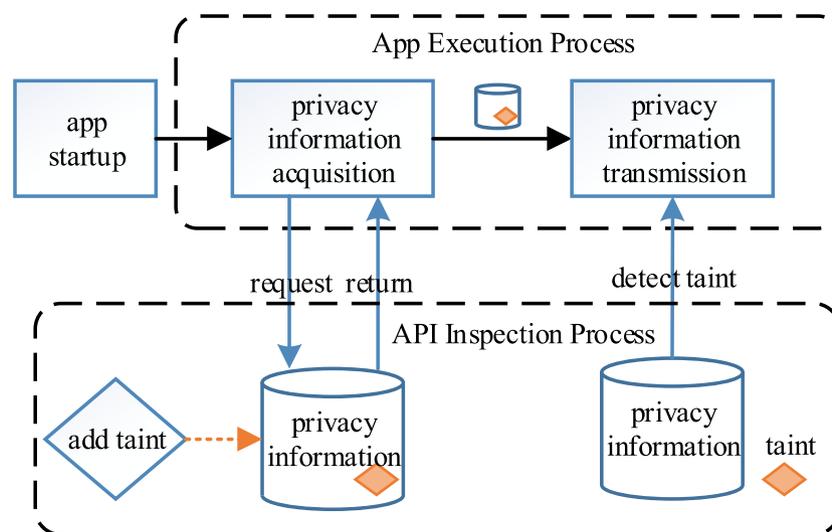


FIGURE 1 Taint tracking-based API call inspection

2.2 | 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 a sensitive API is invoked, the Android system will check whether the caller app obtains the corresponding permissions in the installation process. As shown in Figure 2, permission check in Android system is executed in 2 different system layers, ie, the framework layer and the kernel layer. We record the permission uses of apps by monitoring the permission checks in these 2 layers.

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 `PackageManagerService` in the framework layer to record the caller's User ID (UID), timestamp, and permission name. Other system permissions, including network, file system, Bluetooth, and system log, are checked in the kernel layer using the original file access control mechanism in the Linux system. Specifically, the mapping relationships between these system permissions and kernel Group IDs (GIDs) are described in the system file, ie, `/system/etc/permissions/platform.xml` and are checked in 2 functions, ie, `in_group_p(gid_t grp)` and `in_egroup_p(gid_t grp)`, in the source code, ie, `kernel/groups.c`. Consequently, we add a log function in these 2 functions to record the caller's UID, timestamp, and GID in

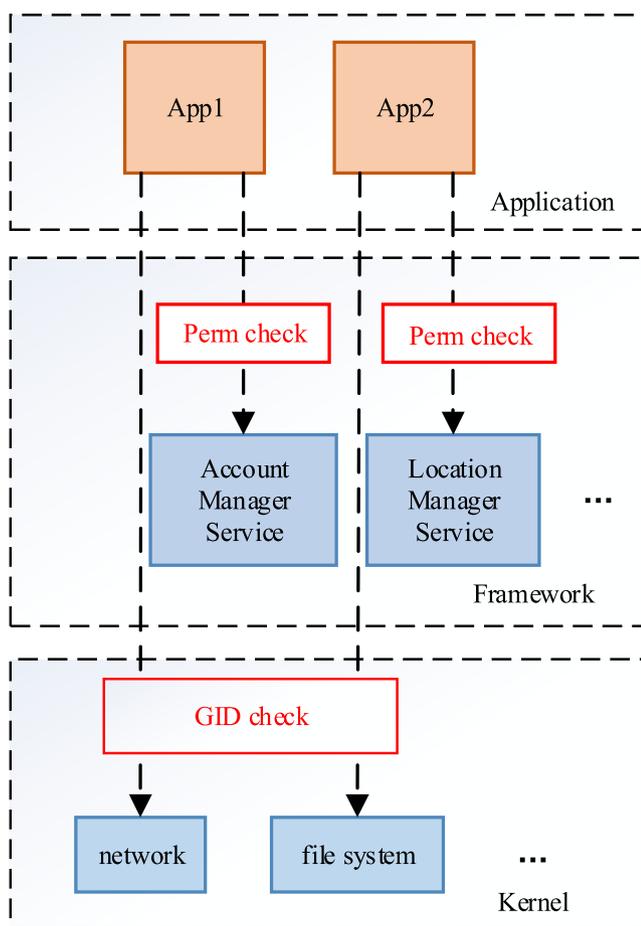


FIGURE 2 Permission use inspection in framework and kernel layer

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.

2.3 | Automated app behavior inspection platform

To automatically collect behavior records from large amounts of apps, we develop an automated app behavior inspection 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 apps reveal more behavioral characteristics on the platform, we simulate user operations, eg, click on the screen, press the hardware button, and slide the screen, using the application exerciser tool Monkey.⁸ Besides, we inject SMS and phone-call events through telnet commands so as to trigger potential interception functions of malwares.

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

- Step 1: Extract the package name and the launchable activity name from the manifest file, ie, `AndroidManifest.xml`.
- Step 2: Make a replica of the customized Android system and run it over an Android emulator.
- Step 3: Redirect the user-space and kernel-space system log 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 2 continuous operations.
- Step 6: After the monkey completes its operations, log on the emulator with telnet and send 2 events of receiving a phone call and 2 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 inspected.

Each app runs for about 18 minutes following the above steps. As some errors may occur during the inspection, the output information of steps 3 to 5 is parsed automatically. If any error is identified, our platform terminates the emulator and excludes the app. The automated app behavior inspection 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.

3 | BEHAVIOR SEQUENCE ANALYSIS BASED MALWARE DETECTION

We first introduce the basic idea of our approach. Then we present malicious behavior detection including offline and online detection schemes, respectively.

3.1 | Basic idea

Malware recognition is a binary classification problem. We classify a series of behavior sequences from different Android apps into 2 groups, ie, malicious and benign behaviors. We denote behavior sequences generated by various apps as $X = \{\vec{x}_1, \vec{x}_2, \dots, \vec{x}_m\}$, where \vec{x}_i is a behavior sequence from the i th app and m is the number of apps. Let $Y = \{y_1, y_2\}$ be the app category, where y_1 and y_2 indicate the malicious and the benign app, respectively. Then we aim to find an appropriate mapping from \vec{x}_i in X to y_j in Y , ie, $f(\vec{x}_i) = y_j$ ($i \in \{1, 2, \dots, m\}$, $j \in \{1, 2\}$), where f is the classification model. To this end, we carefully collect the training data and leverage an appropriate classification model to train a classifier. Then we can use the classifier to accurately classify the behavior sequences generated by Android apps into the right category.

To automatically perform large-scale malicious behavior inspection, we propose an offline malicious behavior detection method. We collect the labeled behavior sequences from both benign and malicious apps by using the automated app behavior inspection platform and convert each behavior sequence of apps into a string of letters. Then the malware recognition is modeled as a text classification problem. Thus, we use an SSK-based SVM classification model and train the model by using the labeled strings so as to detect the malicious behavior. However, the SSK-based model is not suitable for real-time detection, since it requires a complete behavior sequence per app as input data. In practice, users need to spend sufficient time using an app to derive a complete behavior sequence from the app for malicious behavior inspection purpose. Therefore, this method can only be used for large-scale offline malicious behavior detection rather than real-time detection.

As we all known, online detection systems should detect malicious behaviors as quickly as possible so as to block them in time. To fulfill this goal, we propose an online malicious behavior detection method that just requires subsequences of behavior. In fact, the malicious behavior sequence is a subsequence of the complete behavior sequence generated by malware. By manually examining the complete malicious behaviors of different malware in distinct categories, we extract the most effective behavior subsequences, referred to as *sensitive behavior sequence*, along with other dynamic behavior features and environmental features and use an appropriate classifier to differentiate the malicious behaviors from the benign ones.

We elaborate on these 2 malicious behavior detection methods in the following.

3.2 | Offline malicious behavior detection

We collect labeled behavior sequences from both benign and malicious apps by using the automated app behavior inspection platform. By having app behavior sequences labeled with malicious or benign, we use machine learning techniques to mine the features of malicious and benign behavior sequences and then build a classification model to classify unknown apps. In fact, the malicious behavior sequence is a subsequence of the complete behavior sequence generated by malware. If behavior subsequences of an

unknown app are similar to known malicious behavior sequences, we classify it as a malicious app.

Since the API calls can be mapped to the permission uses,⁷ we combine both the permissions of API calls and the permission uses in terms of their timestamps to construct a behavior sequence. Then we use alphabet characters to represent different permissions. In this way, a behavior sequence from an app is converted into a string. We choose an appropriate machine learning technique to classify the strings into 2 groups. Thus, the app behavior classification is modeled as a text classification problem.

We use string kernel-based SVM to solve the text classification problem. SVM is a supervised binary classification algorithm that can use a kernel function to map linearly nonseparable data to high-dimensional space, in which we can find a hyperplane to accurately divide the data into 2 parts. To address the text classification problem, string kernel function is one of the common kernel functions that is used to estimate the similarity of 2 strings. There are 3 common string kernel functions for text classification, including vector space model, n -gram kernel function, and SSK. Vector space model is a traditional technique that uses the word frequency for text classification. However, it loses all the word order information. In addition, n -gram kernel function merely exploits the continuous word order. Nevertheless, the malicious behavior subsequences are usually not contiguous in the entire behavior sequence. Therefore, we choose the SSK function to address our text classification problem as the noncontiguous subsequences information can be used.

There are 2 parameters in an SSK function. Denote by k the maximum length of all subsequences, where k is an integer and $k \geq 1$. λ is a decay factor ($0 < \lambda \leq 1$) used to deal with noncontiguous subsequences. The higher values of λ , the larger the interior gaps permitted in the subsequences. If the value of λ is close to 0, the common subsequence is almost contiguous; otherwise, the calculated value of the kernel function will be close to 0 if separated by other behaviors. Therefore, the SSK function can be used to simulate the n -gram kernel function by setting λ a value close to 0.

According to the above analysis, the malware recognition problem can be modeled as a binary classification problem using SSK-based SVM, which consists of 3 steps.

Step 1. Training set generation. After merging the API invocation and permission use records from the automated app behavior inspection platform, we label the records from malware as malicious and randomly obtain the same amount of behavior records from Google Play apps and label them as benign.

Step 2. SSK-based SVM model training and validation. On the basis of SSK function provided by Shogun,⁹ we use SVM to train the classification model and then validate it using 10-fold cross validation. Although the Shogun tool presets a variety of alphabets for special data classification (eg, 4-character DNA alphabet and RNA alphabet), none of these can be used for all APIs and permissions associated with privacy information and fee charging operations. To meet the needs of Android app

behavior classification, we add a new alphabet into the Shogun tool.

Step 3. Varying the parameters. We assign different values to the k and λ parameters of SSK function and then select appropriate values by evaluating the 10-fold cross validation results.

3.3 | Online malicious behavior detection

Since SSK-based malicious behavior detection technique requires a complete behavior data by running an app after a certain period of time, it cannot be used for real-time malicious behavior detection. To address this issue, we design and implement an online multimode sequence matching algorithm for online malware behavior detection. We first manually extract some sensitive sequences from the malware behavior sequences and then effectively perform the sequence matching by designing a novel trie tree-based data structure and multimode matching algorithm. We use the matching result as one of the features for the online malicious behavior detection.

3.3.1 | Feature extraction

We use dynamic behavioral and user-relevant features to effectively distinguish malicious and benign apps. Dynamic behavioral features include the sensitive permission combination uses, as well as sensitive behavior sequences. User-relevant features are used to indicate whether permission uses are caused by user operations.

Dynamic behavioral feature

Sensitive permission combination Since a malware requires a series of permissions to steal privacy information,¹⁰ we analyze extensive permission uses of both malicious and benign apps to extract the sensitive permission combinations by using association rule mining technique. Table 1 shows the list of sensitive permission combinations.

TABLE 1 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

However, these sensitive permissions can be used by benign apps as well. For example, a benign instant messaging app (eg, QQ) can require the READ_CONTACTS for friend recommendation and INTERNET permissions for communication. As a result, the combination of these 2 permission uses for malicious behavior detection can cause false alarms. To address this issue, we adopt sensitive behavior sequences to increase the detection accuracy and reduce the false alarms.

Sensitive behavior sequence We study the raw behavior sequences from 24 well-known categories of malware¹¹ and manually extract some subsequences from them as *sensitive behavior sequences* that can be effectively used for detecting each category of malware. Recall that we can derive a raw behavior sequence of an app by combining the permission uses and the permissions of API invocations for charges and retrieving privacy information in terms of the timestamp. We carefully inspect the raw behavior sequences of various malware from 24 different categories. Since the malicious and benign behavior subsequences are mixed in the raw behavior sequence, the malicious behavior subsequences are not continuous. Therefore, we manually extract 24 groups of sensitive behavior sequences that can be used to effectively detect the malware from each category as shown in Table A. 2. We use these subsequences to check if a raw behavior sequence contains these sensitive behavior sequences.

A perfect matching of sensitive behavior sequences leads to high false negative rate. Moreover, the malicious behavior should be detected as soon as possible so as to protect the privacy data of the users from malware. Therefore, we use the subsequence of the 24 sensitive behavior sequences as malicious behavior features. Denote the maximum length of subsequence as N , where $2 \leq N \leq 5$. Let M be the number of malicious behavior features. The value of M varies in terms of N . We evaluate the value of N in Section 4.

Online multimode sequence matching algorithm To discover sensitive behavior sequences from a continuous behavior sequence generated by an app in real time, we propose a novel online multimode sequence matching algorithm for comparing app behavior sequences with the 24 types of sensitive behavior subsequences.

We segment the real-time original behavior sequences by using a threshold of a delay interval between 2 continuous records. If the delay interval exceeds a threshold, ie, Δt , the current segmentation can be used for sequence matching and the following sequences are put in a new segmentation.

We design a trie tree-based data structure to store the sensitive behavior subsequences for our online sequence matching algorithm. Figure 3 illustrates our customized trie tree. All the nodes in the trie tree are labeled by indexes. In addition, the nodes store the original pointers pointing to child nodes and a flag indicating whether it is the end of a subsequence of sensitive behavior sequences. Moreover, a hash map is used to record the mapping between the node index and the memory address of each node in order to access the nodes directly. Similar to a traditional trie tree, a path from the root down to a node indicates that the current sequence fragment contains a subsequence of sensitive behavior sequences. The tree is constructed by the M subsequences of sensitive behavior sequences. Therefore, there are

Node number	Memory address
0	0x002684
1	0x002694
2	0x0026a4
3	0x0026b8
...	...

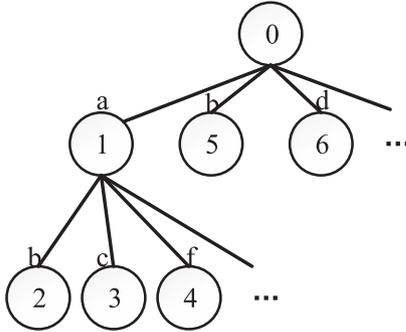


FIGURE 3 Data structure used in online multimode sequence matching

M leaf nodes with a flag that indicates the end of a subsequence of sensitive behavior sequences.

By using this trie tree, we can effectively perform the sequence matching between a segmentation of app behavior sequence and subsequences of sensitive behavior sequences. The online multimode sequence matching algorithm is shown in Table 2. The output of the algorithm is a feature vector with the length M . Each value in the vector indicates whether a subsequence of sensitive behavior sequences is matched by comparing with the current behavior sequence. If the current behavior sequence matches some

TABLE 2 Online multimode sequence matching algorithm

Algorithm: Online Multimode Sequence Matching

```

1. Input1: Behavior records (behavior: List<timestamp, permission>)
2. Input2: Malicious pattern tree(malSet: Set<List<permission>>)
3. Output: result records (results: List<matchNodeId>)
4. Trietree tt = new Trietree();
5. for sItem in malSet:
6.   tt.insert(sItem);
7. struct State {long timestamp, int nodeId};
8. Set stateSet = new Set(State);
9. for lItem in behavior:
10.  for sItem in stateSet:
11.   if lItem.timestamp - sItem.timestamp > MAX_DELAY:
12.    stateSet.erase(sItem);
13.   else:
14.    next = tt.nextState(sItem.nodeId, lItem.permission);
15.    State s = new State(lItem.timestamp, next);
16.    stateSet.erase(sItem);
17.    stateSet.insert(s);
18. State s = new State(lItem.timestamp, 0);
19. next = tt.nextState(0, lItem.permission);
20. if next != 0:
21.  stateSet.insert(new State(lItem.timestamp, next));
22. for sItem in stateSet:
23.  if tt.getNodeAddr(sItem.nodeId)->isWord:
24.   next = sItem.nodeId;
25.  else:
26.   next = -1;
27.  stateSet.erase(sItem);
28.  results.add(next);
29. return results;

```

subsequences of sensitive behavior sequences, the corresponding values are set to 1. Otherwise, the values are set to 0.

User-relevant feature

The sensitive behavior sequences can be generated not only by malware but also by users, since apps used by users can cause fee charging or privacy information acquisition and transmission. To reduce the false positive of malicious behavior detection, we inspect current user operations over the mobile device screen, referred to as user-relevant feature, to determine whether these sensitive behavior subsequences are created by users or malware.

To determine the behavior sequences generated by users, we modify the source code of Android to inspect the user operations. The user operations on our platform are simulated by using the Monkey tool. We explore the source code of both Android system and Monkey to record the simulated user operations. In particular, we find that the 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 user operation events and dispatches them to corresponding app user interface windows in the foreground. As the InputEventReceiver class used by WindowManagerService invokes dispatchInputEvent() method in this procedure, we add a log function in the dispatchInputEvent() method to record the user operation events sent from the Monkey tool.

3.3.2 | Online malicious behavior detection scheme

We preprocess the behavior sequence by using the M features of sensitive behavior subsequences and then add 2 features including the inspection of sensitive permission combination and the user-relevant feature. We check whether the permissions in the segmentation contain a sensitive permission combination. In addition, the time interval between the current permission use and last user operation is used as user-relevant feature. Hence, the feature vector includes these $M + 2$ features for training and classification.

We use Naïve Bayes algorithm and SVM algorithm to classify app behaviors and select an appropriate one by comparing the various metrics, eg, accuracy, false positive rate, etc. We also evaluate the effectiveness of N . If any behavior sequence of an app is classified as malicious by the model, then the app is considered as a malicious one.

4 | EVALUATION

4.1 | Data collection

We design a crawler to download apps automatically from various app markets. The training malware app dataset obtained from Android Malware Genome Project¹¹ contains 1243 malicious apps that are categorized into 34 groups. The training benign app dataset contains 12 582 apps downloaded from different top 500 popular apps of various categories in Google Play Market. In addition, we download 14 733 apps from different top lists of various categories in Anruan Market as the testing app dataset.

To recording all of the app behavior sequences, we take 60 days to run these apps on our automated app behavior inspection platform.

4.1.1 | Data collection for offline malicious behavior detection

We remove the apps that cannot run on our platform or generate insufficient behavior sequences. Through observations on our dataset, we find that the behavior sequences of length less than 4 are insufficient to present apps' behavior. Then we get 628 records from malware, 4351 records from Google Play Market apps, and 3934 records from Anruan Market apps.

The training set consists of 628 records generated by malware and 635 records that are randomly selected from Google Play Market, while the testing set is composed of 3934 records from Anruan Market apps.

4.1.2 | Data collection for online malicious behavior detection

We preprocess the data collected by the automated app behavior inspection platform and manually extract 24 malicious sequence patterns from 24 categories, which involve 390 malicious samples.

The training set is generated as follows. The behavior records from malware samples that containing subsequences of sensitive behavior sequence are marked as malicious, while all the other records are marked as benign. In addition, we randomly select a similar number of behavior records from Google Play Market apps and marked them all as benign. In this way, there are 12 0641 records in total, 117 741 benign ones, and 2900 malicious ones.

Behavior data of 3917 apps from Anruan Market are used for open-world analysis.

4.2 | Performance metrics

We adopt the standard classification metrics to evaluate the experiment results, ie, accuracy, false positive rate (FPR), false negative rate (FNR), recall rate, precision, and F-measure. We denote by $C_{m \rightarrow m}$ the number of correctly classified malicious behavior records and by $C_{b \rightarrow b}$ the number of correctly classified benign behavior records. Denote the number of incorrectly classified malicious behavior records is denoted as $C_{m \rightarrow b}$, and the number of incorrectly classified benign behavior records is denoted as $C_{b \rightarrow m}$. Then we have $C_{all} = C_{m \rightarrow m} + C_{b \rightarrow b} + C_{m \rightarrow b} + C_{b \rightarrow m}$.

$$Accuracy = \frac{C_{m \rightarrow m} + C_{b \rightarrow b}}{C_{all}} \quad (1)$$

$$FPR = \frac{C_{b \rightarrow m}}{C_{b \rightarrow b} + C_{b \rightarrow m}} \quad (2)$$

$$FNR = \frac{C_{m \rightarrow b}}{C_{m \rightarrow m} + C_{m \rightarrow b}} \quad (3)$$

$$Recall = \frac{C_{m \rightarrow m}}{C_{m \rightarrow m} + C_{m \rightarrow b}} \quad (4)$$

$$Precision = \frac{C_{m \rightarrow m}}{C_{m \rightarrow m} + C_{b \rightarrow m}} \quad (5)$$

$$F\text{-measure} = \frac{(\beta^2 + 1)PR}{\beta^2 P + R} \quad (6)$$

In Equation 6, P is the precision in Equation 5 and R is the recall rate in Equation 4. If $\beta = 1$, Equation 6 is so called $F1\text{-measure}$, which weights recall rate and precision equally. If $\beta > 1$, Equation 6 emphasizes recall rate more than precision. Otherwise, if $0 < \beta < 1$, it weights precision more than recall rate.

4.3 | Evaluation results

4.3.1 | Offline malicious behavior detection experiment results

We evaluate the accuracy, FPR, FNR, and recall rate by performing 10-fold cross validation with different values of k and λ of SSK function, as shown in Figures 4–7. It can be observed in Figure 4 that the accuracy increases in terms of λ and reaches a peak when λ is 1

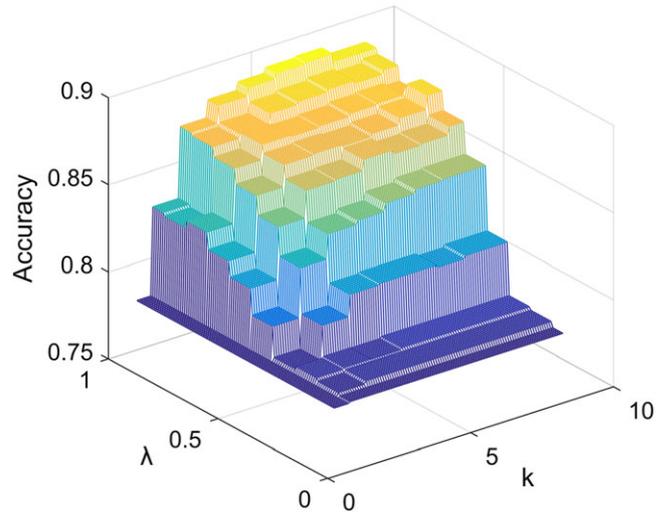


FIGURE 4 Accuracy of offline app classification

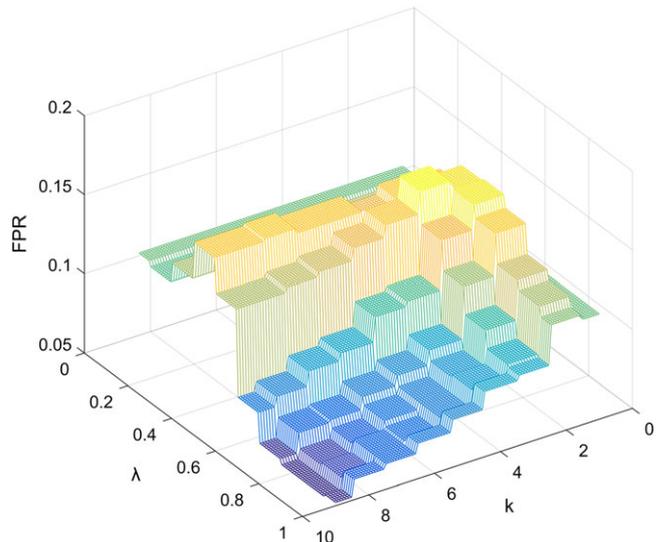


FIGURE 5 False positive rate (FPR) of offline app classification

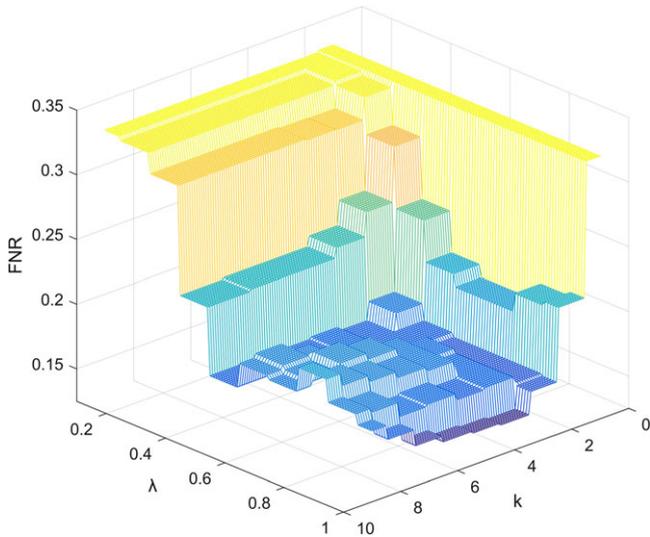


FIGURE 6 False negative rate (FNR) of offline app classification

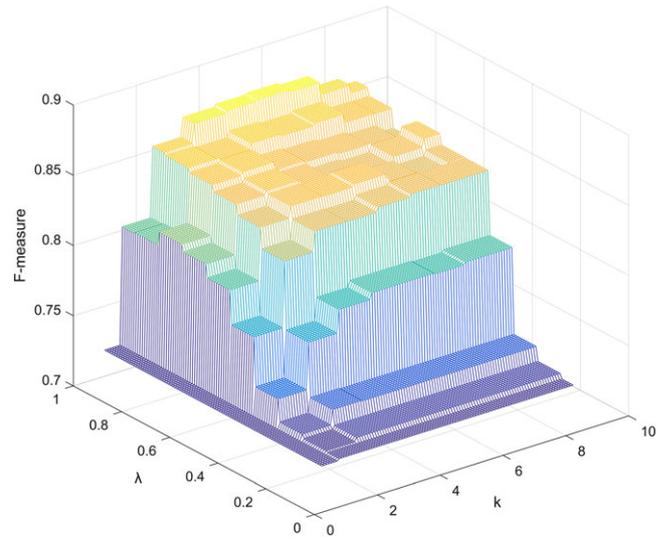


FIGURE 8 F-measure evaluation of offline app classification

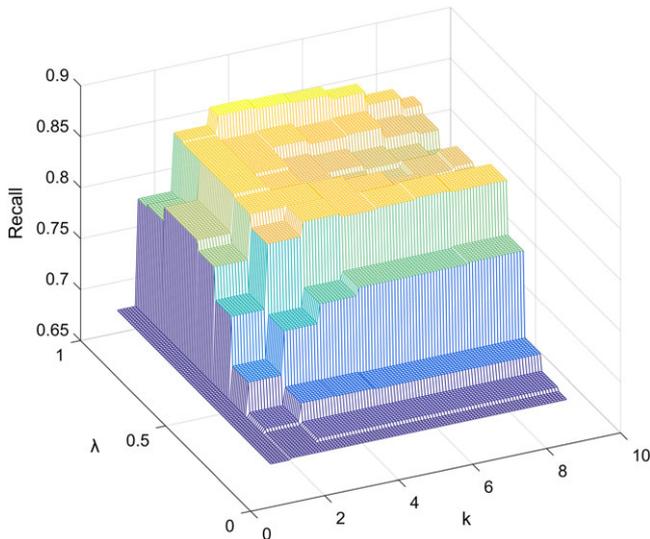


FIGURE 7 Recall rate of offline app classification

a curve of accuracy, FPR, FNR, and recall rate as shown in Figure 9 and a curve of F-measure in terms of k is shown in Figure 10.

As we can see from Figures 9, 10, F-measure reaches the maximum at $k = 6$ and $\lambda = 1$. The accuracy, FPR, and recall rate are 88.8%, 7.2%, and 84.9%, respectively, which shows that 84.9%

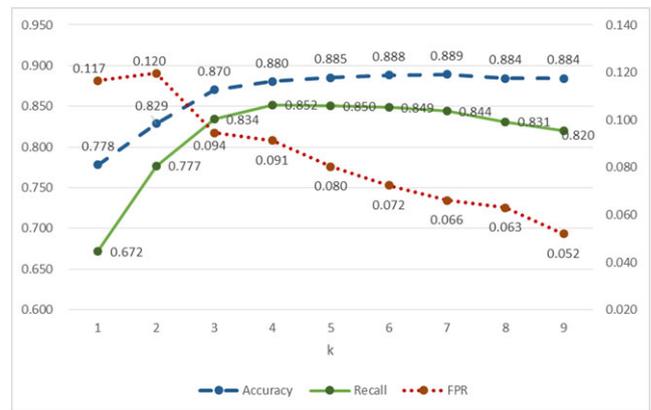


FIGURE 9 Values of different metrics for offline app classification when $\lambda = 1$

and k is in the range of 4 to 8. Figure 5 shows that FPR decreases with the increase of k , except when $k = 1$ the FPR is slightly lower than $k = 2$, but still a high value. When $k = 9$ and λ is in the range of 0.7 to 1, it reaches its minimum. Figure 6 depicts that FNR decreases with the increase of λ with only an exception when $\lambda = 0.4$ and reaches its minimum when λ is 1 and k is in the range of 4 to 7. As seen from Figure 7, recall rate increases first and then decreases in terms of both λ and k and reaches a peak when $\lambda = 1$ and k is in the range of 4 to 5.

To make the model perform well when considering accuracy, FPR, FNR, and recall rate in the whole, we tune the values of k and λ by using F-measure as the metric.

We use F-measure to evaluate our method by changing the value of k and λ . To increase the recall rate as much as possible, we sacrifice the precision so as to keep users from malware. Therefore, we weight recall rate more than precision by using $\beta=1.5$ in Equation 6. Figure 8 illustrates the relationship among F-measure, k , and λ . It is observed that F-measure reaches a maximum where λ is 1 and k is in the range of 3 to 7. To choose a reasonable value of k , we set λ to 1 and then plot

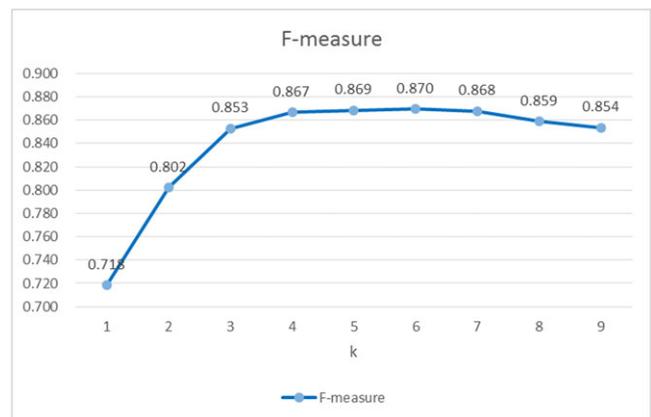


FIGURE 10 F-measure evaluation of offline app classification when $\lambda = 1$

malware samples can be detected using our offline malicious behavior detection method.

We compare the performance of SSK function with the n -gram kernel function. As presented in Section 3, we can use SSK to simulate n -gram kernel function when λ approaches 0. Therefore, we use $\lambda = 1$ and $\lambda = 0.01$, respectively, to roughly compare the performance of SSK and n -gram kernel at $k = 6$. The results are shown in Figure 11. We can see from Figure 11 that all the 3 metrics of SSK function outperforms those of n -gram kernel. It demonstrates that most malicious operations of Android apps are not continuous but isolated by normal operations. This result is consistent with our observations, since a majority of malicious apps are generated by injecting a piece of malicious code into the benign apps.

To validate the offline malicious behavior detection scheme, we apply the classification model to test all the apps obtained from Anruan Market. 1045 out of 3934 apps, ie, around 26.6% of all the apps, are classified as malicious apps. We randomly select 100 samples from these 1045 apps and analyze the behavior records manually. As we can see from Table 3, most of these apps obtain the phone identifiers, including IMEI and IMSI. In addition, we find that 20 detected apps do not carry out obvious malicious behaviors, which we consider as false alarms.

The behavior sequences of a malware collected by our automated app behavior inspection platform may contain both malicious and benign behaviors. However, during the classification training phrase,

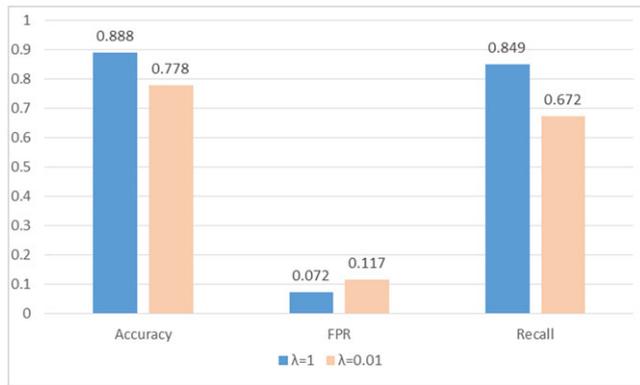


FIGURE 11 Comparison of different metrics for offline app classification when $\lambda = 1$ and 0.01

TABLE 3 Manual analysis of alarmed apps by offline detection algorithm

Malicious Behavior	Number of Apps With This Behavior
Steal phone records	1
Steal location	21
Steal messages	3
Steal phone identifiers	62
Steal user account	2
Steal browse history	2
Steal external storage	14
None	20

The sum of the numbers in the table is more than 100 because one app can have multiple types of malicious behavior

both malicious and benign behavior sequences from malware are labeled as malicious ones. It can cause the classification model overfitting and lead to false positive errors in the testing phrase. Consequently, it is consistent with the experiment result that 20 false alarms occurred.

4.3.2 | Online malicious behavior detection experiment results

In the online malicious behavior detection scheme, we evaluate the performance of both SVM and Naïve Bayes algorithms. We use different number of subsequences of sensitive behavior sequence by using distinct value of N . Figure 12 shows how M is influenced by the value of N . We use SVM and Naïve Bayes algorithms to classify the online experiment dataset. Ten-fold cross validation is applied to validate the effectiveness of these 2 algorithms, and the results are shown in Figure 13 and Figure 14.

Figure 13 shows that SVM can reach an accuracy higher than 98%, with a false negative rate less than 3%. Figure 14 illustrates that false negative rate of Naïve Bayesian Classifier is more than 44% that is much higher than that of SVM. Therefore, we use SVM for detecting malicious behavior records. In Figure 13, it can be observed that the model reaches an accuracy of 99.0% with an FPR of 1.0% and an FNR of 2.3% at $N = 5$.

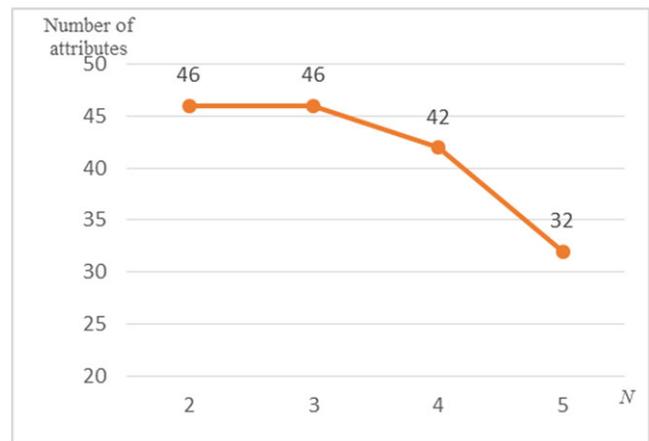


FIGURE 12 Influence of N 's value on M

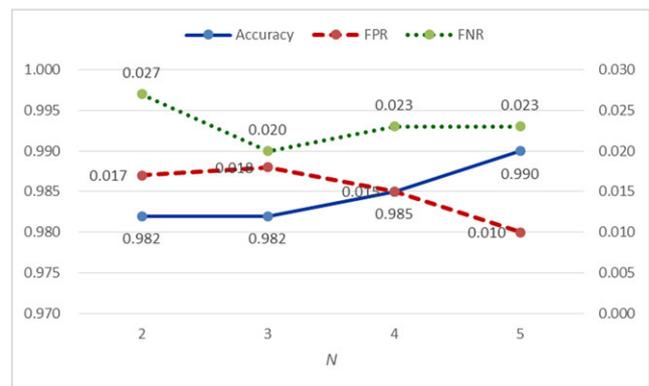


FIGURE 13 Ten-fold cross-validation results of SVM-based online malicious behavior detection

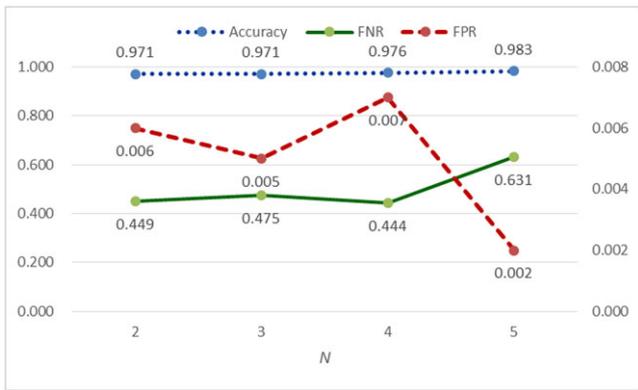


FIGURE 14 Ten-fold cross-validation results of Naïve Bayes-based online malicious behavior detection

We mark an app as malicious if at least one of its behavior records is classified as malicious. Two hundred eighty malware samples out of the 390 total samples, ie, 71.8%, are detected. By manually checking the behavior records from malware samples that are classified as benign, we do not discover malicious behavior. It results from that the undetected malware samples do not perform malicious behaviors during our inspection. Probably, these malware samples are designed to perform malicious activities by meeting some triggering conditions, such as adequate running time, certain operations on the apps, connection to a remote server, etc.

We perform an open-world malware detection on apps from the Anruan Market. As a result, 952 out of 3917 apps, ie, 24.3%, are detected as malicious ones. We randomly select 120 apps from malicious apps and manually check the behavior records. We find that most of them perform certain malicious behavior as shown in Table 4. A majority 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 24 apps that do not perform malicious behavior but still are marked as malicious ones. Our further analysis reveals that these 24 apps generate 2167 segmentations of behavior records, and only 75 behavior records, ie, 3.5%, are misclassified as malicious ones. It demonstrates that our scheme achieves a very low FPR at the behavioral records level.

TABLE 4 Manual analysis of alarmed apps by online detection algorithm

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

5 | RELATED WORK

Techniques for detecting Android malware can be categorized into 3 types, including static analysis techniques, dynamic analysis techniques, and hybrid analysis techniques.

5.1 | Static malware analysis techniques

Static analysis is commonly used for the security analysis of software. According to the distinct analysis targets, static analysis of Android apps can be categorized into installation package analysis, bytecode analysis, and source code analysis.

Installation package analysis was first proposed by Enck et al.¹² They analyzed the requested permissions for possible malicious functions by dividing malicious functions into corresponding permissions. Zhou et al.¹³ collected a large set of malware samples and analyzed the permission uses in depth. Then, many researchers used this dataset for malware detection studies. MAST¹⁴ collected information of permission requests, whether the app contains native code, self-startup behavior, etc, to rank the risks of apps and decide which app requires a deeper scan. Similarly, DREBIN¹⁵ collected more available information 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 to determine whether there exists malicious code. ComDroid¹⁶ analyzed inter-application communication to detect malicious behavior such as broadcast interception and service hijacking. However, binary code or byte code analysis is a tough work. Thus, more researchers decompiled the byte code before scanning. Aafer et al.¹⁷ investigated the API call features after decompiling, including API name and API parameters, but this method can be easily bypassed by malware developers. DroidSIFT¹⁸ constructed dependency graphs of API calls using a semantics-based approach to avoid detection evasion, which achieved lower FNR and false alarm rate.

In addition to malware analysis, static analysis can also be used to detect vulnerabilities in apps that can be utilized by malware, such as interapplication communication vulnerabilities,¹⁶ component hijacking,¹⁹ and capability leaks.²⁰ Static analysis is generally suitable for offline analysis and can be used for market-scale apps analysis, but this approach is ignorant of runtime context and can be easily evaded by malware developers. Thus, some researchers proposed dynamic analysis techniques for detecting runtime anomalies. Dynamic analysis is usually more suitable for being deployed on user devices and can intercept malicious behavior timely.

5.2 | Dynamic malware analysis techniques

The most typical work of dynamic analysis is TaintDroid⁵ proposed by Enck et al, which detected potential privacy information leakage by adding taint information into privacy data. It can detect privacy information leakage effectively at runtime but failed to track leakages caused by native code. Furthermore, when deployed on a smartphone, it demanded for nonnegligible CPU usage and energy consumption. VetDroid²¹ is another taint tracking-based privacy information

leakage detection system, which monitored both explicit permission uses and implicit permission uses to model app behaviors more accurately.

Dynamic analysis can also monitor system API calls to analyze an app's behavior on the upper layer. CopperDroid^{22,23} and DroidScope²⁴ put android apps in a sandbox to inspect the interactions between the sandbox and the external system. The invocations of framework APIs, JNI methods, system calls in the operation system were utilized to model the application behavior. However, it can only be used for offline analysis instead of being deployed on a smartphone. Shabtai et al²⁵ focused on side-channel information generated by the network traffic during an app's runtime. The traffic features were extracted and sent to a remote server, and then the app was classified using machine learning algorithms to detect potential malicious behaviors. AppsPlayground²⁶ combined a set of dynamic analysis techniques, including taint tracking, API inspection, kernel system call inspection, and multiple code exploration techniques to analyze privacy leakages and malicious functions during apps' runtime.

5.3 | Hybrid malware analysis techniques

Some researchers combined static and dynamic analysis techniques to gain both the former's efficiency and flexibility and latter's accuracy. DroidRanger²⁷ filtered suspected malicious apps by analyzing permission requests and byte code then further explored malicious behavior using dynamic system call inspection. It was also effective for finding out zero-day malware from market-scale app samples. DroidDetector²⁸ associated the features from the static analysis with features from dynamic analysis of Android apps and detected malware using deep learning techniques. The features used fell into 3 types: required permissions, sensitive APIs, and dynamic behaviors. This machine learning-based method performed well with the variety of Android malware.

6 | CONCLUSION AND FUTURE WORK

In this paper, we proposed an analysis framework for monitoring, recording, and analyzing app behavior, to detect malicious behavior of Android apps offline and online. We applied SSK-based SVM to recognize Android malware offline and manually extracted sensitive behavior sequences for online malicious behavior detection. Our schemes have a high detection rate for detecting malicious behavior, and a relatively high proportion of malware samples are detected in the dataset used. To detect malware more accurately, we will conduct further studies on how to construct a runtime environment in the emulator that better mimics the environment of a real smartphone to trigger more behaviors of Android apps.

ACKNOWLEDGEMENTS

This work is supported by National Natural Science Foundation of China under Grants 61572130, 61320106007, 61502100, 61532013, and 61402104; Jiangsu Provincial Natural Science Foundation under Grants BK20140648 and BK20150637; 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. International Data Corporation. Worldwide smartphone market will see the first single-digit growth year on record. <http://www.idc.com/getdoc.jsp?containerId=prUS40664915>, Dec 2015.
2. LBE Tech. <http://www.lbesec.com/#/products/2>.
3. 360 Security. <http://www.360securityapps.com/en-us>.
4. Ni Z, Yang M, Ling M, Wu JN, and Luo J. "Real-time detection of malicious behavior in Android apps". In Proceedings of the 4th IEEE International Conference on Advanced Cloud and Big Data (CBD), Chengdu, Sichuan, China, August 13-16, 2016.
5. Enck W, Gilbert P, Chun B-G, et al. TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones. In Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI), Berkeley, CA, USA, 2010, pp. 1-6.
6. Manifest.permission. <https://developer.android.com/reference/android/Manifest.permission.html>.
7. Au K, Zhou Y, Huang Z, and Lie D. Pscout: analyzing the android permission specification. In Proceedings of the 19th ACM conference on Computer and communications security (CCS), 2012: 217-228.
8. UI/Application Exerciser Monkey. <https://developer.android.com/studio/test/monkey.html>.
9. Shogun—a large scale machine learning toolbox. http://www.shogun-toolbox.org/page/about/project_description.
10. Enck W, Ongtang M, McDaniel P. On lightweight mobile phone application certification. In Proceedings of the 16th ACM conference on Computer and communications security. ACM, 2009: 235-245.
11. Android malware genome project. www.malgenomeproject.org.
12. Enck W, Ongtang M, and McDaniel P. Mitigating Android software misuse before it happens. 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.
13. Zhou Y, and Jiang X. Dissecting Android malware: characterization and evolution. In Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P), 20-23 May 2012.
14. Chakradeo S, Reaves B, Traynor P, and Enck W. MAST: triage for market-scale mobile malware analysis. In Proceedings of the 6th ACM conference on Security and privacy in wireless and mobile networks (WiSec), Budapest, Hungary, 2013, pp. 13-24.
15. Arp D, Spreitzenbarth M, Hübner M, Gascon H, and Rieck K. DREBIN: effective and explainable detection of Android malware in your pocket. In Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS), February 2014.
16. Chin E, Felt A P, Greenwood K, and Wagner D. Analyzing inter-application communication in Android. In Proceedings of the 9th international conference on Mobile systems, applications, and services (MobiSys). ACM, 2011.
17. Aafer Y, Du W, and Yin H. DroidAPIMiner: mining API-level features for robust malware detection in Android. 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.
18. Zhang M, Duan Y, Yin H, and Zhao Z. Semantics-aware Android malware classification using weighted contextual API dependency graphs. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS), Scottsdale, Arizona, USA, 2014, pp. 1105-1116.

19. Lu L, Li Z, Wu Z, Lee W, and Jiang G. Chex: statically vetting android apps for component hijacking vulnerabilities. In Proceedings of the 19th ACM conference on Computer and communications security (CCS), 2012.
20. Chan P P, Hui L C, and Yiu S M. Droidchecker: analyzing android applications for capability leak. In Proceedings of the 5th ACM conference on Security and Privacy in Wireless and Mobile Networks (WiSec), 2012.
21. Zhang Y, Yang M, Xu B, et al. Vetting undesirable behaviors in android apps with permission use analysis. In Proceedings of the 2013 ACM SIGSAC conference on Computer and Communications Security (CCS), Berlin, Germany, 2013, pp. 611-622.
22. Reina A, Fattori A, and Cavallaro L. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In Proceedings of the 6th European Workshop on Systems Security (EuroSec), Prague, Czech Republic, April, 2013.
23. Tam K, Khan S J, Fattori A, Cavallaro L. CopperDroid: Automatic reconstruction of Android malware behaviors. In Proceedings of the Network and Distributed System Security Symposium (NDSS), 2015.
24. Yan L K, Yin H. Droidscape: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In Proceedings of the 21st USENIX Security Symposium (USENIX Security). 2012: 569-584.
25. Shabtai A, Tenenboim-Chekina L, Mimran D, Rokach L, Shapira B, and Elovici Y. Mobile malware detection through analysis of deviations in application network behavior. *Comput Secur.* 2014;43:1-18.
26. Rastogi V, Chen Y, and Enck W. AppsPlayground: automatic security analysis of smartphone applications. In Proceedings of the 3rd ACM conference on Data and application security and privacy (CODASPY), San Antonio, Texas, USA, 2013, pp. 209-220.
27. Zhou Y, Wang Z, Zhou W, and Jiang X. Hey, you, get off of my market: detecting malicious apps in official and alternative Android markets. In Proceedings of the 19th Network & Distributed System Security Symposium (NDSS), Hilton San Diego Resort & Spa, 2012.
28. Yuan Z, Lu Y, Xue Y. Droiddetector: Android malware characterization and detection using deep learning. *Tsinghua Sci Technol.* 2016;21(1):114-123.

How to cite this article: Yang M, Wang S, Ling Z, Liu Y, Ni Z. Detection of malicious behavior in android apps through API calls and permission uses analysis. *Concurrency Computat: Pract Exper.* 2017;29:e4172. <https://doi.org/10.1002/cpe.4172>

APPENDIX

A. 1 | Framework layer API calls and corresponding permissions

Framework Layer APIs	Permissions
android.content.ContentResolver.query()	android.permission.READ_CONTACTS, com.android.browser.permission.READ_HISTORY_BOOKMARKS, android.permission.READ_SMS
android.hardware.Camera.EventHandler.handleMessage()	android.permission.CAMERA
android.hardware.SystemSensorManager.SensorEventQueue.dispatchSensorEvent()	android.permission.WAKE_LOCK
android.media.AudioRecord.read()	android.permission.RECORD_AUDIO
android.media.MediaRecorder.prepare()	
com.android.server.accounts.AccountManagerService.validateAccountsInternal()	android.permission.GET_ACCOUNTS
com.android.server.accounts.AccountManagerService.readPasswordInternal()	
com.android.server.accounts.AccountManagerService.invalidateAuthTokenLocked()	
com.android.server.accounts.AccountManagerService.getAccountLocked()	
com.android.server.accounts.AccountManagerService.getExtrasLocked()	
com.android.server.accounts.AccountManagerService.dumpUser()	
com.android.server.accounts.AccountManagerService.insertAccountIntoCacheLocked()	
com.android.server.accounts.AccountManagerService.readUserDataForAccountFromDatabaseLocked()	
com.android.server.accounts.AccountManagerService.readAutoTokensForAccountFromDatabaseLocked()	
com.android.server.location.GpsLocationProvider.reportLocation()	android.permission.ACCESS_FINE_LOCATION
com.android.server.LocationManagerService.handleLocationChanged()	
com.android.internal.telephony.gsm.GSMPhone.handleMessage()	android.permission.READ_PHONE_STATE
com.android.internal.telephony.uicc.SIMRecords.setMsisdnNumber()	
com.android.internal.telephony.uicc.SIMRecords.handleMessage()	

A. 2 | Sensitive behavior sequence

No.	Malware Category	Sensitive Behavior Sequence
S ₁	AnserverBot	ACCESS_WIFI_STATE CHANGE_WIFI_STATE READ_PHONE_STATE ACCESS_NETWORK_STATE INTERNET
S ₂	Bgserv	ACCESS_WIFI_STATE CHANGE_WIFI_STATE READ_PHONE_STATE INTERNET
S ₃	DroidCoupon	READ_PHONE_STATE ACCESS_NETWORK_STATE READ_PHONE_STATE ACCESS_NETWORK_STATE
S ₄	DroidKungFu1	READ_EXTERNAL_STORAGE INTERNET READ_PHONE_STATE ACCESS_NETWORK_STATE INTERNET
S ₅	DroidKungFu3	ACCESS_NETWORK_STATE READ_PHONE_STATE ACCESS_NETWORK_STATE INTERNET
S ₆	DroidKungFu4	ACCESS_WIFI_STATE CHANGE_WIFI_STATE ACCESS_COARSE_LOCATION WRITE_SMS READ_SMS READ_PHONE_STATE ACCESS_NETWORK_STATE READ_PHONE_STATE
S ₇	DroidKungFuSapp	ACCESS_NETWORK_STATE READ_PHONE_STATE READ_EXTERNAL_STORAGE INTERNET
S ₈	DroidKungFuUpdate	READ_PHONE_STATE ACCESS_NETWORK_STATE ACCESS_ALL_EXTERNAL_STORAGE READ_EXTERNAL_STORAGE INTERNET
S ₉	Endofday	SEND_SMS SEND_SMS_NO_CONFIRMATION
S ₁₀	Geinimi	READ_PHONE_STATE ACCESS_FINE_LOCATION
S ₁₁	GGTracker	READ_PHONE_STATE INTERNET READ_SMS
S ₁₂	GingerMaster	INTERNET VIBRATE READ_PHONE_STATE INTERNET
S ₁₃	GoldDream	INTERNET READ_PHONE_STATE ACCESS_FINE_LOCATION
S ₁₄	Gone60	READ_HISTORY_BOOKMARKS READ_CALL_LOG READ_SMS READ_CONTACTS INTERNET
S ₁₅	HippoSMS	READ_SMS WRITE_SMS SEND_SMS SEND_SMS_NO_CONFIRMATION

(Continued)

No.	Malware Category	Sensitive Behavior Sequence
S ₁₆	jsMSHider	INSTALL_PACKAGES READ_PHONE_STATE ACCESS_FINE_LOCATION ACCESS_NETWORK_STATE
S ₁₇	NickySpy	READ_PHONE_STATE ACCESS_FINE_LOCATION READ_PHONE_STATE INTERNET
S ₁₈	Pjapps	READ_PHONE_STATE RECEIVE_SMS INTERNET
S ₁₉	RogueLemon	READ_PHONE_STATE READ_EXTERNAL_STORAGE INTERNET
S ₂₀	RogueSPush	READ_PHONE_STATE ACCESS_NETWORK_STATE INTERNET WRITE_EXTERNAL_STORAGE ACCESS_WIFI_STATE
S ₂₁	SndApps	GET_ACCOUNTS
S ₂₂	YZHC	READ_HISTORY_BOOKMARKS READ_SMS READ_EXTERNAL_STORAGE READ_CALL_LOG
S ₂₃	zHash	READ_PHONE_STATE READ_HISTORY_BOOKMARKS
S ₂₄	Zsone	INTERNET ACCESS_COARSE_LOCATION ACCESS_FINE_LOCATION SEND_SMS SEND_SMS_NO_CONFIRMATION

(Continues)