Cluster Comput
DOI 10.1007/s10586-016-0703-5

DroidWard: An Effective Dynamic Analysis Method for Vetting Android Applications

Yubin Yang[1] · **Zongtao Wei**[2] · **Yong Xu**[1] · **Haiwu He**[3] · **Wei Wang**[2]

Received: 6 September 2016 / Revised: 30 October 2016 / Accepted: 21 November 2016

Abstract As the number of Android malicious applications has explosively increased, effectively vetting Android applications (apps) has become an emerging issue. Traditional static analysis is ineffective for vetting apps whose code have been obfuscated or encrypted. Dynamic analysis is suitable to deal with the obfuscation and encryption of codes. However, existing dynamic analysis methods cannot effectively vet the applications, as a limited number of dynamic features have been explored from apps that have become increasingly sophisticated. In this work, we propose an effective dynamic analysis method called DroidWard in the aim to extract most relevant and effective features to characterize malicious behavior and to improve the detection accuracy of malicious apps. In addition to using the existing 9 features, DroidWard extracts 6 novel types of effective features from apps through dynamic analysis. DroidWard runs apps, extracts features and identifies benign and malicious apps with Support Vector Machine (SVM), Decision Tree (DTree) and Random Forest. 666 Android apps are used in the experiments and the evaluation results show that DroidWard correctly classifies 98.54% of malicious apps with 1.55% of false positives. Compared to existing work, DroidWard improves the TPR with 16.07% and suppresses the FPR with 1.31% with SVM, indicating that it is more effective than existing methods.

Keywords Android security · Malware analysis · Malware detection · Dynamic analysis

1 Introduction

Android platform has dominated the markets of mobile devices in recent year. The number of Android applications (apps) has seen a massive surge. Meanwhile, Android platform has also become the primary target of attackers because of its huge market share and open source characteristics. It was indicated that the number of malicious apps in the third party Android app market accounted for more than 5–8% [1], while a large number of malicious apps have been found [2,3] in the official app market, Google Play. On the one hand, although Google has made great efforts to enhance the security of Android ecosystem [3], its security issue remains and is still serious. On the other hand, the major anti-virus software on the market is not effective to secure Android system [4]. Zhou et al [5] indicated that four representative mobile anti-virus softwares, namely, AVG, Lookout, Norton and TrendMicro, were only able to detect 79.6 and 20.2% of the malicious applications (malapps), in the best or the worst case respectively. In addition, in order to escape the detection, a large number of malicious apps use a series of self-protection deformation technology, such as the well-known obfuscation,

✉ Yubin Yang
ronald_yang@126.com

Zongtao Wei
weizongtao@bjtu.edu.cn

Yong Xu
yxu@scut.edu.cn

Haiwu He
haiwuhe@cstnet.cn

Wei Wang
wangwei1@bjtu.edu.cn

1 School of Computer Science & Engineering, South China University of Technology, 510641 Guangzhou, China

2 School of Computer and Information Technology, Beijing Jiaotong University, 100044 Beijing, China

3 Computer Network Information Center, Chinese Academy of Sciences, 100190 Beijing, China

bytecode encryption, reflective code and native code execution. Traditional static analysis methods are not effective for the detection of obfuscated or bytecode encrypted apps. Dynamic analysis is performed by executing apps on a real or virtual processor. It has become a major technique because of its capability of the detection of obfuscated or bytecode encrypted apps. However, existing dynamic analysis methods only used a few dynamic features extracted from apps and thus fail to achieve the desired detection performance [6,7]. Therefore, exploring more effective and relevant features from apps with dynamic analysis so as to improve the detection accuracy of malapps has become an important issue.

In this work, we focus on vetting Android apps with dynamic analysis. Through studying and analyzing the behaviors and the log information of benign apps and malapps in the simulator with dynamic analysis, we explore 6 kinds of novel features. Combining with 9 kinds of existing features [7–11], we thus have a total of 15 kinds of features that can be extracted from each app. By employing three machine learning techniques, namely, Support Vector Machine (SVM), Decision Tree (DTree) and Random Forest, we propose and implement a novel dynamic analysis system called DroidWard to vet and classify the benign and malicious apps. DroidWard automatically runs apps, extracts features, and carries out the classification of benign and malicious apps. 666 Android apps are used in the experiments and the evaluation results show that DroidWard correctly classifies 98.54% of malicious apps with 1.55% false positives. Compared to existing work, DroidWard improves the TPR with 16.07% and suppresses the FPR with 1.31%, indicating it is more effective than existing methods.

We make the following contributions:

- We introduce a system called DroidWard to automatically vet Android apps through dynamic analysis.
- We evaluate DroidWard on a data set of 666 Android apps consisting of 258 malicious apps and 408 benign apps. The evaluation results show that DroidWard outperforms existing methods. It is able to identify 98.49% of malware samples with less than 1.55% of false positives.
- We explore 6 novel dynamic features from apps and employ these features in DroidWard. We discuss and analyze the most distinguishable dynamic features to classify apps with DroidWard.

This paper is organized as follows. Section 2 introduces the related work. Section 3 introduces DroidWard. Section 4 describes the novel features that we explore from apps. Section 5 introduces the data and classifiers. Comparative results are given and discussed in Sects. 6 and 7 concludes this paper.

2 Related work

Computer system and network security has been a widely studied topic as attacks against system or network infrastructure are currently major threats. Many methods have been proposed to ensure network security [12–17], system security [18–20] or cloud security [21–24]. The methods for ensuring and vetting Android apps are basically divided into two categories: static analysis and dynamic analysis. Static analysis normally decompiles an app and then analyzes its source codes prior to its installation. Fuchs et al. proposed ScanDroid [25] to extract the requested permissions claimed in the AndroidManifest.xml files of an app and then to use these permissions to identify malapps. Pandita et al. proposed WHYPER [26] that employed natural language processing techniques to explain why apps need certain permissions. The disadvantage of these methods only used permissions as features. It is unable to detect malapps that bypass requesting permissions. Peiravian et al. [27] combined permissions and API (Application Program Interface) calls and used machine learning methods to detect malapps. Arp et al. presented Drebin [28] to extract a broad static features through static analysis to build a linear SVM classier for the detection of malapps. Apvrille et al. [29] proposed a heuristic engine that statically pre-processed and prioritized samples to accelerate the detection of novel Android malware in the wild. In our previous work [30], we ranked the permissions according to their risks and systematically studied how well Android permissions perform in the detection of malapps. Experimental results show that it is required to extract new and more effective features in order to improve the detection performance [30]. We also developed a static analysis tool called SDFDroid [31] to identify the sensors' types and to generate the sensor data propagation paths in each app, in order to provide an up-to-date overview of sensor usage patterns in current apps by investigating what, why and how embedded sensors are used in all the apps collected from a complete app market. The sensor usage patterns are then used to identify whether an app has potential threat. In order to detect zero-day malapps, we developed a system called Anomadroid (anomaly Android malapp detection system) [32] that profiles the normal behaviors of Android apps based on only benign samples. Any app whose behaviors unacceptably deviate from the normal profile is identified as malicious. We also presented a system called Alde [33] to analyze privacy risk of analytics libraries in the Android ecosystem.

Although static analysis can rapidly identify existing malapps, it fails to analyze obfuscated and encrypted codes that are adopted by a large number of apps. Dynamic analysis is capable of vetting apps even with obfuscated or encrypted codes. Dynamic analysis is mainly based on the sandbox runtime analysis technology. It requires monitoring the flow of sensitive data, such as sensitive API calls in real time.

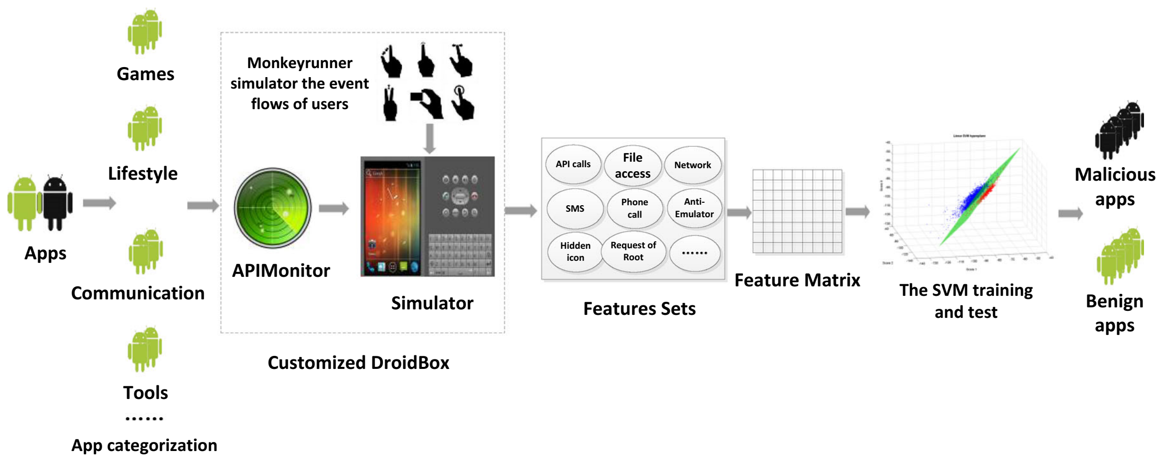


Fig. 1 Framework of DroidWard

The purpose of the monitoring is to analyze all the behaviors of an app in the simulator. Enck et al. proposed a taint tracking tool called TaintDroid [6] to mark the sensitive data. It traces and reviews the data that spreads through the program variables and Inter-Process Communication. However, TaintDroid only considered the data flow, and did not analyze the control flow. Afonso et al. [11] used the number of invocations of API and system calls as features to train various classifiers to analyze Android apps. This method relies on modifying the apps under analysis, which is easily detected and bypassed by malapps. Spreitzenbarth et al. [34] presented a system called Mobile-Sandbox based on TaintDroid and DroidBox. It combines static and dynamic analysis techniques to obtain Android apps' behavior. Martina et al. [7] proposed MARVIN that is also a hybrid of static analysis and dynamic analysis method. It uses a linear classifier and SVM to generate a risk score for an app. The risk score is then used to identify whether the app is malicious. The disadvantage of MARVIN is that it fails to achieve satisfactory detection accuracy with dynamic analysis. In order to thoroughly analyze Android apps through dynamic analysis, in this work, we propose a dynamic analysis system called DroidWard. In addition to the 9 dynamic features that have been used by existing methods, DroidWard extracts 6 novel types of features from apps. These features covers hidden icons, packet size, the number of distinct sensitive API calls and the number of API calls per unit time. The experimental results show that DroidWard outperforms the existing methods in the detection of malapps.

3 The framework of DroidWard

The framework of DroidWard is shown in Figure 1.

DroidWard performs dynamic analysis through three steps:

Table 1 App category and its sub-categories

Category	Sub-category
Tools	System Tools
	Security
	Browser
	Typewriting
Multimedia	Music
	Themes
	Video
	Photography
Lifestyle	Weather
	Other services
Social and communications	Communications
	Social networks
Learning and reading	News
	Office and education
	Reader
Game	Online game
	Leisure
	Action
	Puzzle & Card
Shopping & Payment	Financial Management
	Shopping & Payment

(1) App categorization. Because the behavior of each category of app is different, we need to categorize the apps before dynamic analysis. The categorization information refers to Google Play and typical Chinese app market Anzhi and AppChina. We summarize the categories in Table 1.

(2) Feature extraction. We customize DroidBox[1] in this work to extract dynamic features like the size of the net-

[1] DroidBox. https://github.com/pjlantz/droidbox, 2014.

Table 2 Novel types of dynamic features

Number	Features
1	Anti-simulator
2	Hidden apps' icons
3	Packet size
4	Number of distinct sensitive API calls
5	Number of API calls per unit time
6	Request Root permission

work packets and the sensitive API calls. As the data monitored by DroidBox is limited, we are motivated to modify the source code of DroidBox. We incorporate APIMonitor to monitor the sensitive API calls and use Monkeyrunner [35] to simulate the event flows of users. In the experiments, we use version Android 4.4.4 as the simulator. An app installed on the simulator is driven by the script to start the MainActivity. It is triggered to execute by event flows generated with Monkeyrunner by continuingly simulating user actions. In the experiments, the running time is set to 120 s. The dynamic behaviors of an app in the simulator are recorded in real time. After 120 s, the app will automatically end running and be uninstalled. The next app will be dynamically analyzed with DroidBox afterwards.

(3) Vetting and detection. We employ three classifiers, namely, SVM, Decision Tree and Random Forest for the detection of malapps. Combing the 9 features that have been used in existing work with the novel types of dynamic feature we explore in this work, we form in total 15 types of features to characterize the dynamic behaviors of an app. Each app is thus represented by the 15 features. The three classifiers will then identify whether the app is malicious based on the features.

DroidWard vets each test app for about 4 min including 2 min dynamic analysis. With the three classifiers, DroidWard will provide the detection results. If a test app has been vetted before, it will directly give the vetting results from the database, and this process costs a few seconds.

4 Novel types of dynamic features

In this work, we propose 6 novel types of features based on the comprehensive dynamic analysis of benign and malicious apps. The novel types of features are listed in Table 2.

We describe each feature in the following sub-sections.

Table 3 The attributes of file build.prop in the simulator and in the real devices

	Attributes in the simulator	Attributes in the real device
ro.product.model	sdk	model of the device
ro.build.tags	test-keys	release-keys
ro.kernel.qemu	1	——

4.1 Anti-simulator mechanism

Anti-simulator mechanism refers to the capability of identifying whether an app is run in a simulator. A malapp may not be run or may use unconventional ways to avoid the detection of simulator if the malapp detects that it is installed in a simulator. For example, app samples in Nymaim family identify whether they are installed in a simulator with Qemu File. If an app finds itself installed in the simulator, it will constantly call normal APIs through the API Hammering and consumes lots of resources, resulting in the timeout for the detection of simulator. It will thus bypass the detection of simulator.

The mechanism of anti-simulator is mainly recognized by the simulator files [36], e.g., build.prop files. Table 3 shows the attributes of file build.prop in the simulator and in the real Android mobile devices.

Malapps often detect the file through the API calRuntime.getRuntime().exec("getprop ro.product.model"). Therefore APIMonitor functions can be extended to monitor such API calls.

Compared to real devices, there are some distinct files or directories in the Android simulator, such as /system/bin/ qemu-props and /system/lib/libc_malloc_debug_qemu.so. Malapps usually search this kind of files to determine whether they are installed in the simulator. DroidBox can monitor the file access operations in order to detect whether an app probes the anti-simulator.

4.2 Hidden apps' icons

After an Android app is installed, usually a desktop icon will be generated. Some benign apps, e.g., apps for plug-in, will not generate icon. A number of malapps are motivated to hide their behaviors by not generating visional icons [32]. For example, a malapp named "core service" does not generate an icon in the desktop after it is installed in Android system. It occasionally pushes malicious services in the background. For example, it pushes advertising messages in the notification bar once the network state of the device changes. If a user accidentally click the advertisement, the pushed malware will be automatically downloaded and installed.

Clearly hiding the icon is an important feature of malicious apps. To detect whether the icon is hidden after the app

Table 4 API calls monitored

Number	Class	Method
1	Landroid/content/ContextWrapper	sendBroadcast; sendOrderedBroadcast; sendStickyBroadcast; startActivities; sendStickyOrderedBroadcast; startActivity openFileInput; openFileOutput
2	Landroid/net/Uri	parse
3	Ljava/io	read; write; readLine; newline; writeTo; append; format; print; printf; println
4	Ljava/net/URL	openConnection; openStream
5	Landroid/database/sqlite /SQLiteDatabase	openOrCreateDatabase; query; openDatabase; queryWithFactory; rawQueryWithFactory; rawQuery
6	Landroid/telephone/SmsManager	sendTextMessage; sendDataMessage; sendMultipartTextMessage
7	Landroid/telephone / TelephonyManager	getDeviceId; getSubscriberId; getCallState; getCellLocation
8	Ljava/security/MessageDigest	getInstance; update; digest
9	Ljavax/crypto/Cipher	getInstance; init; doFinal
10	Ljava/lang/Runtime	getRuntime

is installed, we make a screenshot prior to its installation and make another screenshot after its installation. We then use Python Imaging Library (PIL) to compare the two screenshots. If there is no difference between the two screenshots, the app does not hide its icon after its installation. Otherwise it hides.

4.3 Packet size

In general android apps involve network operations. The size of network packets indicates the communication status of apps. For example, apps of social networks such as WeChat and Twitter need to keep running and their packet sizes are usually large. Other native apps such as Notepad will generate small packet size. It is abnormal if packet sizes of these kinds of applications are large.

4.4 Number of distinct sensitive API calls

The number of distinct sensitive API calls is recorded when an app is running. If an app invokes many types of sensitive API calls, it is potentially malicious. Therefore, it is important to record the number of sensitive APIs in the process of an app's execution for the detection of malapps.

The APIs monitored by APIMonitor in this paper are shown in table 4.

4.5 The number of API calls per unit time

Some malapps steal users' privacy information as soon as they are installed, or accept remote instructions to prepare for the next attacks. Repackaged app "Super Marie", for example, immediately uploads user's contacts, the list of installed apps and phone number once it is installed. Therefore, the number of API calls invoked by malapps and by benign apps per unit time may be different. In this work, we use the unit time as 10, 20, 30, 40, 60 and 120 s, respectively.

4.6 Request Root permission

Some apps request Root permissions to perform sensitive actions like system backup, screen shots, if the Android device has been rooted. A malapp usually probes whether its host device has been rooted once it is installed. If the malapp confirms that its host device has been rooted, it will request Root permission. Malapps often execute the following script to access Root [37]:

Process process = Runtime.getRuntime().exec("su");

In this work, we monitor the Runtime.getRuntime() and use the parameters of the API call to determine if the app has requested Root permissions.

5 Data set and classifiers

5.1 Data set

We crawled apps from Anzhi Market [38] in 2015 and collected a number of malapps in the wild. With the help of VirusTotal [39], we are able to label the apps used in the experiments. The app set used in the experiments is described

Table 5 The app set used in the experiments

Number of benign apps	Number of malapps	Total number of apps
408	258	666

Table 6 Numbers of benign apps and malapps in each category

Category	Sub-category	# of benign apps	# of malapps
Tools	System tools	24	28
	Security	11	12
	Browser	7	8
	Typewriting	9	8
Multimedia	Music	14	7
	Themes	12	6
	Video	13	5
	Photography	9	5
Lifestyle	Weather	23	4
	Other Services	31	8
Social and Communications	Communications	20	20
	Social Networks	21	18
Learning and Reading	News	18	6
	Office and Education	13	3
	Reader	15	4
Game	Online Game	51	39
	Leisure	32	22
	Action	35	24
	Puzzle & Card	30	21
Shopping & Payment	Financial Management	12	6
	Shopping & Payment	8	4
Total	——	408	258

in Table 5. The number of benign apps and malapps in each category is described in Table 6.

We randomly select 60% of benign apps and of malapps as the training set. The remaining 40% of apps compose the test set. DroidWard constructs a feature vector for each app to build the detection models with SVM, Decision Tree and Random Forest. The models are then used for the detection.

5.2 Classifiers

We employ Support Vector Machine (SVM), Decision Tree (DTree) and Random Forest (RF) as the classifiers for the detection of malapps.

Support Vector Machine (SVM). SVM [30,40] is a classical method used for binary classification. It finds a hyperplane that separates the d-dimensional data perfectly into its two classes. The optimal separating hyperplane is defined as the one resulting in the maximal margin.

In this work, we use linear SVM for the classification. The output of a linear SVM is $\mathbf{u} = \vec{w} \cdot \vec{x} - \mathrm{b}$, where $\vec{w}$ is the normal vector to the hyperplane and $\vec{x}$ is the input vector. The margin in linear SVM is defined by the distance between the hyperplane and the nearest of the positive and negative samples. Positive samples usually refer to malapps while negative samples refer to benign apps. The solution is to maximize the margin and it can be computed as:$\min \frac{1}{2} ||\vec{w}||^2$ subject to $y_i \, (\vec{w} \cdot \vec{x}_i - b) \geq 1, \forall i$ where $\vec{x}_i$ is the ith training sample and y_i is its correct output of the SVM. The output $\mathbf{u}$ is the classification results for test samples.

Decision Tree (DTree). Decision tree classifiers [41–43] are based on the "divide and conquer" strategy to construct an appropriate tree from a given learning set S containing a set of labeled instances. Each sample is represented by features and the class it belongs to.

As a well-known decision tree method, C4.5 [41] builds decision trees from a set of training data with information entropy. C4.5 chooses the data feature that most effectively splits its set of samples into subsets enriched in one class or the other. Formally, given a learning set S and a data feature X, the Gain Ratio is computed as:

$$GR(S|X) = \frac{IG(S|X)}{-\sum_i \frac{|S_i|}{|S|} log_2 \frac{|S_i|}{|S|}} \qquad \text{(x)}$$

where S_i is the subset of S for which feature X have a value and $|S|$ is the number of instances in S.

The decision trees are constructed as a set of rules during learning phase. These rules are then used to predict the classes of test instances.

Random Forest. Random Forest [30,44] is an ensemble learning method for classification and regression that builds many decision trees at training time and combines their output for the final prediction. Like other ensemble methods, Random Forest often outperforms a single tree on classification accuracy.

6 Experimental results and analysis

6.1 The experimental results

To facilitate comparison, in the experiments, we used 15 features including our proposed 6 novel features as well as the 9 features that have been used in existing work. The features used in our experiments are summarized in Table 7.

Table 7 The dynamic features used in the experiments

#	Feature	Description	Existing/Novel
	File access	File access, including the access of file paths	Existing feature
2	Encrypted operations	API calls generated by encrypted actions including types of encryption and key values	Existing feature
3	Data leakage	Data leakage, including the leaked data type and paths	Existing feature
4	Permission leakage	Permission leakage	Existing feature
5	Dynamic loading	The loading information during runtime	Existing feature
6	Network traffic sent	Network traffic sent, including the destination IP and port number	Existing feature
7	Network traffic received	Network traffic received, including the source IP and port number	Existing feature
8	SMS operation	SMS operation, including SMS content and phone number	Existing feature
9	Telephone operation	Telephone operations	Existing feature
10	Anti-simulator	Anti-simulator mechanism	Novel features 1
11	Hidden apps' icons	Hiding the app's icon after its installation	Novel features 2
12	Packet size	The packet size generated	Novel features 3
13	Number of distinct sensitive API calls	Number of sensitive API calls	Novel features 4
14	Number of API calls per unit time	Number of API calls per unit time	Novel features 5
15	Request Root permission	Request Root permission	Novel features 6

Table 8 The experimental results with 9 existing features and with an additional novel feature we have explored

	The feature set	Accuracy (%)	TPR (%)	FPR (%)
1	9 existing features [7]	86.36	82.47	2.86
2	9 existing features + category of apps	92.42	89.88	2.32
3	9 existing features + novel features 1	88.63	84.37	2.48
4	9 existing features + novel features 2	89.36	86.02	2.56
5	9 existing features + novel features 3	88.63	85.11	2.63
6	9 existing features + novel features 4	87.88	84.21	2.70
7	9 existing features + novel features 6	91.67	88.89	2.38

Table 9 The experimental results with 15 features

Classifier	Feature set	Accuracy (%)	TPR (%)	FPR (%)
SVM	15 features	98.49	98.54	1.55
Decision Tree	15 features	98.05	98.03	1.94
Random Forest	15 features	97.59	97.55	2.33

In order to evaluate the detection performance with the proposed novel features, we conducted a series of experiments with different combinations of the feature sets. Table 8 shows the true positive rates (TPR) and false positive rates (FPR) for the detection of malapps.

Combing the existing 9 features with our proposed 6 features, the detection accuracy, TPR and FPR generated with DroidWard are shown in Table 9. The Receiver Operating Characteristic (ROC) curves are usually used to evaluate the detection performance of different detection models 错误!未找到引用源。. The ROC curves of the detection performance with SVM using 15 features and 9 existing features are shown in Figure 2. In our experiments we use 5-fold cross-validation.

It is seen from Table 8 that the detection performance improves even using a single novel feature we proposed. From Table 9, we observe that all the three classifiers, namely, SVM, Decision Tree and Random Forest achieve high detection rates with low false positive rates. It is also seen from Figure 2 that the detection accuracy of SVM increases from 86.36 to 98.49% after combining the 6 novel features with 9 existing features.

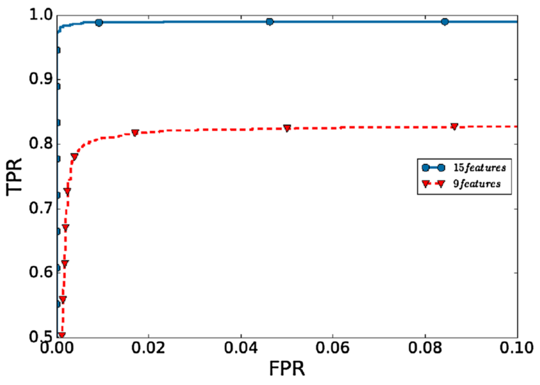


Fig. 2 The ROC with SVM classifier when using 15 features and 9 existing features

6.2 Analysis of the mechanism of anti-simulator

The mechanism of anti-simulator is an effective feature for the detection of malapps. In the experiments, we find that there are 5 apps having anti-simulator mechanism and all these 5 apps are malicious. Among the 5 apps, 3 are detected by checking the file build.prop and the other 2 are detected with File Qemu. When the 5 apps are run in a real device with aLogcat recording the system's log, we find that the 5 apps have obvious malicious behaviors. For example, the app named Lightdd pushes advertisements to users after it is installed. Potentially harmful apps will be automatically downloaded if the user click the advertisement. In addition, the app stealthily sends IMEI information to the destination IP 221.181.xx.xx.

6.3 Analysis of behaviors of hiding icons

In the experiments, we find that 9 apps that hide their icons are malicious. However, 2 benign apps also hide their icons. Through comprehensive investigation, it is observed that the 9 apps that hide their icons on the desktop have shown their malicious behaviors, such as accessing system's files, accessing contact information, or reading SMS. For example, the app named "classic calendar" [45] does not generate icon after its installation. However, it is active in background for regularly sending users' privacy information to a remote host. This privacy information includes IMEI, location information, SMS, etc. The 2 benign apps also generate no icon on the desktop. One app is a wallpaper type and the other app is a plug-in for multimedia players.

6.4 Analysis of number of API calls per unit time

In the experiments, we use 15 features for vetting the apps. For the feature "number of API calls per unit time", we set "unit time" as 10, 20, 30, 40, 60 and 120s, respectively. The

Table 10 The detection results with different "unit time"

	Unite time (s)	TPR (%)	FPR (%)
1	10	97.56	2
2	20	98.04	2.33
3	30	98.54	1.55
4	40	98.05	1.58
5	60	96.71	1.95
6	120	96.58	2.04

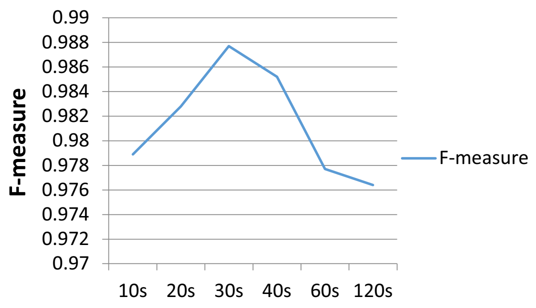


Fig. 3 The F-measure scores with different "unit time"

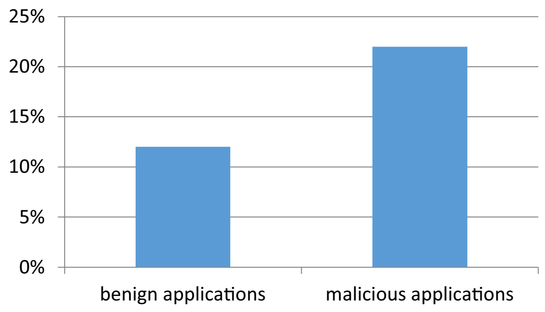


Fig. 4 Proportion of malapps and benign apps that request Root permissions

detection results are shown in Table 10 and in Fig. 3. It is seen that the detection results reach best if "unit time" is set as 30s. It is shown that the number of API calls per unit time is different between benign apps and malapps. This means that malapps have great possibility to call sensitive APIs to steal user's privacy information shortly after installation, especially in about 30 s. This feature is thus effective for the vetting of apps.

6.5 Analysis of requesting Root permissions

Figure 4 shows the proportion of malapps and benign apps that request Root permissions. It is observed that the proportion of malapps that request Root permissions is significantly higher than that of benign apps. In total there are 105 apps requesting Root permission. Most of these apps are malicious and are categorized as tools, social communications or games. They are motivated to make themselves attractive, easy to use, and to trick Root permission. In the experiments, for example, a malapp named "Root assistant" regularly pops up Root permission request. Once granted, another app called "timer" will be automatically downloaded in the background and installed. The malapp "timer" regularly steals and sends user's privacy information, including IMEI, IMSI to a remote host IP as 223.154.xx.xx.

7 Conclusion

As the behaviors of Android apps have become increasingly sophisticated, effectively vetting and detecting malapps has become a crucial yet challenging task. Malapps have used techniques such as code obfuscation or encryption to escape the detection with traditional static analysis methods. Dynamic analysis methods vet apps by running them on a real device or on a simulator and are able to deal with the obfuscation and encryption of codes. However, existing dynamic analysis methods are not effective to vet apps, as only a limited number of features has been explored from apps. In this work, we propose an effective dynamic analysis method called DroidWard in the aim to extract most relevant and effective features to characterize malicious behavior and to improve the detection accuracy of malicious apps. DroidWard extracts 6 novel types of effective features from apps through dynamic analysis. The 6 features are anti-simulator, hidden apps' icons, packet size, the number of distinct API calls, the number of API calls per unit time and request Root permission. DroidWard runs apps, extracts features and identifies benign and malicious apps with three classifiers, namely, Support Vector Machine (SVM), Decision Tree and Random Forest. 666 Android apps are used in the experiments and the evaluation results show that combining our 6 novel types of features and the other 9 features that have been used in existing work, DroidWard correctly classifies 98.54% of malicious apps with 1.55% of false positives. Compared to existing work, DroidWard improves the TPR with 16.07% and suppresses the FPR with 1.31%, indicating that it is more effective than existing methods. Through thorough investigation, we find that all the misclassified benign apps request sensitive API calls or permissions including starting a service on startup, installing packages, sending and receiving SMS. The main reason for the false negatives is that they are inactive during dynamic analysis.

One limitation of our method is that the number of test samples is still limited. The other limitation is that the event flows generated by Monkeyrunner are not diverse. As for the future work, we are exploring more effective dynamic features from apps to improve the performance of malapp detection. How to diversely generate user event flows so as to best simulate the behaviors of apps is also being investigated.

Acknowledgements This work was supported in part by the Scientific Research Foundation through the Returned Overseas Chinese Scholars, Ministry of Education of China, under Grant K14C300020, in part by Shanghai Key Laboratory of Integrated Administration Technologies for Information Security, in part by ZTE Corporation, and in part by the 111 Project under Grant B14005.

References

1. F-Secure, Threat Report 2015. https://www.f-secure.com/documents/996508/1030743/Threat_Report_2015.pdf (2015)
2. Greenberg, A.: Scanner identifies thousands of malicious Android apps on Google Play, other markets. http://www.scmagazine.com/scanner-identifies-thousands-of-malicious-android-apps-on-google-play-other-markets/article/435387/ (2015)
3. Hirst, S.: Lookout Discovers SocialPath Malware in Google Play Store. https://vpncreative.net/2015/01/10/lookout-socialpath-malware-google-play (2015)
4. Lockheimer, H.: Android and Security. http://googlemobile.blogspot.com/2014/02/android-and-security.html (2014)
5. Zhou, Y., Jiang, X.: Dissecting android malware: characterization and evolution. IEEE Symposium on Security and Privacy, pp. 95–109, 2012
6. Enck, W., Gilbert, P., Han, S., et al.: TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans. Comput. Syst. (TOCS) **32**(2), 5 (2014)
7. Lindorfer, M., Neugschwandtner, M., Platzer, C.: Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis. 39th IEEE Annual Computer Software and Applications Conference (COMPSAC), vol. 2, pp. 422–433, 2015
8. Lindorfer, M., Neugschwandtner, M., Weichselbaum, L., et al.: Andrubis–1,000,000 apps later: a view on current Android malware behaviors. Third International IEEE Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS), p 3-17, 2014
9. Felt, A. P., Chin, E., Hanna, S., et al.: Android permissions demystified. 18th ACM Conference on Computer and communications security, pp. 627-638, 2011
10. Dietz, M., Shekhar, S., Pisetsky, Y., et al.: QUIRE: lightweight provenance for smart phone operating systems. USENIX Security Symposium, vol. 31, 2011
11. Afonso, V.M., de Amorim, M.F., Grégio, A.R.A., et al.: Identifying Android malware using dynamically obtained features. J. Comput. Virol. Hacking Tech. **11**(1), 9–17 (2015)
12. Wang, W., Guan, X., Zhang, X.: Processing of massive audit data streams for real-time anomaly intrusion detection. Comput. Commun. **31**(1), 58–72 (2008)
13. Wang, W., Liu, J., Pitsilis, G., et al.: Abstracting massive data for lightweight intrusion detection in computer networks. Information Sciences (online first), 2016
14. Wang, W., Guyet, T., Quiniou, R., et al.: Autonomic intrusion detection: adaptively detecting anomalies over unlabeled audit data streams in computer networks. Knowl.-Based Syst. **70**, 103–117 (2014)
15. Wang, W., Battiti, R.: Identifying intrusions in computer networks with principal component analysis, First International Conference on Availability, Reliability and Security. IEEE, p 1-8, 2006
16. Zhang, X., Furtlehner, C., Germain-Renaud, C., et al.: Data stream clustering with affinity propagation. IEEE Trans. Knowl. Data Eng. **26**(7), 1644–1656 (2014)
17. Zhang, X.L., Lee, T.M.D., Pitsilis, G.: Securing recommender systems against shilling attacks using social-based clustering. J. Comput. Sci. Technol. **28**(4), 616–624 (2013)

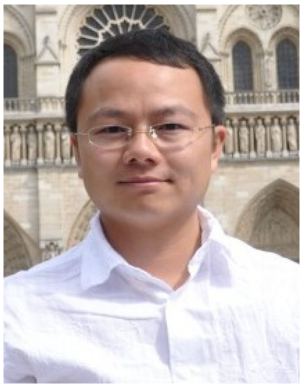
18. Wang, W., Zhang, X., Gombault, S.: Constructing attribute weights from computer audit data for effective intrusion detection. J. Syst. Softw. **82**(12), 1974–1981 (2009)
19. Guan, X., Wang, W., Zhang, X.: Fast intrusion detection based on a non-negative matrix factorization model. J. Netw. Comput. Appl. **32**(1), 31–44 (2009)
20. Wang, W., Guan, X., Zhang, X., Yang, L.: Profiling program behavior for anomaly intrusion detection based on the transition and frequency property of computer audit data. Comput. Secur. **25**(7), 539–550 (2006)
21. Huang, X., Li, J., Li, J., et al.: Securely outsourcing attribute-based encryption with checkability. IEEE Trans. Parallel Distrib. Syst. **25**(8), 2201–2210 (2014)
22. Li, J., Li, J., Chen, X., et al.: Identity-based encryption with outsourced revocation in cloud computing. IEEE Trans. Comput. **64**(2), 425–437 (2015)
23. Li, J., Li, Y.K., Chen, X., et al.: A hybrid cloud approach for secure authorized deduplication. IEEE Trans. Parallel Distrib. Syst. **26**(5), 1206–1216 (2015)
24. Li, J., Chen, X., Li, M., et al.: Secure deduplication with efficient and reliable convergent key management. IEEE Trans. Parallel Distrib Syst. **25**(6), 1615–1625 (2014)
25. Fuchs, A.P., Chaudhuri, A., Foster, J.S.: Scandroid: automated security certification of android. Technical report, University of Maryland (2009)
26. Pandita, R., Xiao, X., Yang, W., et al.: Whyper: towards automating risk assessment of mobile applications, Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13), pp. 527-542, 2013
27. Peiravian, N., Zhu, X.: Machine learning for android malware detection using permission and api calls. IEEE 25th International Conference on Tools with Artificial Intelligence. IEEE, pp. 300-305, 2013
28. Arp, D., Spreitzenbarth, M., Hubner, M., et al.: DREBIN: effective and explainable detection of android malware in your pocket. In: The 2014 Network and Distributed System Security Symposium (NDSS), pp. 1–12
29. Apvrille, A., Strazzere, T.: Reducing the window of opportunity for Android malware Gotta catch'em all. J. Comput. Virol. **8**(1–2), 61–71 (2012)
30. Wang, W., Wang, X., Feng, D., Liu, J., Han, Z., Zhang, X.: Exploring permission-induced risk in android applications for malicious application detection. IEEE Transactions on Information Forensics and Security **9**, pp. 1869–1882 (2014)
31. Liu X, Liu J, Wang W, Exploring sensor usage behaviors of Android applications based on data flow analysis. IPCCC, p 1-8, 2015
32. Su, D., Wang, W., Wang, X., Liu, J.: Anomadroid: profiling Android applications' behaviors for identifying unknown malapps. 15th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (IEEE TrustCom), 2016
33. Liu, X., Zhu, S., Wang, W., Liu, J.: Alde: privacy risk analysis of analytics libraries in the android ecosystem. 12th EAI International Conference on Security and Privacy in Communication Networks (SecureComm), 2016
34. Spreitzenbarth, M., Freiling, F., Echtler, F., et al.: Mobile-sandbox: having a deeper look into android applications. Proceedings of the 28th Annual ACM Symposium on Applied Computing. ACM, pp. 1808-1815, 2013
35. Monkeyrunner. https://developer.android.com/studio/test/monkeyrunner/index.html
36. Apvrille, A.: Apktool: a tool for reverse engineering android apk files. https://ibotpeaches.github.io/Apktool/
37. Ho, T.H., Dean, D., Gu, X., et al.: PREC: practical root exploit containment for android devices. Proceedings of the 4th ACM conference on data and application security and privacy. ACM, pp. 187-198, 2014
38. Anzhi Market. http://www.anzhi.com
39. Virustotal. https://www.virustotal.com/
40. Burges, C.J.C.: A tutorial on support vector machines for pattern recognition. Data Min. Knowl. Discov. **2**(2), 121–167 (1998)
41. Quinlan, J.: C4.5: programs for machine learning. Morgan Kaufmann Publishers, Burlington (1993)
42. Wang, W., Gombault, S., Guyet, T.: Towards fast detecting intrusions: using key attributes of network traffic, Internet Monitoring and Protection, ICIMP'08. The Third International Conference on. IEEE , p 86–91, 2008
43. Wang, W., He, Y., Liu, J., et al.: Constructing important features from massive network traffic for lightweight intrusion detection. IET Inf. Secur. **9**(6), 374–379 (2015)
44. Breiman, L.: Random forests. Mach. Learn. **45**(1), 5–32 (2001)
45. Le Thanh, H.: Analysis of malware families on android mobiles: detection characteristics recognizable by ordinary phone users and how to fix it. J. Inf. Secur. **4**(04), 213 (2013)

Yubin Yang is currently Ph.D. candidate at the School of Computer Science & Engineering, South China University of Technology, China. He previously earned a M.Sc. Degree at the University of New South Wales, Australia in 1999 and a B.Sc. degree at Guangdong University of Technology, China in 1997. His current research interests focus on information security.

Zongtao Wei is currently pursuing the M.Sc degree with the School of Computer and Information Technology, Beijing Jiaotong University, China. He received the B.Sc. degree from Beijing Jiaotong University in 2014. His main research interests include mobile and network security.

Yong Xu received the B.Sc., M.Sc. and Ph.D. degrees in Mathematics from Nanjing University, China, in 1993, 1996 and 1999 respectively. From 1999 to 2001, he was a postdoc research fellow in Computer Science at South China University of Technology and became a faculty member afterward. He is currently a Professor with the School of Computer Science & Engineering at South China University of Technology, China. His research interests include image analysis, image/video recognition and image quality assessment. He is the member of IEEE Computer Society and of ACM.

Haiwu He is a 100 Talent Professor at the CNIC (Computer Network Information Center), Chinese Academy of Sciences in Beijing. He has been a Chunhui Scholar of Ministry of Education of China since 2013. He received his M.Sc. and Ph.D. degrees in computing from the University of Sciences and Technologies of Lille, France, respectively in 2002 and 2005. He was a postdoctoral researcher at INRIA Saclay, France in 2007. He was a research engineer expert at INRIA Rhone-Alpes in Lyon, France from 2008 to 2014. He has published about 30 refereed journal and conference papers. His research interest covers P2P distributed system, Cloud computing and BigData.



Wei Wang is currently associate professor in the School of Computer and Information Technology, Beijing Jiaotong University, China. He earned his Ph.D. degree in control science and engineering from Xi'an Jiaotong University, China, in 2006. He was a postdoctoral researcher in University of Trento, Italy, from 2005–2006. He was a postdoctoral researcher in TELECOM Bretagne and in INRIA, France, from 2007–2008. He was a European ERCIM Fellow in Norwegian University of Science and Technology (NTNU), Norway, and in Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, from 2009–2011. He visited INRIA, ETH, NTNU, CNR, and New York University Polytechnic. He is young AE of Frontiers of Computer Science Journal. He has authored or co-authored over 50 peer-reviewed papers in various journals and international conferences. His main research interests include mobile, computer and network security.