MalScan: Fast Market-Wide Mobile Malware Scanning by Social-Network Centrality Analysis

Yueming Wu*,‡, XiaoDi Li¶, Deqing Zou*,‡,§ ✉, Wei Yang¶, Xin Zhang*,‡, Hai Jin*,†
*National Engineering Research Center for Big Data Technology and System, Cluster and Grid Computing Lab
Services Computing Technology and System Lab, Big Data Security Engineering Research Center
†School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China
‡School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, 430074, China
§Shenzhen Huazhong University of Science and Technology Research Institute
¶University of Texas at Dallas

***Abstract*—Malware scanning of an app market is expected to be scalable and effective. However, existing approaches use either syntax-based features which can be evaded by transformation attacks or semantic-based features which are usually extracted by performing expensive program analysis. Therefor, in this paper, we propose a lightweight graph-based approach to perform Android malware detection. Instead of traditional heavyweight static analysis, we treat function call graphs of apps as social networks and perform social-network-based centrality analysis to represent the semantic features of the graphs. Our key insight is that centrality provides a succinct and fault-tolerant representation of graph semantics, especially for graphs with certain amount of inaccurate information (*e.g.,* inaccurate call graphs). We implement a prototype system, *MalScan*, and evaluate it on datasets of 15,285 benign samples and 15,430 malicious samples. Experimental results show that *MalScan* is capable of detecting Android malware with up to 98% accuracy under one second which is more than 100 times faster than two state-of-the-art approaches, namely *MaMaDroid* and *Drebin*. We also demonstrate the feasibility of *MalScan* on market-wide malware scanning by performing a statistical study on over 3 million apps. Finally, in a corpus of dataset collected from Google-Play app market, *MalScan* is able to identify 18 zero-day malware including malware samples that can evade detection of existing tools.**

***Index Terms*—Lightweight feature, Android Malware, API Centrality, Market-wide**

I. Introduction

The explosive growth of Android devices and applications (app for short) has brought in several Android markets and spurred the growth of Android malware. Millions of apps have been installed by Android users around the world from various app markets. Up to the end of the third quarter of 2018, the number of new malware apps had an increase of over 40 percent compared to the same period last year[1]. Stopping the spread of malware primarily relies on the effort from the app markets, since they are the first place to provide installations of Android apps. Therefor, market-wide mobile malware scanning can be the primary task to prevent the fast malware spreading.

[1]https://www.gdatasoftware.com/blog/2018/11/31255-cyber-attacks-on-android-devices-on-the-rise.

Android malware scanning depends on the techniques for detecting Android malware. Existing mobile malware detection approaches extract program features [1], [2], [3], [4] to distinguish benign apps from malware. However, many of the techniques can be easily evaded by obfuscation because these techniques are lack of semantic and contextual information of the program behaviors. To overcome the challenge, several systems have been proposed to focus on distilling an app's program semantics into a graph representation and detecting malware by matching the graphs. Nevertheless, these graph-based techniques such as *DroidSIFT* [5] and *Apposcopy* [6] have two main limitations. First, graph matching is typically time-consuming because a graph often contains thousands of nodes. For instance, the average running time of *DroidSIFT* and *Apposcopy* to analyze an app is 175.8 seconds [5] and 275 seconds [6], respectively. Additionally, these graph-based systems conduct graph matching based on similarity to graphs of existing malware making the systems perform poorly on new malware instances due to the constant evolution of Android malware [7]. Therefor, these graph-based techniques are not scalable and realistic to complete market-wide malware scanning.

To further enhance the state-of-the-art techniques, approaches such as *MaMaDroid* [8] tried to use coarser-granularity information (*e.g.,* using package-level information instead of method-level information) and divide graphs into subgraphs (*e.g.,* using multiple pairwise invocation relationships instead of a call graph representing all invocation relationships). It has validated the robustness against Android malware evolution and increase the lifetime of the trained model. However, the average running time of an app is 165.63 seconds (Table VIII and Figure 7) which is not applicable for malware scanning of an app market. Additionally, the subgraphs (*e.g.,* the pairwise invocations) cannot fully reflect dependencies among method calls, thus the approaches are lack of key information to distinguish some of the malicious apps from benign ones. Moreover, such approach requires a sizable amount of memory when performing classification because of its large feature sets by extracting all pairwise invocations [8]. Another state-of-the-art technique for malware

scanning is *MassVet* [9], which abstracts the UIs of an app as a directed graph where each node is a view and each edge is a relationship description. It has validated the high efficiency on malware scanning, however, it can only detect repackaged malware and can cause a false negative when the app is a new malware.

To address these limitations, in this paper, we propose *MalScan*, a semantic-preserving market-wide malware scanning system that can accurately detect a malicious Android app in at least 0.7 seconds and at most 4.92 seconds on average (Table VIII and Figure 7). To preserve the program semantic information while perform fast processing on all the information, we regard the function call graph as a complex social network and perform centrality analysis [10], [11] to represent the semantic features of the graph. Both call graphs and social networks are static graphs representing dynamic behaviors. Call graphs represent the program behaviors and each edge of a call graph may represent multiple function invocations. Social networks represent the social behaviors and each edge of a social network may represent multiple social communications. Centrality measures the 'importance' of a function in the whole function call graph and reflects the structural attribute/properties of the graph. Because of its succinct representation, centrality works well on graphs with inaccurate information [12], [13], [14], [15], making centrality a perfect candidate to represent graphs obtained from low-cost program analysis (*e.g.,* context- and flow-insensitive analysis). By performing centrality analysis on each sensitive API method, *MalScan* provides a balance between abstracting graph details to defend against obfuscation and preserving semantic information to distinguish between malware and benign apps.

We evaluate *MalScan* from seven perspectives over malware spanning from January 2011 to December 2018 to better observe the robustness of *MalScan* against the evolution in Android malware. Our first four experiments focus on the effectiveness of malware detection, robustness against evolution of Android apps, robustness against adversarial attack, and runtime overheads of *MalScan*. In addition, we conduct a statistical study to investigate the scalability of *MalScan* on real-world app market (*e.g.,* Google-Play). We totally crawl over 3 million apps' information and the analysis result suggests that *MalScan* is capable of scanning malware on Google-Play app market. Furthermore, we utilize and combine the different centrality experimental results on malware detection. The experimental result indicates that it can indeed improve the effectiveness on Android app classification. Finally, we demonstrate the capability of *MalScan* in detecting real-world zero-day malware. Specifically, in a corpus of apps collected from Google-Play app market, *MalScan* is able to identify 18 zero-day malware including malware samples that can evade detection of existing tools [16].

In summary, this paper makes the following contributions:

- We propose a novel lightweight method to perform classification on Android malware by analyzing the centrality of sensitive API calls within a function call graph of an app.
- We design and implement a prototype system, *MalScan*, a novel, automatic, and efficient system that can perform classification on large-scale Android malware with high accuracy.
- We conduct a comprehensive evaluation using 15,285 benign samples and 15,430 malicious samples. Experimental results show that *MalScan* can complete the classification of an app in 0.7 seconds on average with up to 98% accuracy.

Paper organization. The remainder of the paper is organized as follows. Section II presents the preliminary study on degree centrality of Android apps. Section III introduces our system. Section IV reports the experimental results. Section V discusses the future work. Section VI describes the related work. Section VII concludes the present paper.

II. Preliminary Study on Centrality

A social network is a social structure made up of a set of social actors (such as individuals or organizations), sets of dyadic ties, and other social interactions between actors[2]. The source code of an app is made up of a set of functions, and there are several call relationships between them. If we regard functions as actors and the call relationships as social interactions, the function call graph of an app can be regarded as a social network.

Centrality concepts were first developed in social network analysis which quantify the importance of a node in the network and have the potential to unveil the structural pattern of the network. Centrality measures are very useful for network analysis, and much work has been proposed to use centrality measures in different areas, such as biological network [12], co-authorship network [13], transportation network [14], criminal network [15], affiliation network [17]. There have been proposed many different centrality measures, such as degree centrality [10], katz centrality [11], closeness centrality [10], harmonic centrality [18], betweenness centrality [19], percolation centrality [20], cross-clique centrality [21], dissimilarity-based centrality [22].

On the one hand, centrality measures can indicate the importance of a node within a network and have the potential to unveil the structural patterns and behaviors. On the other hand, malware usually invokes sensitive API calls to perform malicious activities. Therefor, we perform a preliminary study to investigate a question:

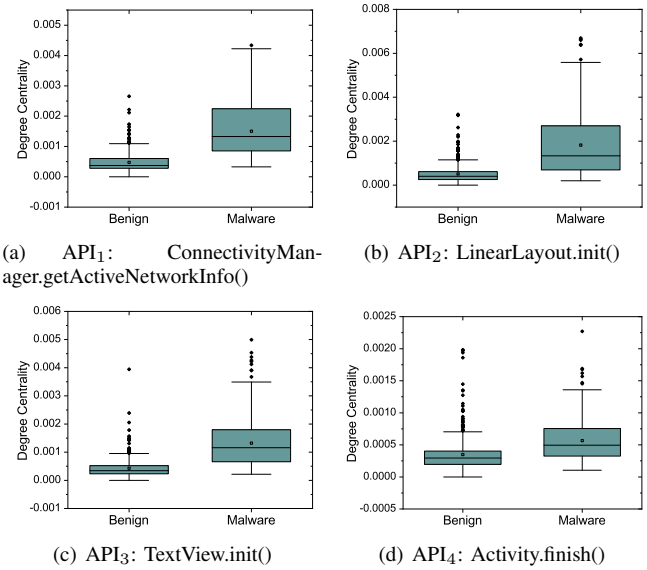
Can the centrality of sensitive API calls reflect the difference between benign apps and malicious apps? In other words, is there a significant difference between the centrality of sensitive API calls in benign apps and malicious apps?

To answer the proposed question, we first randomly select 500 benign apps and 500 malicious apps from AndroZoo [23], then the call graphs are extracted by using static analysis. Given a call graph, we perform centrality analysis only for

[2]https://en.wikipedia.org/wiki/Social_network.

TABLE I: Summary of results of *one way-ANOVA*, with α = 0.05 for degree centrality values of sensitive API call *ConnectivityManager.getActiveNetworkInfo()* in benign apps and malicious apps. [SS: sum of squares, df: degree of freedom, MS: mean square, F: calculated F-value, P: calculated p-value, F crit: critical value of F]

ANOVA: Single factor						
Summary						
Groups	**Count**	**Sum**	**Average**	**Variance**		
Degree centrality in benign apps	500	0.237218	0.000474	1.08E-07		
Degree centrality in malicious apps	500	0.751806	0.001504	6.51E-07		
ANOVA						
Source of variance	**SS**	**df**	**MS**	**F**	**P-value**	**F crit**
Between groups	0.000265	1	0.000265	697.6574	5.3E-117	3.850793
Within groups	0.000379	998	3.8E-07			
Total	0.000644	999				



(a) API_1: ConnectivityManager.getActiveNetworkInfo()

(b) API_2: LinearLayout.init()

(c) API_3: TextView.init()

(d) API_4: Activity.finish()

Fig. 1: The degree centrality distributions of sensitive API calls between benign apps and malicious apps

the node representing security-sensitive methods (by using a list of security-sensitive methods [24]). We then conduct a preliminary frequency analysis to check whether centrality can indeed suggest the inherent differences between benign apps and malicious apps. We select the top 10 frequently-invoked sensitive API calls as the test object. Due to the limitation of the page, we only show a portion of our results in Figure 1. From the results presented in Figure 1, we find that the degree centrality of sensitive API calls can be considerably different between benign apps and malicious apps. To obtain more determinate results, we apply *Analysis of Variance (ANOVA)* [25] to research the difference of these centrality values between benign apps and malicious apps. Here null hypothesis H_0 is that the centrality of sensitive API calls between benign apps and malicious apps are similar and there is no significant difference in their means. We apply *one-way ANOVA* to test whether we can reject or accept the null hypothesis.

P-value is the probability when the null hypothesis H_0 is true after performing statistical test with a pre-determined probability α (we select α = 0.05 in our tests). If the calculated p-value is below α, then the null hypothesis H_0 is rejected. Table I depicts the summary results by performing *one-way ANOVA* on the centrality values in Figure 1(a). As shown in Table I, the average value of degree centrality of *ConnectivityManager.getActiveNetworkInfo()* in malicious apps is 0.001504 while is 0.000474 in benign apps. Particularly, the p-value in Table I is 5.3E-117 which is extremely less than 0.05, by this we can reject the hypotheses H_0. In other words, the alternate hypothesis H_1 (*i.e.,* the difference of degree centrality of *ConnectivityManager.getActiveNetworkInfo()* between benign apps and malicious apps is significant) is accepted.

TABLE II: P-values of degree centrality in benign apps and malicious apps in Figure 1. [API_1: *ConnectivityManager.getActiveNetworkInfo()*, API_2: *LinearLayout.init()*, API_3: *TextView.init()*, API_4: *Activity.finish()*]

API Call	**API_1**	**API_2**	**API_3**	**API_4**
P-value	5.3E-117	1.20667E-85	8.05623E-91	3.9192E-30

We also apply *one-way ANOVA* on the other dataset in Figure 1. However, due to the limited page, we only show the p-values in Table II[3]. From the results shown in Table II, all the p-values are extremely less than 0.05, which indicates that the degree centrality of sensitive API calls can reflect the inherent difference between benign apps and malicious apps.

Therefor, based on the observation, we build a model and propose a lightweight Android malware detection system by analyzing the centrality of sensitive API calls within a call graph.

III. System Architecture

A. System Overview

As shown in Figure 2, *MalScan*'s operation goes through three main phases: *Static Analysis*, *Centrality Analysis*, and *Classification*.

- ***Static Analysis:*** This phase aims at extracting the function call graph of an app based on static analysis, where each node is a function that can be an API call or a user-defined function.
- ***Centrality Analysis:*** After obtaining the call graph of an app, we then compute the centrality of sensitive API calls

[3]The summary results of Figure 1(b) to (d) are available in https://github.com/malscan-android/malscan.

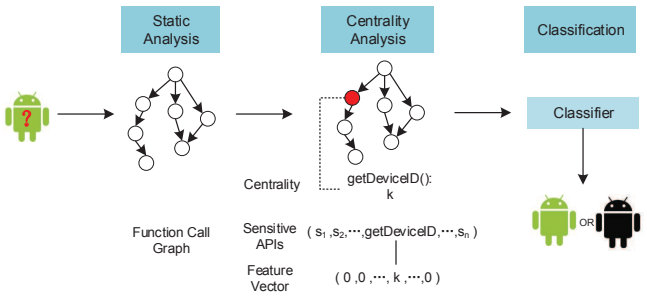


Fig. 2: System architecture of *MalScan*

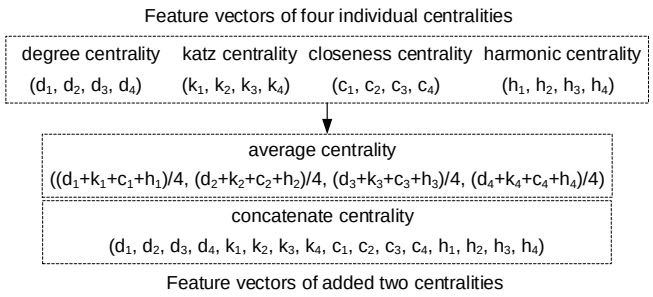


Fig. 3: Illustration of the construction of average centrality and concatenate centrality

within the graph. The output of this phase is the feature vector.

- ***Classification:*** In the final phase, given the feature vector, we can accurately and efficiently classify the app as either benign or malicious by using a machining learning classifier.

B. Static Analysis and Centrality Analysis

In this paper, we aim at proposing a graph-based market-wide malware scanning system, which requires high efficiency on app processing and graph analysis. Therefor, we conduct low-cost program analysis (*e.g.,* context- and flow-insensitive analysis) to extract succinct function call graphs based on an Android reverse engineering tool, Androguard [26].

As API calls are used by the Android apps to access operating system functionality and system resources, they can be used as representations of the behaviours of Android apps. Particularly, Android malware usually invokes some security-related API calls to perform malicious activities. For instance, *getDeviceID()* can get your phone's IMEI and *getLine1Number()* can obtain your phone number. Therefor, in order to characterize malicious behaviors, we focus on these security-related API calls, namely sensitive API calls on the basis of the results reported by PScout [24] which consist of 21,986 sensitive API calls.

In centrality analysis, we pay attention to extract the centrality of sensitive API calls. There have been proposed several definitions of centrality in a social network, for example:

- *Degree centrality* [10] of a node is the fraction of nodes it is connected to. The degree centrality values are normalized by dividing by the maximum possible degree in a simple graph N - 1 where N is the number of nodes in the graph.
$$C_D(v) = \frac{deg(v)}{N-1}$$
Note that $deg(v)$ is the degree of node v.
- *Katz centrality* [11] is a generalization of degree centrality. Degree centrality measures the number of direct neighbors, and Katz centrality measures the number of all nodes that can be connected through a path, while the contributions of distant nodes are penalized. Let A be the adjacency matrix of a graph under consideration.
$$C_K(i) = \sum_{k=1}^{\infty} \sum_{j=1}^{n} \alpha (A^k)_{ij}$$
Note that the above definition uses the fact that the element at location (i, j) of A^k reflects the total number of k degree connections between nodes i and j. The value of the attenuation factor α has to be chosen such that it is smaller than the reciprocal of the absolute value of the largest eigenvalue of A.
- *Closeness centrality* [10] of a node is the average length of the shortest path between the node and all other nodes in the graph. Its normalized form is generally given by the previous value multiplied by N-1, where N is the number of nodes in the graph.
$$C_C(v) = \frac{N-1}{\sum_y d(t,v)}$$
Note that $d(t, v)$ is the distance between nodes v and t.
- *Harmonic centrality* [18] reverses the sum and reciprocal operations in the definition of closeness centrality.
$$C_H(v) = \frac{\sum_{t \neq v} \frac{1}{d(t,v)}}{N-1}$$
Note that $d(t, v)$ is the distance between nodes v and t and N is the number of nodes in the graph.
- *Betweenness centrality* [19] quantifies the number of times a node acts as a bridge along the shortest path between two other nodes.
$$C_B(v) = \sum_{s \neq v \neq t \epsilon V} \frac{\sigma_{st}(v)}{\sigma_{st}}$$
Note that σ_{st} is total number of shortest paths from node s to node t and $\sigma_{st}(v)$ is the number of those paths that pass through v. The betweenness may be normalized by dividing through the number of pairs of nodes not including v, which for directed graphs is (n-1)(n-2) and for undirected graphs is (n-1)(n-2)/2 where n is the number of nodes in the graph.

Given a call graph, we then compute the centrality of sensitive API calls. Some sensitive API calls that are not contained in this call graph are represented as 0 in the feature vector. We select total four different centrality measures which are degree centrality, katz centrality, closeness centrality, and harmonic centrality to commence our experiments. We exclude betweenness centrality due to the low efficiency on *Centrality Analysis*, it requires about 71 seconds to complete extracting the centrality of a call graph while the number of nodes is 11,240. Additionally, it is generally to measure

the importance of a vertex in a social network by combining multiple centrality measures. Therefor, we construct another two types of centrality by integrating the four individual centrality measures: the one is to calculate the average value of the former four centrality measures (average centrality) and the other is to concatenate the former four centrality measures (concatenate centrality). Figure 3 shows the construction of these two types of centrality. In our experiments, the dimension of the feature vector of four individual centrality measures is the total number of sensitive API calls reported in PScout [24], which is 21,986. Therefor, as shown in Figure 3, the dimension of the feature vector of average centrality and concatenate centrality is 21,986 and 21,986*4=87,944, respectively.

C. Classification

Our final phase focuses on classification, *i.e.,* labeling apps as either benign or malicious. To this end, we select three different classification algorithms: 1-Nearest Neighbor (1-NN), 3-Nearest Neighbor (3-NN), and Random Forest to complete classification. These three classifiers are implemented by using a python library scikit-learn [27]. For the Random Forest, we adopt the default parameters to commence our experiments [4]. Each model is trained by using feature vectors obtained from a training dataset and then performing classification on a testing dataset. All the experimental results are presented in Section IV by performing 10-fold cross validations on our datasets.

IV. Experimental Evaluation

In this section, we conduct seven experiments to check *MalScan*'s capability on detecting Android malware. Specifically, we first evaluate the effectiveness of *MalScan* by classifying datasets that are developed during the same year. Then we examine the robustness of *MalScan* in two ways: the first is to classify newer samples by models trained on old datasets, the second is to detect adversarial samples crafted by adversarial attack. Next, we present the runtime performance of *MalScan*. Then, we validate the feasibility of *MalScan* on market-wide malware scanning. Finally, we introduce the combination of different centrality measures of *MalScan* on malware scanning and demonstrate the ability on detecting zero-day malware.

A. Datasets and Metrics

Datasets used to evaluate *MalScan* include total 30,715 samples, which are available in github[5], by this researchers can conduct reproducible experiments. We crawled these APK files from AndroZoo [23] which currently contains over nine millions APK files, and each of which has been detected by several different antivirus products in VirusTotal [16]. Our final datasets include 15,285 benign apps and 15,430 malicious apps. In addition, the time period of our datasets ranges from January 2011 to December 2018, by this we can conduct a more detailed evaluation to verify how robust *MalScan* is to changes in Android apps. Table III lists the summary of our datasets.

[4]More detailed information of parameters are available in the official website: https://scikit-learn.org/.

[5]https://github.com/malscan-android/malscan.

TABLE III: Summary of datasets used in our experiments

Dataset	Benign	Malware	Total	Average Size (MB)
2011	1,920	1,916	3,836	2.49
2012	1,875	2,000	3,875	3.73
2013	1,896	2,000	3,896	6.56
2014	1,826	1,982	3,808	7.15
2015	1,811	1,839	3,650	8.35
2016	2,015	1,940	3,955	5.12
2017	1,884	1,834	3,718	7.92
2018	2,058	1,919	3,977	8.15
Total	**15,285**	**15,430**	**30,715**	**6.17**

TABLE IV: Descriptions of used metrics in our experiments

Metrics	Abbr	Definition
True Positive	**TP**	#samples correctly classified as malicious
True Negative	**TN**	#samples correctly classified as benign
False Positive	**FP**	#samples incorrectly classified as malicious
False Negative	**FN**	#samples incorrectly classified as benign
True Positive Rate	**TPR**	TP/(TP+FN)
False Negative Rate	**FNR**	FN/(TP+FN)
True Negative Rate	**TNR**	TN/(TN+FP)
False Positive Rate	**FPR**	FP/(TN+FP)
Accuracy	**A**	(TP+TN)/(TP+TN+FP+FN)
Precision	**P**	TP/(TP+FP)
Recall	**R**	TP/(TP+FN)
F-measure	**F1**	2*P*R/(P+R)

To evaluate *MalScan*, we conduct experiments by performing 10-fold cross validations using the datasets. Additionally, we use the widely used metrics as shown in Table IV to measure the effectiveness of *MalScan*. Note that all experiments are compared with following two state-of-the-art Android malware detection systems: *MaMaDroid* and *Drebin*.

- *MaMaDroid* [8]: a state-of-the-art Android malware detection system, which leverages the sequences of abstracted function calls obtained from a call graph to build a behavioral model, and uses it to extract features to conduct classification.
- *Drebin* [4]: a state-of-the-art Android malware detection system, which performs a broad static analysis for extracting as many features as possible from an app, and embeds them in a joint vector space to classify malware.

B. Detection Effectiveness

We first evaluate how well *MalScan* on detecting malware by training and testing using samples that are developed in the same year. To this end, we conduct experiments on eight datasets as depicted in Table III by performing 10-fold cross validations. Table V presents the detection results achieved by *MalScan*, *MaMaDroid*, and *Drebin* on each dataset, respectively. The results include f-measure and accuracy for each experiment. In order to verify the effectiveness of different centrality measures on detecting Android malware, we first conduct four individual centrality experiments. Additionally, it is generally to measure the importance of a vertex in a social

TABLE V: Experimental results of *MalScan*, *MaMaDroid*, and *Drebin* classification with datasets from the same year

Dataset	2011		2012		2013		2014		2015		2016		2017		2018	
Metrics	F1	A	F1	A	F1	A	F1	A	F1	A	F1	A	F1	A	F1	A
Degree	95.5	95.4	96.7	96.5	96.4	96.2	95.6	96.4	**98.1**	**98.1**	**98.5**	**98.5**	97.0	97.0	97.9	98.0
Closeness	96.2	96.1	96.5	96.3	96.7	96.5	96.7	96.5	97.6	97.6	97.3	97.3	97.7	97.7	**98.8**	**98.8**
Harmonic	96.9	96.8	98.0	97.9	97.2	97.1	96.8	96.6	96.0	96.1	97.2	97.3	96.8	96.8	97.9	98.0
Katz	96.0	95.8	96.4	96.1	96.9	96.8	96.6	96.5	97.4	97.4	98.0	98.0	**98.4**	**98.4**	98.0	98.1
Average	97.2	97.2	97.9	97.9	97.2	97.1	96.6	96.5	96.7	96.7	97.7	97.7	96.9	97.0	98.3	98.3
Concatenate	**97.5**	**97.5**	**98.1**	**98.0**	**97.5**	**97.4**	**97.7**	**97.6**	97.6	97.6	97.7	97.7	97.1	97.1	98.4	98.5
MaMaDroid	94.4	94.2	95.6	95.3	94.3	93.9	95.9	95.6	95.6	95.4	96.0	96.0	96.5	96.5	97.3	97.2
Drebin	96.4	96.4	96.4	96.4	97.4	97.4	94.8	94.8	94.7	94.7	95.3	95.3	91.0	91.0	91.2	91.2

network by combining multiple centrality metrics. Therefor, we add another two experiments by integrating the former four individual centrality measures as shown in Figure 3. In a word, we evaluate *MalScan* by conducting six experiments on each dataset per year.

As shown in Table V, we see that for each dataset, *MalScan* can maintain a high f-measure and accuracy above 95% for all six experiments. In addition, the detection performances vary according to the selected centrality measures. For instance, the f-measure is 98.1% when we choose degree centrality to conduct classification on 2015 dataset while is 97.4% when we select katz centrality. This observation is mainly due to that the definitions are different between selected centrality measures. Particularly, the results of concatenate centrality is generally better than other centralities, which is because of the more comprehensive features in concatenate centrality (Figure 3).

Compared to *MaMaDroid*, *MalScan* can achieve better performance on all datasets in terms of accuracy. As for f-measure, it outperforms *MaMaDroid* on most of datasets except 2014 dataset. The f-measure is 95.6% when we conduct classification by using degree centrality measure on 2014 dataset, while *MaMaDroid* can achieve 95.9% f-measure. However, the maximum f-measure and accuracy of six experiments in *MalScan* is all above *MaMaDroid*. Such results indicate that *MalScan* can obtain better performance than abstraction-based approach when performing classification on dataset in the same year. This happens because the abstraction of API calls can cause some false positives, for instance, API calls *TelephonyManager.getDeviceId()* and *SmsManager.sendTextMessage()* can be abstracted into the same package and family, which are *android.telephony* and *android*, respectively.

Through the results shown in Table V, we can see that the f-measure and accuracy of another compared Android malware detection method, *Drebin*, both take up 97.4% which are almost the same as the maximum f-measure and accuracy of *MalScan* on 2013 dataset. However, it performs poorer than *MalScan* on most of datasets. In general, *MalScan* can achieve higher f-measure and accuracy on datasets from 2014 to 2018. Particularly, it significant outperforms *Drebin* on 2017 and 2018 datasets. In addition, the maximum f-measure and accuracy of six experiments in *MalScan* on all datasets are all above *Drebin*. It is reasonable that *MalScan* performs better than *Drebin* because *MalScan* considers the program structural semantics while *Drebin* ignores them.

TABLE VI: Mean values of f-measure and accuracy of *MalScan*, *MaMaDroid*, and *Drebin* to perform classification on newer samples by training an old dataset

Testing Gap	**one year**		**two year**		**three year**		**four year**	
Metrics	F1	A	F1	A	F1	A	F1	A
Degree	82.03	82.95	67.83	74.00	57.01	69.02	36.24	60.64
Closeness	80.80	82.85	62.98	72.23	53.98	67.99	37.04	60.95
Harmonic	86.19	86.91	71.34	76.69	62.35	70.97	48.48	64.33
Katz	82.43	83.17	65.65	72.16	56.43	68.52	35.48	58.95
Average	86.63	87.22	69.34	76.11	63.27	71.25	49.22	64.55
Concatenate	**88.20**	**88.37**	70.74	**77.21**	59.66	**71.59**	47.42	**64.72**
MaMaDroid	84.30	83.80	**75.76**	75.78	**67.92**	68.94	**57.31**	62.38
Drebin	82.18	83.20	71.73	74.01	66.05	70.32	39.54	59.60

In conclusion, *MalScan* can achieve better performance than *MaMaDroid* and *Drebin* on detecting malware by training and testing using samples that are developed in the same year.

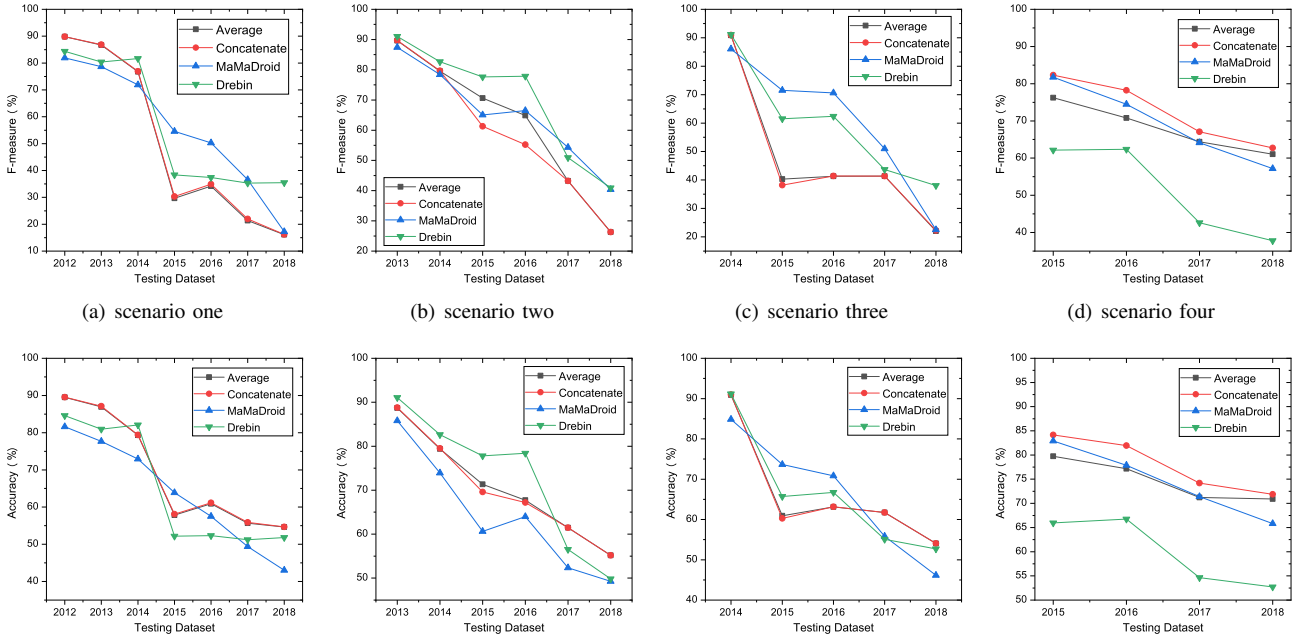
C. Robustness against Android Evolution

For testing the resilience of *MalScan* on the evolution of Android apps. We create four scenarios and conduct experiments on each scenario, the f-measure and accuracy of *MalScan*, *MaMaDroid*, and *Drebin* are presented in Figure 4. In the first scenario, each system is trained by using 2011 dataset, and then classify the samples from 2012 to 2018. In the second scenario, we use the samples randomly selected from datasets before 2012 (*i.e.,* 2011 and 2012 datasets) as the training data and test the samples from 2013 to 2018[6]. Similar to the previous scenarios, the training samples in the third scenario are randomly selected from datasets before 2013, and the testing samples are from 2014 to 2018. Our final scenario includes randomly-chosen training samples from datasets before 2014, and conduct classification on the samples in 2015, 2016, 2017, and 2018.

In order to show more clearly in figures, we only present the average and concatenate experimental results of *MalScan* in Figure 4[7]. In scenario one, as shown in Figure 4(a) and 4(e), the f-measure and accuracy of *MalScan* is above *MaMaDroid* and *Drebin* when conduct testing on 2012 and 2013 datasets. However, the f-measure and accuracy of *MalScan* and *Drebin* both drop a lot when the year of testing dataset increases to 2015. This is because the sensitive API calls between 2011

[6]To maintain the coherence of the number of these four scenarios, the number of training samples in the last three scenarios are the same as 2011 dataset, which are 1,920 benign apps and 1,916 malicious apps.

[7]More detailed results can be available in the following website: https://github.com/malscan-android/malscan.



(a) scenario one (b) scenario two (c) scenario three (d) scenario four

(e) scenario one (f) scenario two (g) scenario three (h) scenario four

Fig. 4: The f-measure and accuracy of *MalScan*, *MaMaDroid*, and *Drebin* to perform classification on newer samples by training an old dataset

and 2015 dataset vary a lot, and both *MalScan* and *Drebin* are based on the analysis of sensitive API calls. In scenario two, all the methods can both achieve high f-measure and accuracy when testing on 2013 dataset as shown in Figure 4(b) and 4(f). *Drebin* is able to maintain the best performance on datasets from 2013 to 2016. In scenario three, the f-measure and accuracy of *MalScan* and *Drebin* both drop a lot when the year of testing dataset increases from 2014 to 2015. The reason is the same as scenario one, for instance, the overlap ratio[8] of sensitive API calls invoked by malware samples between the training dataset and 2014 dataset is 80% while drop to 56% when the year of testing dataset increases to 2015. In other words, almost half of sensitive API calls in the malware samples of 2015 dataset do not appear in the malware samples of training dataset, which can cause a high false negative. In the last scenario, both *MalScan* and *MaMaDroid* can obtain better performance than *Drebin*, the f-measure and accuracy of concatenate experiment of *MalScan* are higher than *MaMaDroid* and *Drebin* on all testing datasets.

To research the overall performance on detecting newer samples by using an old dataset for training, we present the mean values of f-measure and accuracy of *MalScan*, *MaMaDroid*, and *Drebin* on detecting newer datasets whose time period ranges from one year to four year. Table VI presents the results. As for f-measure, *MalScan* is able to maintain the best performance when the time period between testing dataset and training dataset differs by one year. However, when the time period increases to two year, three year, and four year, *MaMaDroid* performs slight better than the others. This is mainly due to the abstraction of API calls, which is more resilient to the evolution of Android apps. As for accuracy, *MalScan* can achieve better performance than *MaMaDroid* and *Drebin* on detecting newer samples by training an old dataset.

In general, compared with *MaMaDroid* and *Drebin*, *MalScan* can obtain an approximate good effect on detecting newer samples by using an old dataset.

[8]Given two sets of sensitive API calls S_1 and S_2, the overlap ratio between S_1 and S_2 is defined as $O(S_1, S_2) = \frac{|S_1 \cap S_2|}{max(|S_1|,|S_2|)}$.

TABLE VII: Attack details in this evaluation

Attack terms	Descriptions
Attack Scenario	The adversary knows both the feature set and the training set, and also has access to the target system as a black box
Attack Algorithm	Modified JSMA
Attack Dataset	2011 dataset
Attack Detectors	1NN (*MalScan*), Random Forest (*MaMaDroid*), and SVM (*Drebin*)

D. Robustness against Adversarial Attack

In order to research the robustness of *MalScan* to adversarial samples, we leverage a recent and state-of-the-art adversarial attack tool [28] to complete our evaluation. It can calculate the perturbations, modify source files, and rebuild the modified APK file to craft adversarial samples of Android malware. Due to open source of all the datasets and algorithms of *MaMaDroid*, *Drebin*, and *MalScan*, the adversary has access to all datasets. As for machine learning models (*i.e.,* classifiers), they can be obtained by reimplemented the algorithms presented in corresponding papers. Therefor, our attack scenario

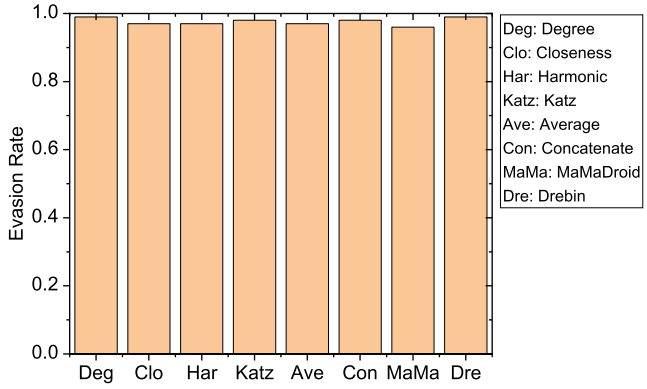


Fig. 5: The evasion rate of adversarial examples crafted by JSMA on *MalScan*, *MaMaDroid*, and *Drebin*

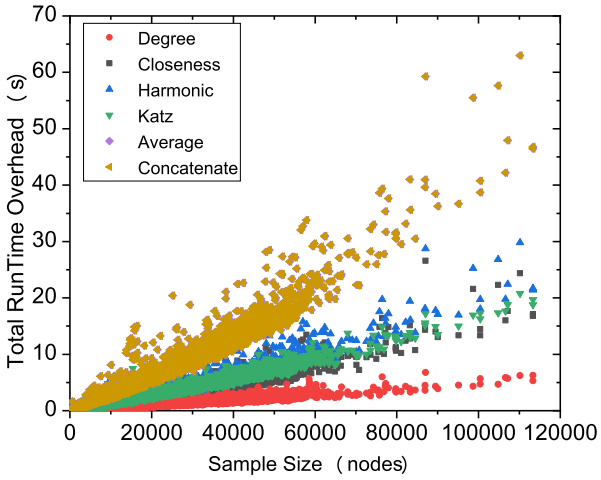


Fig. 6: Total runtime overheads of *MalScan* on different sample size by using Random Forest

in this section is that the adversary knows all datasets, and also has access to the target detector as a black box. Specifically, we choose our 2011 dataset as our test object to commence the evaluation. Moreover, the attack algorithm we select is modified JSMA [28], which crafts adversarial examples by using the forward derivatives of the classifier[9]. As described in [8], *MaMaDroid* can achieve better performance when it adopts Random Forest to detect malware. As for *MalScan*, we find that 1NN is able to maintain better effectiveness on detecting malware. Therefor, the selected machine learning classifiers in this section are 1NN, Random Forest, and SVM for *MalScan*, *MaMaDroid*, and *Drebin*, respectively. Table VII summarizes these attack details.

Figure 5 presents the evasion rate (*i.e.,* FNR) of crafted malware samples on *MalScan*, *MaMaDroid*, and *Drebin*. It shows that all the three systems can be evaded easily by these crafted adversarial samples. This happens because the attack launched by [28] is a tailor-made attack, it can make corresponding changes to the attack steps according to the different algorithms implemented by classifiers, until the crafted adversarial sample can be misclassified by the detector or exit when the number of iterations reaches the set threshold. However, the attack cost of *Drebin* is the lowest, since it only extracts syntax features and the adversary only needs to add code containing the required features (*e.g.,* restricted API calls) but never being invoked or executed. As for *MaMaDroid* and *MalScan*, it requires more cost for the adversary to complete the attack due to the consideration of program semantics. For instance, the added features are not the simple restricted API calls but calls from some callers to some callees. Moreover, when the adversary attacks on average or concatenate centrality detector, the attack cost is more since they are constructed by integrated four individual centrality measures. Therefor, the adversary must consider four different algorithms of centrality extraction for crafting adversarial samples.

In conclusion, *MalScan*, *MaMaDroid*, and *Drebin* are not robust enough in the face of tailor-made adversarial attack.

[9]More detail procedures are in [28].

E. Runtime Overhead

In this section, we estimate the runtime overhead of *MalScan*, *MaMaDroid*, and *Drebin* by using a dataset randomly selected from our 30,715 samples, which consists of 3,000 benign apps and 3,000 malicious apps. The average number of nodes in these 3,000 benign samples and 3,000 malicious samples are 17,669 and 12,991, respectively.

Given a new app, *MalScan* consists of three main phases to analyze it: (1) Function call graph extraction, (2) Feature extraction, and (3) Classification. The runtime overheads of these three main phases are illustrated in Table VIII. It is required an average of 0.67 seconds to construct the call graph for a given APK file. In feature extraction, the runtime overhead differs according to the selected centrality measures. On average, 0.03 seconds is required for degree centrality extraction, which is considerably less than the other three centrality measures. To extract the average centrality and concatenate centrality of a node within a graph, we first need to obtain the four individual centrality values (Figure 3). Therefor, it takes both about an average of 4.26 seconds to construct the feature vector for average centrality and concatenate centrality. Given a feature vector, we can perform classification by using a trained machine learning model[10]. As shown in Table VIII, the runtime overhead of classification varies according to not only the selected centrality measures but also the selected classification algorithms. As for centrality measures, the runtime overhead of concatenate centrality is higher than the other five centrality measures. This happens due to that the dimension of feature vector of concatenate centrality is four times longer than the other five centrality measures.

Moreover, as shown in Figure 6, the total running time to process an app by *MalScan* is generally positive related to the

[10]In this evaluation, we trained machine learning models by using these 6,000 apps.

TABLE VIII: The average runtime overhead (seconds) of *MalScan*, *MaMaDroid*, and *Drebin* on different phases

Phases	Graph Construction	Feature Extraction	Classification			
			1NN	3NN	RandomForest	SVM
Degree	0.6654	0.0293	0.0093	0.0766	0.0015	—
Closeness		1.1007	0.0093	0.0886	0.0014	
Harmonic		1.6275	0.0139	0.0697	0.0015	
Katz		1.4984	0.0110	0.1102	0.0014	
Average		4.2560	0.0139	0.0731	0.0014	
Concatenate		4.2556	0.0456	0.2478	0.0016	
MaMaDroid	163.1773	2.4562	0.4867	1.4501	0.0007	—
Drebin	—	82.3846	—			0.3090

size of app's function call graph. The more nodes in a call graph, the longer the running time. Overall, given a new app, as shown in Figure 7, *MalScan* can classify it as either benign or malicious in an average of 0.7 seconds when we select degree centrality to construct the feature vector and Random Forest as the classification model.

We also evaluate the runtime overhead of *MaMaDroid* and *Drebin* on detecting an app. As for *MaMaDroid*, it consists of three main procedures to complete the classification. In the first step, for constructing a more precise call graph of an app, *MaMaDroid* performs heavyweight program analysis to ensure contexts and flows preserved. So it takes an average of 163.18 seconds to extract the call graph per app in our 6,000 samples. Whereas, it takes about 2.46 seconds to finish the feature extraction step per app. In the final step, classification using Random Forest is fastest: 0.0007 seconds on average. Therefor, as shown in Figure 7, in total, *MaMaDroid* consumes about 165.63 seconds for a complete classification on an app from our 6,000 samples when we select Random Forest as the classification model. As for *Drebin*, on the one hand, it extracts features not only from disassembled code but also from the manifest. On the other hand, these features include several complex features (*e.g.,* network address) and the number of extracted features is more than 90,000. Therefor, it takes about 82.38 seconds on average to extract features per app in our 6,000 samples. Given a feature vector, *Drebin* consumes about 0.31 seconds to flag it as either benign or malicious by using a SVM classifier.

In conclusion, *MalScan* is tremendously scalable and efficient than *MaMaDroid* and *Drebin*.

F. Market-wide Case Study

TABLE IX: The number and average size of apps collected from Google-Play app market

Year	#Apps	Average Size (MB)
2011	79,192	2.09
2012	158,402	3.05
2013	350,736	4.35
2014	1,025,021	6.04
2015	524,026	8.69
2016	994,635	9.79
2017	207,280	11.01
2018	198,303	12.17
Total	**3,537,595**	**7.73**

To validate the feasibility of *MalScan* on malware scanning of Google-Play scale Android app stores, we conduct

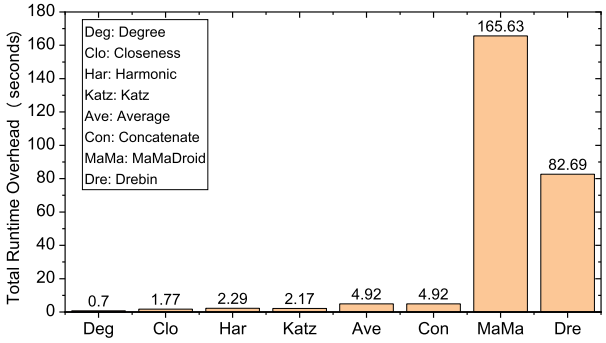


Fig. 7: Total average runtime overheads of *MalScan* by using Random Forest, *MaMaDroid* by using Random Forest, and *Drebin* by using SVM

a statistical research of app size in Google-Play app market from AndroZoo [23]. We collect some information of apps in Google-Play app market which includes sha256, package name, apk size, and dex date. As shown in Table IX, the time period of these apps is from January 2011 to December 2018, and the total number of collected apps in Google-Play app market is 3,537,595[11]. Table IX presents the average size of collected apps in different year, it can be seen that the average size of apps will grow larger over time, and the average size of total 3,537,595 apps is 7.73 MB.

We also introduce the *Cumulative Distribution Functions* (CDFs) of the size of these apps in Google-Play app market and our randomly selected 6,000 apps. As shown in Figure 8, in general, apps in Google-Play app market are slightly larger than apps in *MalScan* (randomly selected 6,000 apps). Because *MalScan* is a graph-based malware scanning system, and the total running time to process an app is generally positive related to the size of app's function call graph (Figure 6). Therefor, we randomly select 6,000 apps from collected 3,537,595 apps and conduct static analysis to extract the function call graphs. After obtaining the 6,000 function call graphs, we then gather the size of these graphs. Specifically, from the results presented in Table X, the average graph size of apps in Google-Play app market is about 1.44 times larger than in *MalScan*. In addition, the ratio of average running time for *MalScan* to complete classification on Google-Play_6000

[11]All the collected information of these 3,537,595 apps can be available in the following website: https://github.com/malscan-android/malscan.

TABLE X: The average size of apps in Google-Play app market and apps used in *MalScan*, average runtime overheads to complete classification on these apps by using Random Forest and the size of ratio between Google-Play_6000 and MalScan_6000

Apps	Average Size (MB)	Average Size (Nodes)	Average RunTime Overheads (second)					
			Degree	Closeness	Harmonic	Katz	Average	Concatenate
Google-Play_3537595	7.73	—	—					
Google-Play_6000	7.53	22,143	0.9983	2.5761	3.3961	3.1121	7.2068	7.2052
MalScan_6000	6.29	15,330	0.6962	1.7676	2.2926	2.1652	4.9228	4.9226
Ratio (GP_6000/MS_6000)	1.1971	1.4444	1.4339	1.4574	1.4813	1.4373	1.4640	1.4637

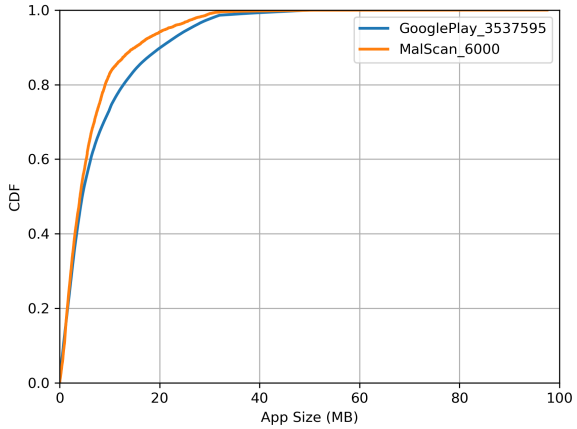


Fig. 8: CDFs of the app size (MB) in Google-Play app market (3,537,595 apps) and *MalScan* (6,000 apps)

and MalScan_6000 is almost around 1.44. Such result also indicates that the total running time of *MalScan* to process an app is generally positive related to the size of app's function call graph. Therefor, when we adopt *MalScan* to perform malware scanning on Google-Play app market, the average runtime overhead to process an app may be around one second when we select degree centrality to form the feature vector. Such high efficiency suggests that *MalScan* can enable frequent market-wide scanning of Google-Play scale Android app markets.

G. Combination of Centrality Measures

In this section, we propose the combination of different centrality measures on detecting Android malware. As discussed in the former experiments, we totally select four individual centrality measures and add another two centrality measures (Figure 3). In reality, these six experimental results can be complementary, for instance, we can use majority-voting to flag an app as either benign or malicious. In other words, an app we consider as malware when it is reported to be malicious by one or more of the six centrality experiments. For testing the feasibility of majority-voting, we conduct an experiment on scenario one in Section IV.C. We leverage a trained model by using 2011 dataset and test on datasets from 2012 to 2014, respectively. The thresholds to flag an app as malicious in our experiment are one (Vote_1), two (Vote_2), three (Vote_3), four (Vote_4), five (Vote_5), and all (Vote_6).

TABLE XI: The f-measure and true positive rate (TPR) of *MalScan* by adopting majority-voting, *MaMaDroid* and *Drebin* to perform classification on 2012-2014 datasets by using 2011 dataset for training

TestingYear	2012		2013		2014	
Metrics	F1	TPR	F1	TPR	F1	TPR
Degree	85.23	83.10	83.07	75.45	74.37	60.54
Closeness	82.93	74.70	73.83	63.05	68.96	54.59
Harmonic	89.73	89.35	86.32	83.10	76.54	65.84
Katz	86.25	85.15	79.43	74.35	76.66	65.29
Average	89.80	89.35	86.70	82.95	76.77	65.69
Concatenate	89.83	89.40	86.88	83.10	76.93	65.94
Vote_1	88.47	**96.90**	85.20	**92.00**	82.31	**81.99**
Vote_2	**91.44**	92.65	**88.80**	86.45	**84.07**	74.97
Vote_3	90.48	90.20	87.30	84.40	76.82	66.15
Vote_4	89.32	85.75	81.98	72.80	76.79	64.18
Vote_5	80.43	68.85	73.76	60.00	67.09	51.11
Vote_6	68.86	52.90	66.64	51.00	51.16	34.46
MaMaDroid	81.94	80.18	78.68	78.79	71.90	68.33
Drebin	84.37	71.90	80.37	63.60	81.64	65.69

As shown in Table XI[12], when we select two as the threshold to flag an app as either benign or malicious (*i.e.,* an app we consider as malware when it is reported to be malicious by two or more of the six centrality experiments), the f-measure is highest among all the experimental results. Particularly, the *True Positive Rates* (TPRs) improve significantly when we adopt majority-voting to detect malware. When we train a model by using 2011 dataset and perform classification on 2012 dataset, the true positive rate is at most 89.40% if we select concatenate centrality to form the feature vector. However, the true positive rate can increase to 96.90% when we adopt majority-voting to flag an app as either benign or malicious. In other words, we can detect at most 96.90% malware in 2012 dataset when we employ majority-voting. In addition, parallel processing is a feasible choice when we conduct six centrality experiments. Therefor, the runtime overhead of majority-voting can be only slightly longer than concatenate centrality experiment.

H. Detection of Zero-day Malware

To validate the capability of *MalScan* on detecting real-world zero-day malware, we use our 2018 dataset to train classifiers by adopting 1NN algorithm (we totally obtain six classifiers according to the six different centrality measures).

[12]Due to the limited page, other detailed results are available in the website: https://github.com/malscan-android/malscan.

Next, we crawl 5,000 apps from Google-Play app market and feed them to the trained classifiers. We leverage majority-voting method to flag an app as benign or malicious, and we consider an app as malware when it is reported to be malicious by two or more of the six centrality experiments. Among these apps, *MalScan* reports 22 of them as being malicious. To investigate whether these 22 apps are malware, we upload them to VirusTotal [16] to analyze each of them. Among these 22 apps, 17 of them are reported as malware by at least one antivirus scanner. Of the remaining 5 apps, we manually inspect them. Our manual inspection shows that 1 of these 5 apps contains highly suspicious behaviours, it contains 28 dangerous-level permissions, reads device's memory and CPU information, and writes many sensitive data into several log files. In an effort to check more deeply, we leverage a state-of-the-art Android app analysis system which combines static and dynamic analysis for reporting detailed risky behaviours [29]. From the reported results, we can see that the app executes shell code which can demonstrate that it is indeed a malware.

In conclusion, *MalScan* is able to find 18[13] zero-day malware among 5,000 Google-Play apps, 1 of them is not reported as malware by existing tools [16].

V. Discussion

In our work, we totally select four individual centrality measures and add another two centrality measures to perform classification on Android apps. We plan to test the capability of more different centrality measures on detecting Android malware. Although the robustness of *MalScan* against tailor-made adversarial attack is low, it can be used as the first line of defense because of the high efficiency on malware scanning. After filtering most of malware, other more computational intensive and robust approaches can be used as the second line of defense. By this we can save more times and resources. Moreover, since most Android malware detection systems are closed source, we only compare *MalScan* with two open source systems (*i.e., MaMaDroid* and *Drebin*). We will conduct detailed comparative analysis on more systems in our future work.

VI. Related Work

There has been many proposed approaches on Android malware detection that rely on syntax features [1], [2], [3], [4], [30], [31], [32] or program semantics [5], [6], [8], [33], [34], [35], [36], [37], [38], [39], [40], [41], [42]. *Drebin* [4] uses a broad static analysis to extract as many features as possible from an app, and embeds them in a joint vector space to classify malware. However, it only searches for the presence of particular strings, such as some restricted API calls, rather than considers the program semantics. So it can be easily evaded by attacks on syntax features [28], [43]. *DroidAPIMiner* [2] conducts frequency analysis to identify certain API calls commonly used by malware and then performs a simple data flow analysis to extract features to complete classification. Unfortunately, it suffers from feature explosion because it cannot generalize its feature space.

DroidSIFT [5] extracts the weighted contextual API dependency graph to solve the malware deformation problem based on static analysis. *Apposcopy* [6] utilizes static analysis to extract the data-flow and control-flow properties of an app to identify its malware family. However, both *DroidSIFT* [5] and *Apposcopy* [6] suffer from heavy runtime overhead, they consume 175.8s and 275s to analyze an app, respectively. *MaMaDroid* [8] leverages the sequences of abstracted function calls obtained from a call graph to build a behavioral model and uses it to extract features to conduct classification. This approach is more resilient to API changes and is more robust to the evolution of Android apps. However, it sustains some limitations, the one is that it can be easily evaded by the self-defined packages that looks similar to Android's, Google's or Java's packages [28], the other is that it requires a sizable amount of memory on classification because of its large feature sets and the extraction of call graph [8]. Different from the previous work, to avoid the heavy computation overhead of graph matching, *MalScan* regards the function call graph of an app as a complex social network and then extracts the centrality of sensitive API calls to construct the feature vector. Given a feature vector, *MalScan* can accurately flag it as either benign or malicious.

VII. Conclusion

In this paper, we present a lightweight Android malware detection method based on centrality analysis of sensitive API calls. We implement an automated Android malware detection system, *MalScan*, and conduct a comprehensive evaluation in datasets of 30,715 apps. Experimental results indicate that *MalScan* is capable of detecting Android malware in an average of 0.7 seconds with up to 98% accuracy, which is more than 100 times faster than two state-of-the-art approaches, namely *MaMaDroid* and *Drebin*. We also demonstrate the feasibility of *MalScan* on market-wide mobile malware scanning by performing a statistical study on over 3 millions apps from Google-Play app market. Moreover, in a corpus of dataset collected from Google-Play app market, *MalScan* is able to identify 18 zero-day malware including malware samples that can evade detection of existing tools in VirusTotal [16].

Acknowledgements

We would thank the anonymous reviewers for their insightful comments to improve the quality of the paper. We are also grateful to Xiao Chen for his code sharing of his attack tool. This work is supported by the National Key Research and Development Plan of China under Grant 2017YFB0802205, by the Shenzhen Fundamental Research Program under Grant No. JCYJ20170413114215614, by the Guangdong Provincial Science and Technology Plan Project under Grant No. 2017B010124001 and the Guangdong Provincial Key R&D Plan Project under Grant No. 2019B010139001.

[13]Detailed information can be available in the following website: https://github.com/malscan-android/malscan.

REFERENCES

[1] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Using probabilistic generative models for ranking risks of android apps," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12)*, 2012.

[2] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android," in *Proceedings of the 9th International Conference on Security and Privacy in Communication Systems (SecureComm'13)*, 2013.

[3] W. Wang, X. Wang, D. Feng, J. Liu, Z. Han, and X. Zhang, "Exploring permission-induced risk in android applications for malicious application detection," *IEEE Transactions on Information Forensics and Security*, 2014.

[4] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket," in *Proceedings of the 2014 Annual Network and Distributed System Security Symposium (NDSS'14)*, 2014.

[5] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "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'14)*, 2014.

[6] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of android malware through static analysis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*, 2014.

[7] G. Suarez-Tangil and G. Stringhini, "Eight years of rider measurement in the android malware ecosystem: evolution and lessons learned," *arXiv preprint arXiv:1801.08115*, 2018.

[8] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini, "Mamadroid: Detecting android malware by building markov chains of behavioral models," in *Proceedings of the 2017 Annual Symposium on Network and Distributed System Security (NDSS'17)*, 2017.

[9] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu, "Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale," in *Proceedings of the 24th USENIX Security Symposium (USENIX Security'15)*, 2015.

[10] L. C. Freeman, "Centrality in social networks conceptual clarification," *Social networks*, 1978.

[11] L. Katz, "A new status index derived from sociometric analysis," *Psychometrika*, 1953.

[12] H. Jeong, S. P. Mason, A.-L. Barabási, and Z. N. Oltvai, "Lethality and centrality in protein networks," *Nature*, 2001.

[13] X. Liu, J. Bollen, M. L. Nelson, and H. Van de Sompel, "Co-authorship networks in the digital library research community," *Information Processing & Management*, 2005.

[14] R. Guimera, S. Mossa, A. Turtschi, and L. N. Amaral, "The worldwide air transportation network: Anomalous centrality, community structure, and cities' global roles," *National Academy of Sciences*, 2005.

[15] N. Coles, "It's not what you know-it's who you know that counts. analysing serious crime groups as social networks," *British Journal of Criminology*, 2001.

[16] "Virustotal - free online virus, malware and url scanner," https://www.virustotal.com/, 2019.

[17] K. Faust, "Centrality in affiliation networks," *Social Networks*, 1997.

[18] M. Marchiori and V. Latora, "Harmony in the small-world," *Physica A: Statistical Mechanics and its Applications*, 2000.

[19] L. C. Freeman, "A set of measures of centrality based on betweenness," *Sociometry*, 1977.

[20] M. Piraveenan, M. Prokopenko, and L. Hossain, "Percolation centrality: Quantifying graph-theoretic impact of nodes during percolation in networks," *PloS one*, 2013.

[21] M. R. Faghani and U. T. Nguyen, "A study of xss worm propagation and detection mechanisms in online social networks," *IEEE Transactions on Information Forensics and Security*, 2013.

[22] A. Alvarez-Socorro, G. Herrera-Almarza, and L. González-Díaz, "Eigencentrality based on dissimilarity measures reveals central nodes in complex networks," *Scientific Reports*, 2015.

[23] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th Working Conference on Mining Software Repositories (MSR'16)*, 2016.

[24] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: Analyzing the android permission specification," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12)*, 2012.

[25] D. C. Howell, *Statistical methods for psychology*, 2009.

[26] A. Desnos, "Androguard," https://github.com/androguard/androguard, 2011.

[27] "scikit-learn," https://scikit-learn.org/, 2019.

[28] X. Chen, C. Li, D. Wang, S. Wen, J. Zhang, S. Nepal, Y. Xiang, and K. Ren, "Android hiv: A study of repackaging malware for evading machine-learning detection," *IEEE Transactions on Information Forensics and Security*, 2019.

[29] "Sandroid - an automatic android application analysis system," http://sanddroid.xjtu.edu.cn/, 2019.

[30] B. P. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Android permissions: A perspective combining risks and benefits," in *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies (ACMT'12)*, 2012.

[31] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets." in *Proceedings of the 2012 Annual Symposium on Network and Distributed System Security (NDSS'12)*, 2012.

[32] J. Li, L. Sun, Q. Yan, Z. Li, W. Srisa-an, and H. Ye, "Significant permission identification for machine-learning-based android malware detection," *IEEE Transactions on Industrial Informatics*, 2018.

[33] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, "Appcontext: Differentiating malicious and benign mobile app behaviors using context," in *Proceedings of the 37th International Conference on Software Engineering (ICSE'15)*, 2015.

[34] J. Allen, M. Landen, S. Chaba, Y. Ji, S. P. H. Chung, and W. Lee, "Improving accuracy of android malware detection with lightweight contextual awareness," in *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC'18)*, 2018.

[35] W. Yang, M. Prasad, and T. Xie, "Enmobile: Entity-based characterization and analysis of mobile malware," in *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*, 2018.

[36] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden, "Mining apps for abnormal usage of sensitive data," in *Proceedings of the 37th International Conference on Software Engineering (ICSE'15)*, 2015.

[37] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems*, 2014.

[38] A. Machiry, N. Redini, E. Gustafson, Y. Fratantonio, Y. R. Choe, C. Kruegel, and G. Vigna, "Using loops for malware classification resilient to feature-unaware perturbations," in *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC'18)*, 2018.

[39] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli, "Madam: Effective and efficient behavior-based android malware detection and prevention," *IEEE Transactions on Dependable and Secure Computing*, 2018.

[40] A. Narayanan, M. Chandramohan, L. Chen, and Y. Liu, "A multi-view context-aware approach to android malware detection and malicious code localization," *Empirical Software Engineering*, 2018.

[41] Y. Feng, O. Bastani, R. Martins, I. Dillig, and S. Anand, "Automated synthesis of semantic malware signatures using maximum satisfiability," in *Proceedings of the 2017 Annual Symposium on Network and Distributed System Security (NDSS'17)*, 2017.

[42] J. Garcia, M. Hammad, and S. Malek, "Lightweight, obfuscation-resilient detection and family identification of android malware," *ACM Transactions on Software Engineering and Methodology*, 2018.

[43] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel, "Adversarial examples for malware detection," in *Proceedings of the 2017 European Symposium on Research in Computer Security (ESORICS'17)*, 2017.