MalScan: Android Malware Detection Based on Social-Network Centrality Analysis

Yueming Wu, Wenqi Suo, Siyue Feng, Deqing Zou, Wei Yang, Yang Liu, and Hai Jin, *Fellow*, *IEEE*

Abstract—Malware scanning of an app market is expected to be scalable and effective. However, existing approaches use syntax-based features that can be evaded by transformation attacks or semantic-based features which are usually extracted by expensive program analysis. Therefore, to address the scalability challenges of traditional heavyweight static analysis, we propose a graph-based lightweight approach *MalScan* for Android malware detection. *MalScan* considers the function call graph as a complex social network and employs centrality analysis on sensitive *application program interfaces* (APIs) to express the semantic characteristics of the graph. On this basis, machine learning algorithms and ensemble learning algorithms are applied to classify the extracted features. We evaluate *MalScan* on datasets of 104,892 benign apps and 108,640 malwares, and the results of experiments indicate that *MalScan* outperforms six state-of-the-art detectors and can quickly detect Android malware with an f-value as high as 99%. In addition, there are also significant improvements in the robustness of Android app evolution and robustness to obfuscation. Finally, we conduct an exhaustive statistical study of over one million applications in the Google-Play app market and successfully identify 498 zero-day malware, which further validates the feasibility of *MalScan* on market-wide malware scanning.

Index Terms—Android Malware, Lightweight Feature, API Centrality, Market-Wide

1 Introduction

The openness of the Android system significantly enriches the capabilities of Android smartphones and propelled the rapid expansion of the Android market. However, the openness of Android applications also makes it a prime target for malicious software attacks. In 2022, Kaspersky's mobile products and technology detected a total of 1,661,743 malicious installation packages, which is a decrease of 1,803,013 compared to the previous year [1]. While the quantity of Android malware has shown a decline, research indicates that attacks are becoming increasingly intricate in terms of both malware functionality and delivery methods. As the preferred platform for users to download Android applications, application markets are being exploited by attackers to widely disseminate malicious software and infect users' devices. Consequently, conducting comprehensive scans for mobile malware across the entire market is an urgent necessity to curb the rapid proliferation of malicious software.

Existing Android malware detection techniques fall into two main categories: dynamic analysis and static analysis. Current methods of detecting malware based on dynamic analysis [2], [3], [4], [5], [6] generally execute the application in a sandbox, thereby tracking and monitoring the behavior of the application. Dynamic analysis allows for a more accurate acquisition of program features and behaviors, but it is time-consuming and requires a lot of system resources. In addition, the APIs in Android malware may cause harm to the computer during the execution of dynamic analysis. Malware detection methods based on static analysis can be primarily categorized into two main types: syntax-based and semantic-based. Syntax-based approaches [7], [8], [9], [10], [11] enable efficient Android malware detection, for instance, *Drebin* [8] extracts information such as applied permissions and specific APIs to detect malware. However, since syntax analysis only retrieves specific strings (*e.g.*, certain sensitive APIs) and does not consider program semantics, many attack techniques can easily circumvent program behavior through obfuscation and encryption. To overcome the lack of syntax analysis leading to misclassification, semantic-based approaches [12], [13], [14], [15], [16] detect malware by distilling the program's semantics into graph form through graph matching. Due to the large size of most applications, such analysis is time-consuming and difficult to apply on a large scale. In addition, these techniques perform graph matching by similarity to existing malware graphs, which has poor system performance for new malware [14], making it difficult to perform large-scale detection in today's world of evolving malware. *MaMaDroid* [15] makes an effort to employ coarse-grained information, divides the graph into subgraphs, and significantly increases the detection's robustness. However, due to the high memory usage and lengthy processing, it is not appropriate for malware to scan an application market.

In summary, all of these methods have limitations and are difficult to apply widely in the malware detection market. To address these limitations, our previous work [17]

- *Y. Wu, W. Suo, S. Feng, and D. Zou are with National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, 430074, China. S. Feng and D. Zou are corresponding authors. E-mails: {fengsiyue, deqingzou}@hust.edu.cn.*
- *Y. Wu and D. Zou are also with Jinyinhu Laboratory, Wuhan, 430074, China.*
- *W. Yang is with University of Texas at Dallas, United States.*
- *Y. Liu is with Nanyang Technological University, Singapore.*
- *H. Jin is with National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China.*

proposes a semantic-based lightweight approach, which initiates by extracting succinct function call graphs, followed by centrality analysis on sensitive API calls. This approach not only ensures semantic information retention but also significantly reduces the time required for graph analysis. This paper is an extension of our previous work, and we supplement our present work with a substantial amount of new material.

Firstly, our previous work utilized more than 20,000 sensitive APIs. However, during the analysis of feature vectors, we discover that only a part of sensitive APIs are meaningful. To reduce feature dimensions and minimize memory and space consumption, we select the most meaningful sensitive APIs as features. Secondly, to improve the accuracy of malware detection, we expand the training data set, encompassing 101,913 benign applications and 109,945 malware. Thirdly, to underscore the significance of vertices in social networks, we add three centrality analysis algorithms (*i.e.*, eigenvector centrality, pagerank centrality, and authority centrality) to extract structural features of the graph. Additionally, we employ four machine learning algorithms (*i.e.*, Adaboost, Decision Tree, GBDT, and XGBoost) to extend the classification algorithm. Additionally, we introduce the concept of ensemble learning to integrate predictions from multiple classifiers, thereby constructing an exceptional and comprehensive strong supervised model. Finally, we also incorporate evaluations of adversarial code obfuscation and case study experiments to provide a more comprehensive assessment of *Malscan*'s performance.

In summary, we make enhancements to our prior work to establish a comprehensive and stable market-wide malware scanning system, *MalScan* [17], which preserves semantic features. We consider function call graphs as intricate social networks and employ centrality analysis of sensitive APIs to extract semantic information effectively. In experimentation, we randomly select datasets spanning the years 2017 to 2021 from *AndroZoo* [18], evaluating *MalScan*'s detection performance from five distinct angles. In addition, we compare *MalScan* to six advanced Android malware detection tools to provide a more intuitive assessment of *MalScan*'s detection capabilities. Finally, we scan millions of applications on Google-Play application market with *MalScan* and successfully identify 498 instances of zero-day malware. This accomplishment solidly attests to *MalScan*'s competence in real-world malicious software detection.

In conclusion, we make the following contributions:

- We propose a novel approach for identifying Android malware, achieved through conducting centrality analysis on critical API calls within the application's function call graph.
- We devise an efficient and accurate system *MalScan* for Android malware detection. Based on the original work, we further improve the effectiveness and robustness of detection through ensemble learning algorithms.
- We conduct comparative evaluations on a dataset of 104,892 benign apps and 108,640 malicious apps. Experiments indicate that *MalScan* is superior to *Drebin* [8], *MaMaDroid* [15], *HomDroid* [19], *Xmal* [20], *RAMDA* [21], *MSDroid* [22].
- We scan millions of applications on the Google-Play application market with *Malscan* and recognize 498 zero-day malware samples. They are downloaded more than 97 million times. We have reported them to Google and hope they can cope with the issue as soon as possible.

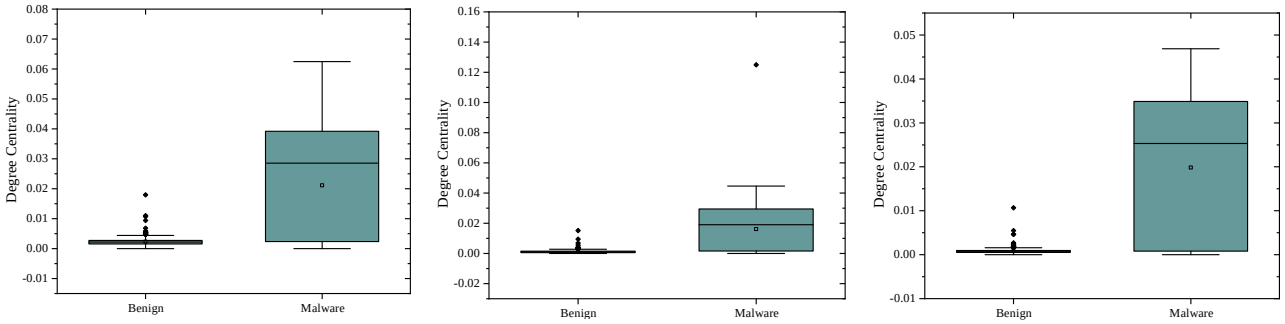
Paper organization. The remainder of the paper is organized as follows. Section 2 presents the preliminary study on the degree centrality of Android apps. Section 3 introduces our system. Section 4 reports the experimental results. Section 5 discusses future work. Section 6 describes the related work. Section 7 concludes the present paper.

2 Preliminary Study of Centrality

Social network is a network system composed of social relationships between individual members of a society. The source code of an application consists of a series of functions with diverse call relationships between them. Therefore, the application's function call graph can be perceived as a social network, likening functions to distinct members and call connections to social interactions. Centrality is a commonly used concept in social network analysis. It refers to the extent to which the nodes in a social network are centered in the whole network and serves as an indicator for evaluating the importance and influence of the nodes in the network. Centrality analysis assists in identifying pivotal elements and their roles across various systems, and it is widely employed in network analysis across diverse fields [23], [24], [25], [26], [27]. Different interpretations of the importance of nodes lead to diverse metrics for determining centrality. As a result, many different centrality measures have been proposed, such as degree centrality [28], closeness centrality [28], katz centrality [29], betweenness centrality [30], pagerank centrality [31], percolation centrality [32], cross-clique centrality [33], dissimilarity-based centrality [34].

In social networks, centrality is a measure of the importance or influence of nodes within the network structure. In the context of sensitive API calls, benign software and malware may exhibit distinct behavioral patterns. Benign software typically employs sensitive APIs extensively to perform legitimate, normal operations. This results in a relatively dispersed distribution of their sensitive API calls, covering multiple functional domains, thereby leading to lower centrality within the network. In contrast, malware may have more specialized objectives, selectively invoking sensitive APIs to carry out malicious activities. Consequently, these API calls become concentrated in specific functionalities, resulting in higher centrality of malware within the network. Building upon this observation, we propose a concept: *Analyzing the centrality of sensitive API calls can unveil distinctive behavioral patterns that differentiate benign software from malicious entities.*

To verify the suggested hypothesis, we initially choose a random sample of 500 benign applications and 500 malicious applications from *AndroZoo* [18] and extract the call graphs. Following this, we perform centrality analysis specifically on sensitive API nodes (according to a security-sensitive method list [35]). Subsequently, we perform initial frequency analysis on the top 10 frequently utilized sensitive APIs, aiming to assess whether centrality analysis could effectively distinguish intrinsic differences between benign and malicious applications. Given page constraints, we provide a partial representation of the results within Figure 1. As shown in Figure 1, it becomes apparent that there



(a) API_1: Method.invoke() (b) API_2: Class.forName() (c) API_3: Class.getDeclaredMethod()

Fig. 1: Distributions of sensitive API calls' degree centrality

exists a notable disparity in the degree centrality of sensitive API calls between malicious and benign applications. This robustly underscores the viability of employing centrality analysis on sensitive APIs for malicious software detection. Building upon this insight, we leverage centrality analysis on sensitive API calls within the call graph to develop a lightweight Android malicious software detection system.

3 System Architecture

3.1 System Overview

Illustrated in Figure 2, *MalScan* involves four main phases: *Static Analysis*, *Centrality Analysis*, *Classification*, and *Ensemble Learning*.

- ***Static Analysis:*** The objective of this phase is to derive the function call graph of an application through static analysis. The input of this phase is the apps and the output is the corresponding function call graphs.
- ***Centrality Analysis:*** This phase aims to compute the centrality of sensitive API calls within the graph to obtain the corresponding feature vectors. The input of this phase is the function call graph and the output is the corresponding feature vectors.
- ***Classification:*** This phase employs machine learning classifiers to categorize the application into benign or malicious. The input of this phase is the feature vectors and the output is the prediction results of the classifier.
- ***Ensemble Learning:*** This phase aims to integrate the prediction results of multiple classifiers and improve the accuracy of the model. The input of this phase is the prediction results for different classifiers and the output is the final prediction result of Ensemble Learning.

3.2 Static Analysis and Centrality Analysis

We design a market-wide graph-based malicious software detection system. Aiming to achieve a lightweight objective, our research focuses on efficiently accomplishing application processing and graph analysis. Therefore, during the process of static analysis, we employ the Android reverse analysis tool *Androguard* [36] to extract succinct function call graphs. The analysis approach of *Androguard* centers around lightweight objectives, primarily focusing on context-insensitive and flow-insensitive analysis. This signifies that it doesn't need to consider the specific code execution order and contextual details, thereby reducing the complexity and cost of analysis. Given that Android applications interact with the operating system's functionalities and system resources through API calls, these API invocations convey the behavior of Android applications. In particular, malicious Android software frequently exploits certain security-related API calls to carry out their malicious behavior. For example, *getLine1Number()* can retrieve phone numbers, and *getLastKnownLocation()* can access geographical location information. Hence, when describing the characteristics of malicious behavior, we place a strong emphasis on sensitive API calls.

However, due to the vast number of sensitive APIs, a large portion of irrelevant APIs not only wastes computational resources but also reduces detection efficiency. According to the experiments in the recent work [35], tracking all APIs and tracking only the selected 426 highly representative security-related sensitive APIs resulted in precision/recall rates of 91.6%/90.2% and 96.8%/93.7%, respectively. Tracking fewer APIs achieved higher precision and recall compared to tracking all APIs. This improvement can be attributed to the fact that most APIs are seldom or rarely invoked by Android applications, and an excessive number of features leads to model overfitting. Therefore, we select 426 highly representative security-related sensitive APIs, as identified in the study, to accurately characterize malicious behavior. It consists of three different API call sets. The first API call set is the top 260 API calls with the highest correlation with malware, the second API call set is 112 API calls that relate to restrictive permissions, and the third API call set is 70 API calls that are relevant to sensitive operations. Finally, 426 sensitive API calls are obtained by computing the union set of these three API call sets[1].

In Section 2, we demonstrate the feasibility of centrality analysis of function call graphs to extract semantic features. Therefore, we adopt different centrality analysis methods for sensitive APIs to achieve efficient graph analysis. We select seven different centralities to commence our experiments.

- *Degree centrality* [28] refers to the number of edges directly connected to a node, which is essentially the node's degree. The degree centrality values are normalized by dividing them by the maximum potential degree (*i.e.,* N - 1) in a simple graph, where N represents the number of nodes in the graph.
$$C_D(v) = \frac{deg(v)}{N-1} \quad (1)$$
Note that $deg(v)$ refers to the degree of node v.
- *Closeness centrality* [28] of a node measures the average shortest path length from the node to other nodes. The

1. The detailed list of the 426 critical framework APIs can be accessed at https://apichecker.github.io/.

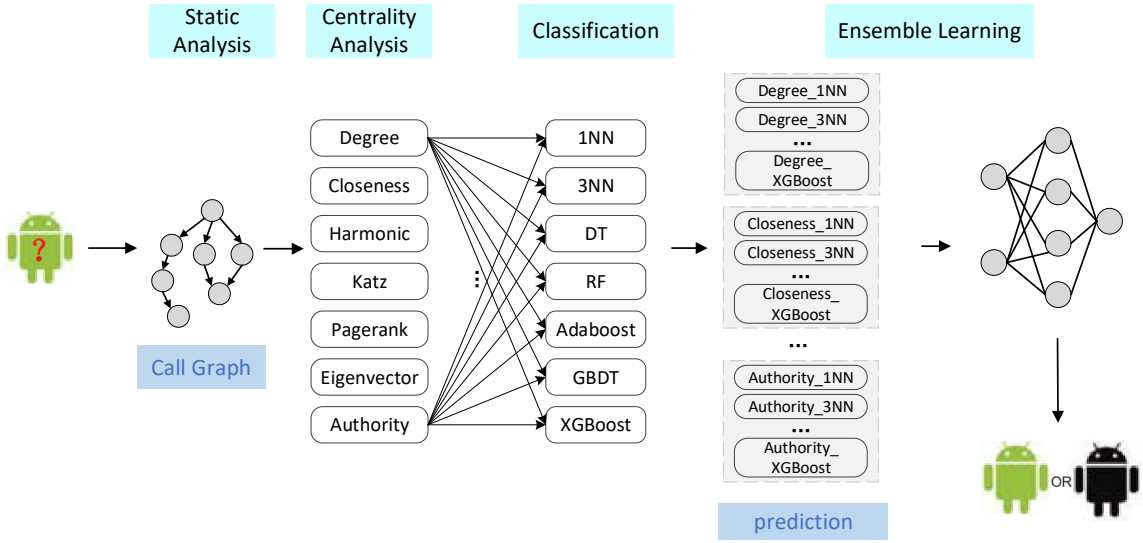


Fig. 2: System architecture of *MalScan*

value of closeness centrality is the sum of the shortest path lengths between the node and other nodes, which is then normalized by dividing it by N-1, where N represents the overall count of nodes.

$$C_C(v) = \frac{N-1}{\sum_y d(t,v)} \tag{2}$$

Note that $d(t,v)$ stands for the shortest path length between nodes v and t.

- *Harmonic centrality* [37] reverses the sum and reciprocal operations in the closeness centrality definition.

$$C_H(v) = \frac{\sum_{t \neq v} \frac{1}{d(t,v)}}{N-1} \tag{3}$$

Note that $d(t,v)$ is the distance between nodes v and t and N is the number of nodes in the graph.

- *Eigenvector centrality* [38] is a function of neighboring node centrality, meaning that if a node is connected to neighbors with higher centrality, its own centrality will also increase. Let A be the adjacency matrix of a graph and the element a_{ij} = 1 if vertex *i* is linked to vertex *j*, and a_{ij} = 0 otherwise.

$$C_E(i) = \frac{1}{\lambda} \sum_{j=1}^{n} a_{ij} C_E(j) \tag{4}$$

Note that C_E is the eigenvector of the matrix and constant λ is its corresponding eigenvalue.

- *Katz centrality* [29] can be viewed as a variant of eigenvector centrality. It gives each node a small amount of centrality β for free to work around the problem of null eigenvector centrality score. Hence, each node has a minimum, positive amount of centrality that it can transfer to other nodes by referring to them.

$$C_K(i) = \alpha \sum_{j=1}^{n} a_{ij} (C_K(j) + \beta) \tag{5}$$

Note that α is an attenuation factor in $(0,1)$ and β is a constant. If $\alpha = \frac{1}{\lambda_{max}}$ and $\beta = 0$, then Katz centrality is equivalent to eigenvector centrality, where λ_{max} is the largest eigenvalue of the adjacency matrix A.

- *Pagerank centrality* [31] further considers the out-degree of the neighbor nodes of the current computational node on the basis of Katz centrality. The contribution of neighbor nodes with high out-degree to the current computation node should be penalized to some extent. It assigns numerical weights to nodes based on their out-degree with the purpose of measuring their relative importance.

$$C_P(i) = d \sum_{j=1}^{n} a_{ji} \frac{C_P(j)}{L(j)} + \frac{1-d}{N} \tag{6}$$

Note that $L(j)$ signifies the count of neighbors that node j has, while d represents the damping factor.

- *Kleinberg centrality* [39] proposes the *hyperlink-induced topic search* (HITS) algorithm to quantify the authority and hubs attributes of nodes. Nodes with high authorities scores contain contributed original information and are followed by many other hub nodes; nodes with high hub scores summarize a lot of information and point to many authorities nodes. Let x_i and y_i represent the authority centrality and hub centrality of node i, respectively.

$$x_i = \alpha \sum_k a_{k,i}\, y_k \tag{7}$$

$$y_i = \beta \sum_k a_{i,k}\, x_k \tag{8}$$

Note that α and β are constants. considering the function call graph as a social network, sensitive API function nodes are important because they contain valuable content, rather than linking to other important vertices, which are more in line with the characteristics of authorities nodes and perform poorly in hub centrality feature extraction. Therefore, we choose authority centrality to measure the importance of the sensitive API in the function call graph.

3.3 Classification

The purpose of this session is to label whether the app is malicious or benign with single machine learning algorithms and centrality algorithms. We select seven different machine learning algorithms which are the most commonly used

algorithms for classification algorithms: *1-nearest neighbor* (1-NN), *3-nearest neighbor* (3-NN), *decision tree* (DT), *random forest* (RF), *AdaBoost*, *GBDT*, and *XGBoost* to complete classification. Each machine learning model is trained separately using seven different feature vectors obtained in various centrality analysis stages, resulting in 7*7=49 classifier models. Thus, when performing classification on the input dataset, each app in the dataset will get 49 predictions from distinct classifiers. All the experimental results are presented in Section 4 by performing 10-fold cross-validation on our datasets.

3.4 Ensemble Learning

In the final phase, to further improve the accuracy of malware detection, we employ ensemble learning to synthesize multiple models trained in the classification phase. During the classification phase, we employ machine learning algorithms with the aim of obtaining a stable and well-performing classifier in all aspects. However, due to the differences in the emphasis on centrality analysis and the principles of classifier algorithms, only multiple weakly supervised models with preferences are obtained. Hence, it can be a challenge to achieve an impartial and robust supervised model by relying only on a single centrality analysis algorithm and a single machine learning algorithm. Ensemble learning combines multiple weakly supervised models in order to obtain a superior and more comprehensive strongly supervised model. In this manner, the other weak classifiers can correct the error back even if a weak classifier makes an inaccurate prediction.

Following the classification phase, 49 prediction results are obtained for each APK file. Each classifier has a specific focus and gets varying results. Consequently, applying the ensemble learning concept to aggregate the predictions of individual classifiers to obtain a more comprehensive and strongly supervised model is an effective means to improve accuracy. To obtain the final malware detection results, we employ *deep neural network* (DNN) on the results generated in the classification phase to perform binary prediction. We evaluated DNN architectures with varying numbers of layers and determined that a three-layer DNN was sufficient to meet our experimental requirements. Therefore, we select the three-layer DNN as the final architecture for our ensemble learning network. We select the cross-entropy function as the loss function and choose the Adam optimizer with a learning rate of 0.0001. We set the hyperparameters Batch Size to 32 and epoch to 50. In the classification phase, a ten-fold cross-validation method is employed to maximize the utility of the dataset and to minimize model evaluation errors arising from uneven data distribution and randomness. Specifically, the dataset is divided into ten subsets, where nine subsets are used as training data and one subset as testing data in a rotating manner. In the ensemble phase, the predicted outcomes from the ten rounds of test data for each classifier model are combined to form the final prediction result for that classifier. Subsequently, the prediction results from the 49 classifier models in the classification phase are integrated to create a dataset for training the ensemble learning model DNN.

4 Experimental Evaluation

In this section, we mainly answer the following questions:

TABLE 1: Details of the experimental dataset

Dataset	Benign	Malware	Total	Average Size (MB)
2017	21,000	21,000	42,000	2.49
2018	21,000	20,955	41,955	3.73
2019	20,985	20,999	41,984	6.56
2020	20,993	23,324	44,317	7.15
2021	20,914	22,362	43,276	8.35
Total	**104,892**	**108,640**	**213,532**	**6.17**

- *RQ1: What is the malware detection performance of MalScan with different methods?*
- *RQ2: How effective is MalScan trained with the old dataset in detecting new samples?*
- *RQ3: What is the performance of MalScan in classifying obfuscated Android malware?*
- *RQ4: What is MalScan's runtime overhead while identifying Android malware?*
- *RQ5: Can MalScan achieve large-scale Android malware detection?*

4.1 Experimental Settings

4.1.1 Dataset

We extend the dataset to improve the accuracy of malware detection and the detail of the dataset is available in github[2], from which researchers are able to conduct repeatable experiments. *AndroZoo* [18] platform hosts an extensive collection of over nine million APK files, each of which has undergone comprehensive scrutiny by various antivirus software on *VirusTotal* [40]. We crawl a subset of 104,892 benign applications and 108,640 malicious applications from *AndroZoo* to form our dataset. It's important to note that Android malware is in a constant state of flux, with attackers continuously creating new variants and techniques to evade existing security measures. Therefore, in order to comprehensively assess the robustness and reliability of Malscan, we have chosen datasets from different time periods (2017-2021) to validate the performance of the Android malware detection system, *Malscan*. By utilizing datasets that span multiple time intervals, we can gain a better understanding of *Malscan*'s performance across different periods and evaluate its resilience in addressing various forms of malicious code evolution. Table 1 presents the specific details of the datasets.

4.1.2 Implementation

We run all experiments on a server with 32 cores of CPU. In the static analysis process, we utilize the tool *Androguard* [36] to parse Android apps and generate function call graphs. In the centrality analysis process, we treat the function call graph as a social network and leverage the python library *networkx* [41] to extract distinct centrality vectors. In the classification phase, the machine learning models (*e.g.*, 1-NN, 3-NN, DT, RF, Adaboost, XGBoost, GBDT) are implemented with the python library *scikit-learn* [42]. In the ensemble learning phase, we construct and train a DNN model based on *PyTorch* [43].

4.1.3 Comparision

All experiments are compared with the following advanced Android malware detection methods:

- *Drebin* [8]: *Drebin* utilizes comprehensive static analysis to extract as many diverse features as possible from applications, and then embeds these features into a unified vector

2. https://github.com/MalScanCodes/MalScan.

space, thereby achieving the categorization of malicious software.

- *MaMaDroid* [15]: *MaMaDroid* employs Markov chains to construct behavioral models from the extracted API call sequences within the function invocation graph. It utilizes the state transition probabilities within the model as feature vectors for classification purposes.
- *HomDroid* [19]: *HomDroid* employs community detection to partition an application's function call graph into subgraphs, utilizes homophily analysis to identify the most suspicious subgraph, and subsequently extracts features to apply machine learning classifiers for malware detection.
- *Xmal* [20]: *Xmal* extracts API calls and permissions from APK files and inputs them as features into a *multi-layer perceptron* (MLP) to predict whether an application is malicious. The MLP classifier integrates an attention layer to learn and weigh the features' significance.
- *RAMDA* [21]: *RAMDA* employs a variational autoencoder (FD-VAE) to extract static features such as permission requests, intent action declarations, and sensitive API calls from APK files, generating corresponding binary vector representations. These representations are subsequently fed into a multilayer perceptron (MLP) for Android malware detection.
- *MSDroid* [22]: *MSDroid* decomposes the function call graph into subgraphs rooted in sensitive API calls and employs graph encoding techniques to represent the code attributes and domain knowledge of these subgraphs. Subsequently, graph neural networks (GNNs) are utilized to process these subgraphs, enabling g malware detection.

4.1.4 Metrics

We introduce the metrics *TP* (True Positive), *TN* (True Negative), *FP* (False Positive), and *FN* (False Negative) to measure the results of classifier recognition. The first letter of the two metrics indicates whether the classifier's recognition result is correct, with correct being denoted by the initial *T* of True and incorrect by the initial *F* of False. The second letter indicates the result of the classifier's determination, with *P* indicating that the classifier determined a malicious sample and *N* indicating that the classifier determined a benign sample. Therefore, *TP*, *TN*, *FP*, and *FN* respectively represent samples correctly classified as malicious, samples correctly classified as benign, samples incorrectly classified as malicious, and samples incorrectly classified as benign. Based on the number of samples counted by *TP*, *TN*, *FP*, and *FN*, metrics are calculated to evaluate the model. The definitions of the metrics and the corresponding formulas are as follows: *Precision* (P), *Recall* (R), and *F-measure* (F1). $P = \frac{TP}{TP+FP}$, $R = \frac{TP}{TP+FN}$, $F1 = \frac{2*P*R}{P+R}$.

4.2 RQ1:Detection Effectiveness

To demonstrate the performance of different centrality measures and machine learning algorithms in *MalScan*, we conducted experiments on datasets spanning five years (2017-2021) as shown in Table 1. Table 2 presents the F-measure results achieved by *MalScan* on the datasets from each year, using various centrality analysis methods and machine learning algorithms. Furthermore, in order to provide a clearer depiction of *MalScan*'s overall performance on each dataset, the table also presents the average f-measure values when *MalScan* classifies datasets from the years 2017 to 2021. To validate the efficacy of various centrality measures in the identification of Android malware, we carry out seven distinct centrality experiments. In particular, for each centrality-derived feature, we employ seven different machine learning algorithms for classification. In other words, according to different features and classification algorithms, we evaluate *MalScan* by conducting 7*7=49 experiments on each dataset per year.

Table 2 shows that the experimental effects vary across datasets and *MalScan* consistently maintain a high f-measure above 98% for almost experiments. Furthermore, the detection performance exhibits variation based on the chosen centrality measures and machine learning algorithms. For instance, when 1NN is chosen as the classifier to classify the 2021 dataset, the f-measure is 92.08% when we select eigenvector centrality while is 98.70% when we select degree centrality. The primary reason for this occurrence is the differing definitions of the selected centrality metrics. In contrast, when opting for degree centrality to derive feature vectors, the f-measure reaches 98.13% with Decision Tree as the chosen classifier for the 2020 dataset, while it achieves 99.29% with the selection of XGBoost. Table 2 indicates that detection is optimal when the pagerank centrality analysis algorithm and XGBoost machine learning algorithm are selected, ranking first in 2018-2021 datasets (F1 values of 98.2%, 99.3%, 99.2%, and 99.0%, respectively) and second (F1 value of 98.2%) in 2017 dataset.

MalScan introduces improvements over the original design methodology. To more clearly demonstrate its effectiveness in malware detection, we conduct comparative experiments between *MalScan*, the original method (referred to as *MalScan_origin* in this paper), and six selected benchmark tools, using datasets spanning five years (2017-2021). Table 3 presents the F1 scores from these comparative experiments. The ensemble of different centrality and different algorithms are applied in the ensemble learning phase. We integrate the prediction results generated by 49 individual classifiers in the classification stage and combine these prediction results with the corresponding labels as datasets feeding into DNN for binary classification training. Table 2 and Table 3 indicate that *MalScan* achieves F1 scores of 98.58%, 98.48%, 99.44%, 99.33%, and 99.27% in the 2017 to 2021 datasets, respectively. These scores outperform the best-performing models in the classification stage in each dataset, whose F1 scores are 98.3%, 98.2%, 99.3%, 99.2%, and 99.0%. The improvement of ensemble learning is not significant because the original baseline effects are excellent. In summary, compared to experiments based on a single centrality and a single algorithm, ensemble learning has the best experimental performance on all datasets. It is more stable among each dataset due to the more comprehensive features of ensemble learning. As a result, we employ the ensemble learning method for subsequent overall effectiveness experiments.

The experimental results presented in Table 3 indicate that *MalScan* outperforms other comparative methods in malware detection within the 2017 to 2021 datasets. This superiority can be attributed to our ability to capture the significant features of nodes in the graph structure through a multidimensional centrality analysis of sensitive APIs

TABLE 2: The F1 of *MalScan* using different methods and datasets

Dataset	**2017**							**2018**							**2019**						
Metrics	1NN	3NN	RF	DT	ADA	GBDT	XGB	1NN	3NN	RF	DT	ADA	GBDT	XGB	1NN	3NN	RF	DT	ADA	GBDT	XGB
Degree	97.4	97.2	98.1	97.1	97.8	97.4	98.2	97.4	97.4	98.0	97.4	97.9	97.8	98.2	99.0	98.9	99.2	98.9	99.2	99.1	99.3
Closeness	97.2	97.2	98.1	97.1	97.8	97.4	98.2	97.4	97.2	98.1	97.4	98.0	97.7	98.2	98.6	98.9	99.2	98.9	99.2	99.0	99.3
Harmonic	97.1	97.2	98.2	97.3	97.9	97.5	**98.3**	97.4	97.0	97.9	97.3	97.8	97.6	98.0	98.0	98.0	98.3	98.1	98.3	98.2	98.4
Katz	97.7	97.7	98.1	97.3	97.9	97.6	98.2	97.9	97.7	98.0	97.3	97.9	97.5	98.1	99.2	99.0	99.1	98.7	99.1	98.8	99.2
Eigenvector	90.4	90.6	87.9	87.7	87.8	87.8	92.9	87.9	90.6	88.6	88.2	88.3	88.3	93.8	89.0	93.4	86.5	86.4	86.4	86.4	94.1
Pagerank	97.5	97.4	98.1	97.0	97.8	97.3	98.2	97.5	97.5	97.8	97.2	97.6	97.4	**98.2**	98.7	98.9	98.4	98.1	98.4	98.1	**99.3**
Authority	97.3	97.2	98.0	97.2	97.8	97.4	98.2	97.2	97.1	97.8	97.0	97.6	97.3	98.2	98.9	98.7	98.4	98.1	98.4	98.3	99.3
Dataset	**2020**							**2021**							**Mean**						
Metrics	1NN	3NN	RF	DT	ADA	GBDT	XGB	1NN	3NN	RF	DT	ADA	GBDT	XGB	1NN	3NN	RF	DT	ADA	GBDT	XGB
Degree	98.8	98.7	99.1	98.7	99.1	98.9	99.2	98.7	98.2	99.0	98.6	98.9	98.7	99.0	98.3	98.1	98.7	98.2	98.6	98.4	98.8
Closeness	98.9	98.8	99.1	98.8	99.1	98.9	99.2	98.3	97.7	99.0	98.5	98.9	98.6	99.0	98.1	97.9	98.7	98.1	98.6	98.3	98.8
Harmonic	98.6	98.6	98.9	98.6	98.9	98.8	99.0	97.9	97.3	98.4	98.0	98.4	98.1	98.4	97.8	97.6	98.4	97.9	98.3	98.0	98.4
Katz	98.8	98.8	99.1	98.8	99.1	98.9	99.1	98.6	98.0	99.0	98.6	99.0	98.7	99.0	98.4	98.2	98.6	98.1	98.6	98.3	98.7
Eigenvector	94.1	94.3	90.6	90.4	90.5	90.5	94.9	92.1	92.5	91.7	91.6	91.7	91.7	93.4	90.7	92.3	89.0	88.9	88.9	88.9	93.8
Pagerank	98.7	98.7	98.9	98.6	98.8	98.7	**99.2**	98.6	97.9	99.0	98.6	98.9	98.6	**99.0**	98.2	98.1	98.4	97.9	98.3	98.0	**98.8**
Authority	98.7	98.6	98.9	98.6	98.8	98.8	99.1	98.1	97.4	98.7	98.3	98.6	98.4	98.9	98.0	97.8	98.3	97.8	98.2	98.0	98.7

TABLE 3: The F1 Score of *MalScan*, *Drebin*, *MaMaDroid*, *MalScan_origin*, *HomDroid*, *Xmal*, *RAMDA* and *MSDroid* classification with datasets from the same year

Dataset	**2017**	**2018**	**2019**	**2020**	**2021**
MalScan	**98.58**	**98.48**	**99.44**	**99.33**	**99.27**
Drebin	94.25	95.32	95.41	98.52	97.14
MaMaDroid	97.33	97.17	96.63	98.61	97.82
MalScan_origin	97.39	97.41	98.98	98.83	98.70
HomDroid	98.02	97.38	97.63	98.73	98.59
Xmal	97.91	97.29	97.15	98.87	98.26
RAMDA	94.06	94.47	94.88	96.72	96.86
MSDroid	93.85	94.31	95.02	97.11	96.43

in the function call graph. Such a comprehensive feature representation enhances the model's capability to recognize critical behavioral patterns. Furthermore, the integration strategy that combines multiple machine learning classifiers further improves the robustness and generalization of the classification process. This multi-level, multi-algorithm integration framework effectively enhances detection performance against complex malware patterns, highlighting the advantages of this approach in feature extraction and classification processes.

Compared to the original method *MalScan_origin*, *MalScan* demonstrates significant advantages in the richness of feature representation, the capture of global structure, and the robustness of classification by employing multi-dimensional centrality extraction for sensitive APIs and integrating multiple classifiers. This enhanced approach facilitates more efficient detection of malware. *Drebin*, *Xmal*, and *RAMDA* extract diverse features from programs, including permissions, intents, and API calls. However, the homogeneity of their feature vectors is insufficient to capture the structural semantics of complex malware samples. Additionally, *MaMaDroid* may produce potential false positives during the abstraction of API calls. For instance, API calls such as *TelephonyManager.getDeviceId()* and *SmsManager.sendTextMessage()* can be abstracted under the same package and family, namely *android.telephony* and *android*, respectively. Both *HomDroid* and *MalScan* focus on the invocation of sensitive APIs and analyze features through graphical structures. However, *HomDroid* employs a single community detection algorithm and machine learning classifier, which limits its capacity for comprehensive evaluation of potential malicious behaviors within programs. Although *MSDroid* also utilizes graph-based malware detection, it focuses on local code fragments rooted in sensitive API calls, which may overlook critical information within the overall program logic. Additionally, the performance of *graph neural networks* (GNNs) in processing graph data is highly dependent on the quality and diversity of the input features, which may result in a decline in classification performance if feature selection is suboptimal.

Summary: *For a single classifier, MalScan's effectiveness performs best when the pagerank centrality analysis algorithm and XGBoost machine learning algorithm are selected. In addition, MalScan's ensemble learning phase achieves excellent F1 scores and outperforms the best-performing models in the classification stage in each dataset. MalScan exhibits superior malware detection capabilities compared to six advanced methods when trained and tested on samples originating from the same year.*

4.3 RQ2:Robustness against Android Evolution

To gauge the adaptability of *MalScan* to the evolving dynamics of Android malware, we devise and execute a series of experiments encompassing four distinct scenarios. Each scenario involves training the system model on datasets from different years and subsequently classifying samples from the remaining years. Specifically, these four scenarios employ datasets from the years 2017, 2017-2018, 2017-2019, and 2017-2020 for training. Subsequently, the trained models are employed to classify samples from the years 2018-2021, 2019-2021, 2020-2021, and solely the year 2021, respectively. In particular, to maintain consistency across these four scenarios, the training sample sizes for the subsequent three scenarios matched that of the 2017 dataset, comprising 20,000 benign applications and 20,000 malicious applications.

To investigate the overarching efficacy of employing an outdated dataset for training in the context of detecting more recent samples, Table 4 presents the average F1 scores obtained during the classification phase of *MalScan*, focusing on the detection of more recent datasets spanning from one to four years. According to the table, the detection performances vary according to the chosen centrality measures, machine learning algorithm and time interval of the datasets. In general, *MalScan* performs relatively well in cross-year experiments, achieving F1 values above 80% for most experiments. However, there is always a poor

TABLE 4: F1 Values of *MalScan* to perform detection on more recent samples by training an outdated dataset

Dataset	one year							two year						
Metrics	1NN	3NN	RF	DT	ADA	GBDT	XGB	1NN	3NN	RF	DT	ADA	GBDT	XGB
Degree	93.41	93.64	94.92	92.78	94.93	93.24	94.98	88.30	88.93	90.24	70.71	90.56	70.85	89.93
Closeness	92.85	93.30	94.81	91.98	94.59	92.37	**95.19**	87.97	88.33	90.64	85.91	89.75	69.62	90.32
Harmonic	92.70	92.91	94.53	88.01	92.16	91.31	94.65	88.09	87.76	90.69	57.84	83.73	84.48	90.35
Katz	93.87	94.05	94.79	92.48	93.74	93.29	94.82	89.13	88.34	90.21	86.58	89.13	88.80	90.61
Eigenvector	86.83	88.21	81.50	80.83	81.04	81.02	90.31	81.48	81.79	69.98	66.22	67.68	67.06	76.25
Pagerank	93.15	93.36	94.50	91.90	93.61	92.30	95.12	88.86	88.88	90.23	68.06	89.19	69.52	90.04
Authority	93.27	93.50	94.51	80.14	93.76	93.24	94.97	87.71	86.24	89.95	73.66	**92.76**	88.77	90.76
Dataset	three year							four year						
Metrics	1NN	3NN	RF	DT	ADA	GBDT	XGB	1NN	3NN	RF	DT	ADA	GBDT	XGB
Degree	87.44	87.76	90.06	49.24	90.78	49.47	88.37	87.06	86.74	86.45	9.86	92.40	10.78	78.94
Closeness	77.80	76.32	90.04	85.61	89.30	48.30	90.20	82.00	78.75	86.71	88.05	82.93	29.06	90.56
Harmonic	85.83	86.25	91.31	17.56	60.12	48.02	**92.76**	79.24	78.90	**93.62**	11.87	86.87	10.67	88.31
Katz	87.45	84.52	89.73	87.65	88.93	88.37	90.23	85.69	83.78	85.57	88.27	91.51	90.62	85.76
Eigenvector	80.67	80.90	56.87	47.78	50.57	49.43	62.89	75.85	75.78	65.15	58.24	59.81	56.95	65.64
Pagerank	86.15	84.89	89.63	47.53	88.87	48.53	89.04	80.37	77.79	85.46	8.69	85.08	8.79	84.85
Authority	86.36	84.95	89.71	90.66	90.00	91.02	89.84	86.25	85.71	86.03	83.26	86.93	83.05	89.93

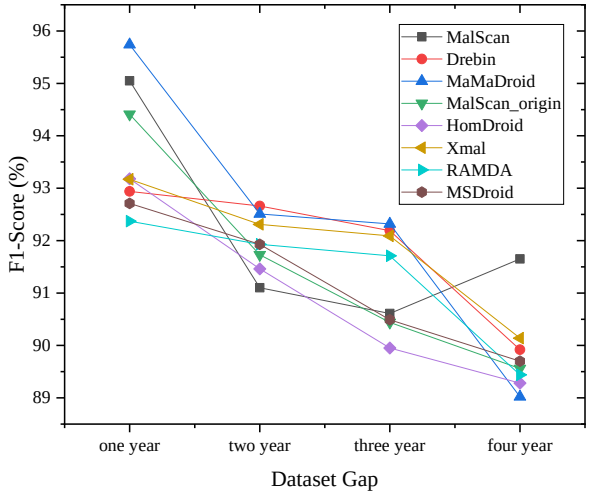


Fig. 3: Average F1 scores of *MalScan*, *Drebin*, *MaMaDroid*, *MalScan_origin*, *HomDroid*, *Xmal*, *RAMDA* and *MSDroid* for classifying recent samples using old training data

performance of partial centrality and the results obtained are unstable. For instance, when Decision Tree is chosen as the classification algorithm and harmonic centrality is used to extract features, the mean value of f-measure for the datasets with a time interval of three years is 17.56%. This is primarily attributed to the enhanced adaptability of abstracted API calls to the evolving dynamics of Android applications. When centrality and the machine learning algorithm are fixed, the F-values of most experiments show a slight decrease as the time span increases. For example, when katz centrality is chosen to extract features and the random forest is used for classification, the F-values of classification are 94.79%, 90.21%, 89.73%, and 85.57% respectively with increasing time span.

Table 4 reveals the best-performing classifiers vary across datasets with different time spans, and it is difficult to find a stable single classifier model to resist Android app evolution. The phase of ensemble learning abates the instability of *MalScan* to a certain extent and achieves better F1 values across data sets with different time spans (95.05%, 90.10%, 90.61%, 91.65%, respectively). To more significantly observe the change in F-values with time span, Figure 3 uses a line graph to show the mean values for different time spans.

To investigate the overall performance of detecting new

TABLE 5: Descriptions of 12 obfuscators used in our experiments

Obfuscator	Descriptions
ClassRename	Change the package name and rename classes
FieldRename	Rename fields
MethodRename	Rename methods
AssetEncryption	Encrypt asset files
ConstStringEncryption	Encrypt constant strings in code
LibEncryption	Encrypt native libs
ResStringEncryption	Encrypt strings in resources
ArithmeticBranch	Insert junk code that is composed by arithmetic computations and a branch instruction
CallIndirection	Modify the control-flow graph without changing the code semantics
Goto	Modify the control-flow graph by adding two new nodes
Nop	Insert random nop instructions within every method implementation
Reorder	Change the order of basic blocks of the control-flow graph

samples when training with outdated datasets, we present the average f-measure of *MalScan* alongside selected comparison tools when detecting new datasets spanning from one to four years. The experimental results are shown in Figure 3, the dataset gap labels indicate the time span between the training and testing sets. For instance, "one year" signifies that when selecting the years 2017, 2018, 2019, and 2020 as the training sets, the corresponding test data will be from 2018, 2019, 2020, and 2021, respectively. To ensure the broad applicability and reliability of the results, the outcome corresponding to the "one year" label is represented by the average F-measure value. Similarly, the labels "two years", "three years", and "four years" follow the same logic. In general, as the time span increases, the performance of all tools demonstrates a declining trend. This indicates that, with the evolution of *MalScan*, detectors trained on outdated datasets gradually become less effective in handling newly emerging samples, especially when the time gap is larger. This performance degradation reflects the impact of the evolving behavior and characteristics of malware on detection tools.

As shown in Figure 3, when the time gap between the test dataset and the training dataset is one year, *MalScan* maintains relatively good performance. However, as the time gap increases to two years, the f-measure of both *MalScan* and *HomDroid* drops significantly. This is mainly

TABLE 6: F1 values of MalScan using different methods and datasets on classifying obfuscated apps

Obfuscator		Degree	Closeness	Harmonic	Katz	Eigenvector	Pagerank	Authority	MalScan
Rename	ClassRename	98.75	98.75	98.63	98.81	93.82	98.87	98.81	**98.99**
	FieldRename	98.65	98.77	98.59	98.77	93.67	**98.88**	97.31	98.87
	MethodRename	98.26	98.2	91.41	97.82	88.16	**98.56**	98.31	97.80
Encryption	AssetEncryption	98.64	98.64	98.46	98.76	93.57	98.76	98.65	**99.00**
	ConstStringEncryption	**98.14**	96.89	95.16	95.77	92.05	95.94	96.42	96.60
	LibEncryption	98.64	98.82	98.58	98.82	93.4	98.87	98.88	**99.06**
	ResStringEncryption	98.57	98.63	98.57	98.75	92.93	**98.93**	98.75	98.78
Code	ArithmeticBranch	98.65	98.77	98.59	98.77	93.49	**98.88**	97.14	98.87
	CallIndirection	89.57	96.13	**96.31**	96.23	87.77	94.99	7.45	95.39
	Goto	98.65	98.77	98.59	98.77	93.16	98.88	97.26	**98.98**
	Nop	98.98	98.92	98.74	99.04	94.70	98.98	97.14	**99.08**
	Reorder	98.81	98.81	98.69	98.93	93.98	98.99	97.05	**99.01**
F1s of all generated samples		97.91	98.34	97.56	98.28	92.58	98.29	93.65	**98.38**

due to the considerable changes in sensitive API calls between datasets over this time span, while *MaMaDroid* rely on abstract processing of sensitive API calls, which makes them more resilient to the evolution of Android applications. Nevertheless, when the time gap further increases to three and four years, *MalScan*'s f-measure decline becomes more moderate, and it even shows an upward trend in the fourth year. This may be because malware undergoes substantial changes in its initial evolution stages, causing a large number of sensitive APIs to be obfuscated, renamed, or discarded, which adversely affects detection performance. However, as the evolution stabilizes, changes in sensitive APIs diminish, enabling *MalScan* to better adapt to these changes, resulting in a slower performance decline and even an improvement in later stages.

Summary: *The performance of a single classifier model fluctuates across datasets with different time spans, yielding unstable results. Ensemble learning improves the stability of Malscan against app evolution to some extent and achieves better F1. In the experiments addressing temporal evolution, all tools exhibit a decline in their f-measure as the time span increases. Although MalScan does not demonstrate a significant advantage compared to other methods, it nonetheless maintains a relatively stable detection performance, indicating a certain degree of resilience to temporal evolution.*

4.4 RQ3:Robustness against Code Obfuscation

To investigate the resistance of *MalScan* to obfuscation, we introduce a modular Python tool *Obfuscapk* [44] to obfuscate the samples and test the obfuscated samples with the trained model. *Obfuscapk* provides several types of obfuscators, including not only some typical obfuscation (*e.g.,* renaming, encryption) but also some advanced code obfuscation (*e.g.,* Goto). Based on representativeness and experimental feasibility, and referencing papers [45] and [46], we select a subset of representative operators from the total of 20/21 operators across different categories for our experiments, ensuring the validity and relevance of the results. Specifically, we select 12 different obfuscators from *Obfuscapk*, including three renaming obfuscators, four encryption obfuscators, and five advanced code obfuscators. The descriptions are given in Table 5.

Due to the high computational cost of generating obfuscated samples, following the methodology outlined in references [45] and [46], we carefully select 1,000 benign applications and 1,000 malicious applications from the 2021 dataset as test samples, while the remaining samples are used to train the classification model. Subsequently, the 12 obfuscators mentioned in Table 5 are used to obfuscate the test samples separately. Specifically, each APK file in the dataset are sequentially processed using 12 different obfuscation tools, with each tool generating an APK file transformed by a distinct obfuscation strategy. In other words, each original APK file is converted into 12 files, each obfuscated using a different method. In practice, some obfuscated samples are not able to generate due to certain errors, and we finally obtain 10,060 obfuscated benign samples and 10,130 obfuscated malicious samples. The trained models are utilized to classify the obfuscated samples to assess the robustness of *MalScan* to obfuscation. Due to the excellent performance of the XGBoost machine learning algorithm and ensemble learning in effectiveness experiments, the results of the XGBoost algorithm for classifying feature vectors generated by different centrality analyses and the results of ensemble learning are chosen for presentation. The experimental results for *Drebin* and *MaMaDroid* are exhibited for comparison.

Table 6 presents *MalScan*'s F-measures for classifying obfuscated applications. Since the typical rename and encryption obfuscations do not change the call relationships between functions in an app, *MalScan* anti-obfuscation works well for most of the typical obfuscators that can correctly classify most obfuscated apps into the corresponding label. However, when the selected obfuscator is CallIndirection, *MalScan*'s detection was much less effective than the other obfuscator experiments. Especially when authority was selected as the centrality analysis, the F-value of XGBoost classification results was only 7.45%. After our in-depth analysis, we find that the number of nodes and edges in a function call graph changes a lot after applying CallIndirection. For instance, after CallIndirection's obfuscation process, a sample that initially has 8,135 nodes and 19,725 edges is transformed to 35,396 nodes and the number of edges increases to 58,407. *MalScan* misclassified the sample due to the significant change. As shown in Table 6, the single classifier performs best when classifying with pagerank centrality analysis and XGBoost, while the ensemble learning algorithm can further improve the stability and effectiveness of anti-confusion on top of that. For example, pangrank centrality analysis (94.99%) does not perform as well as Harmonic centrality analysis (96.31%) when countering CallIndirection confusion, and ensemble learning achieve superior results (95.39%) by fusing multiple classifier algo-

TABLE 7: F1 values of *MalScan*, *Drebin*, *MaMaDroid*, *MalScan_origin*, *HomDroid*, *Xmal*, *RAMDA* and *MSDroid* on classifying obfuscated apps

Obfuscator		MalScan	Drebin	Mamadroid	Malscan_origin	HomDroid	Xmal	RAMDA	MSDroid
Rename	ClassRename	98.99	92.24	99.73	98.75	98.13	93.37	94.03	97.02
	FieldRename	98.87	93.52	97.56	98.65	98.18	94.42	94.58	95.33
	MethodRename	97.80	90.24	97.39	98.26	97.05	91.18	91.87	96.19
Encryption	AssetEncryption	99.00	92.64	97.85	98.64	94.63	93.52	93.12	94.75
	ConstStringEncryption	96.60	91.88	98.72	98.14	93.92	92.77	93.05	93.48
	LibEncryption	99.06	94.07	98.47	98.64	95.77	94.81	94.34	93.64
	ResStringEncryption	98.78	92.83	97.34	98.57	94.89	93.82	93.27	92.27
Code	ArithmeticBranch	98.87	92.65	98.36	98.65	94.01	93.47	93.78	95.41
	CallIndirection	95.39	89.59	96.28	89.57	92.37	90.43	91.3	93.24
	Goto	98.98	92.37	98.21	98.65	94.76	93.52	93.56	95.29
	Nop	99.08	93.99	98.60	98.98	94.9	94.86	94.51	95.13
	Reorder	99.01	94.01	98.89	98.81	95.33	94.99	94.87	95.62
F1s of all generated samples		98.38	92.84	98.27	97.91	96.55	93.78	94.13	95.19

rithms.

At the same time, we evaluated the effectiveness of the selected methods, previously trained on the original training set, on the obfuscation test set, with the corresponding results summarized in Table 7. It is evident that the performance of most methods declines after obfuscation is applied. Specifically, the impact of obfuscation on the effectiveness of the detectors largely depends on how the detectors leverage the features of the APK. String-based detection methods (*e.g., Drebin*, *Xmal*, *RAMDA*) primarily rely on static features extracted from applications, such as API calls and permission requests. These methods focus on the syntactic aspects of the program, making them susceptible to simple syntactic obfuscations. For instance, renaming obfuscation (such as ClassRename and FieldRename) obscures the code by changing the names of classes, fields, or methods, directly affecting the feature extraction process of string-based methods and resulting in a decline in their performance. Although methods like *Drebin* can maintain relatively high detection accuracy under certain types of obfuscation, their detection performance significantly deteriorates when faced with method-level renaming (such as MethodRename) or string encryption (such as ConstStringEncryption), due to their strong dependence on string features.

In contrast, graph-based detection methods (*e.g., MalScan*, *Malscan_origin*, *MaMaDroid*, *HomDroid*, and *MSDroid*) not only rely on the syntactic information of the program but also capture the semantic relationships through the function call graph. This enables them to exhibit enhanced robustness when confronted with complex code obfuscations such as code reordering and control flow alterations. Graph-based methods are capable of effectively handling changes in the code structure, as obfuscations like Goto, Nop, and Reorder do not significantly alter the semantic information of the program, leading to relatively stable performance against such obfuscations. For example, *MalScan* achieves a performance score of 99.01% under Reorder obfuscation, while other graph-based methods consistently score above 95%. However, even graph-based methods experience some performance degradation when faced with complex control flow modifications, such as CallIndirection obfuscation. CallIndirection alters the control flow of function calls without directly affecting the semantic information of the program, posing challenges for analysis methods that rely on the function call graph. While methods like *MalScan* still outperform most string-based methods under such obfuscation, their F1 scores notably decline. For instance, *MalScan* scores 95.39% under CallIndirection obfuscation.

Summary: *The single classifier performs best when classifying with pagerank centrality analysis and XGBoost, while the ensemble learning algorithm can further improve the stability and effectiveness of anti-confusion on top of that. In the comparison experiments, string-based detection methods tend to perform poorly against certain types of obfuscation, while graph-based detection methods typically demonstrate greater robustness due to their comprehensive consideration of both syntactic and semantic information of the program.*

4.5 RQ4:Runtime Overhead

Due to the necessity of evaluating all comparison tools, testing all 40,000 applications would require a substantial amount of time. In this section, we randomly selected 5,000 benign applications and 5,000 malicious applications from the 2021 dataset to estimate the runtime overhead of *MalScan*. Since the sizes of each sample in the dataset are not fixed, we calculate the average number of nodes for all benign and malicious samples to reflect the overall characteristics of the dataset. The average number of nodes for benign and malicious examples in this dataset is 40,273 and 29,093, respectively. For the input test samples, *MalScan* primarily conducts its analysis through four stages: Function call graph extraction, Feature extraction, Classification, and Ensemble learning. Table 8 presents the time overhead for the first three stages of *MalScan* for 5,000 benign and 5,000 malicious applications.

The extraction of function call graphs for a given APK file takes an average of 1.51 seconds. Concerning feature extraction, the runtime performance varies based on the selected centrality metrics. The average time required for degree centrality is 0.09 seconds, significantly lower than other centrality measures. With the provided feature vectors, we can employ well-trained machine learning models for classification, with random forests capable of completing the detection of 10,000 applications within 0.01 seconds[3].

Finally, the results generated by the 49 machine learning models are integrated and learned using DNN at the ensemble learning phase, which takes 0.0868 seconds for 10,000

3. During this evaluation, we utilized 10,000 applications to train our machine learning model.

TABLE 8: The detection time overhead of *Malscan*'s First Three Steps for 5,000 benign software and 5,000 malicious software applications

Phases	Graph Construction(h)	Feature Extraction(h)	Classification(s)						
			1NN	3NN	RF	DT	ADA	GBDT	XGB
Degree	4.18	0.26	2.20	2.52	0.15	0.01	0.52	0.11	0.04
Closeness		4.01	2.32	2.57	0.15	0.01	0.49	0.12	0.03
Harmonic		4.16	2.20	2.47	0.16	0.01	0.49	0.12	0.02
Katz		3.61	2.26	2.49	0.15	0.01	0.51	0.11	0.02
Eigenvector		0.37	2.26	2.48	0.16	0.01	0.47	0.10	0.02
Pagerank		0.36	2.26	2.48	0.15	0.01	0.48	0.12	0.02
Authority		0.36	2.25	2.49	0.15	0.01	0.50	0.11	0.02

samples. Overall, after progressing through the four stages mentioned above, *MalScan* is capable of categorizing a new application as benign or malicious within an average time of 6.23 seconds. In subsequent work, we conduct extended experiments specifically focused on the *MalScan* tool, analyzing datasets containing 10,000 and 20,000 applications. The results indicate that the runtime exhibits a linear growth trend. Based on this trend, we hypothesize that *MalScan* maintains good scalability when handling larger datasets. In future work, we plan to further expand the experimental scale to validate the performance effectiveness with even larger datasets.

To further elucidate the scalability of *MalScan*, we conduct a comparative analysis of its runtime overhead against several other malware detection systems, including *Drebin*, *MaMaDroid*, *Malscan_origin*, *HomDroid*, *Xmal*, *RAMDA*, and *MSDroid*, as illustrated in Table 9. The data presented in the table indicate that *MalScan_origin* significantly outperforms other malware detection systems in terms of scalability and efficiency, requiring an average of only 0.68 seconds to detect a single malware instance. As an enhanced version of the original method, *MalScan* exhibits an average detection time of 6.23 seconds per application, which is notably higher than the 0.68 seconds required by the single-classifier *MalScan_origin*. However, *MalScan* significantly improves the effectiveness and robustness of malware detection through its multi-classifier ensemble strategy and more complex feature extraction processes. Given the performance enhancements offered by these methodologies, the additional time overhead is both reasonable and acceptable in practical applications.

Although *Xmal* and *RAMDA* demonstrate shorter detection times of 5.17 seconds and 4.92 seconds, respectively, this increased speed comes at the expense of performance. Specifically, *Xmal* relies solely on static features such as permission requirements, intent declarations, and sensitive API calls, while *RAMDA* only extracts API calls and permission information as its features. Both approaches depend on a relatively simplistic feature representation, lacking a deeper analysis of program structural semantics and global behavioral patterns. This limitation results in suboptimal performance in the detection of complex malware, making it difficult to capture the latent malicious behavior patterns within applications. In the case of *MaMaDroid*, the need to construct a more accurate call graph to preserve context and data flow necessitates an extensive program analysis, taking approximately 165.63 seconds to complete a full classification of each application. *Drebin*, on the other hand, requires the extraction of eight distinct feature sets from disassembled code and manifest files, including some complex features such as network addresses, with the total number of extracted features exceeding 90,000, leading to a highly time-consuming detection process. The time consumption for *HomDroid* and *MSDroid* primarily arises from their intricate graph analysis and processing of function call graphs. *HomDroid* utilizes complex community detection algorithms to partition the application's function call graph into multiple subgraphs, followed by homophily analysis on each subgraph. In contrast, *MSDroid* decomposes the function call graph into subgraphs associated with sensitive API calls, employing graph encoding and GNNs for subgraph inference. These processes necessitate handling complex, large-scale graph structures, involving high computational complexity, especially when the number of subgraphs is substantial, leading to a significant increase in computational cost.

Summary: *Given a new application, a single classifier can detect as fast as 1.6s when choosing between degree centrality analysis and random forest for classification. Furthermore, with the incorporation of ensemble learning, MalScan achieves an average classification time of merely 6.23 seconds for distinguishing between benign and malicious samples. Overall, MalScan achieves a good balance between runtime efficiency and detection accuracy, demonstrating superior detection performance and robustness on large-scale datasets compared to other methods, highlighting its advantages in scalability and stability.*

4.6 RQ5:Market-wide Case Study

To authenticate *MalScan*'s capability in detecting real-world zero-day malicious software, we collect one million apps from the Google-Play app market in February 2023 for the *MalScan* case study, where the average size of the apps is 41.92 MB. Since the ensemble learning experiments perform more consistently and with higher stability and effectiveness in previous detection, we use our 2017-2021 dataset to train the model by adopting an ensemble learning algorithm (49 classifiers are integrated based on 7 different centrality measures and 7 different machine learning algorithms). Next, we feed the trained classifiers with apps crawled from the Google-Play app market.

Out of these million apps, *MalScan* report 523 as malicious. To investigate whether these 523 apps are malicious, we subject them to analysis by uploading them to *VirusTotal* [40]. Out of these 523 applications, 396 are reported as malware by at least one anti-virus scanner. For the remaining 127 apps, we used a state-of-the-art Android app analysis system, *SanDroid* [47], for a more in-depth examination. This system merges static and dynamic analysis to provide com-

TABLE 9: The comparative runtime overheads of *Malscan*, *Drebin*, *MaMaDroid*, *Malscan_origin*, *HomDroid*, *Xmal*, *RAMDA* and *MSDroid* on analyzing our dataset

Methods	MalScan	Drebin	MaMaDroid	MalScan_origin	HomDroid	Xmal	RAMDA	MSDroid
Average Runtime(s)	6.23	121.41	229.52	0.68	52.75	5.17	4.92	60.41

prehensive insights into risky behavior. By analyzing the above 127 apps, we discover highly suspicious behavior in 102 of them. Analysis of the above reports of software with highly suspicious behavior shows that most of the software contains a variety of risky behaviors such as containing sensitive files, connecting to the Internet, encrypting or decrypting data, and executing shell code, which is often achieved by calling sensitive APIs. From the report, it is clear that most of the malware suffers from data leakage. This software obtains sensitive information such as the user's geographic location, phone number, and device ID by calling sensitive APIs (*e.g., LocationManager.getLastKnownLocation()*, *TelephonyManager.getDeviceId()*), and some software can also steal the user's privacy by obtaining camera, recording and other dangerous permission, thus gaining illegal benefits and bringing huge economic losses to users and even leading to privacy leakage problems. In addition, some malware can gain high-risk privileges by calling sensitive APIs. For example, the application Omi can execute shellcode by calling the function Runtime.exec(), which can be used to exploit specific vulnerabilities in the system and gain higher privileges to modify the device.

In summary, *MalScan* successfully identifies 498 instances of zero-day malware among a pool of one million Google Play applications and 102 of which are not reported as malware by the existing tool [40]. In order to understand the distribution of malware in the Google Play marketplace, we conduct statistical research on the 498 zero-day malware detected, and it is worth noting that 114 of the 498 zero-day malware detected by *MalScan* have been removed from the Google-Play app market. We gather specific details for the remaining 384 applications, encompassing their app category, size, download count, and user ratings. The statistics for the remaining 384 apps are shown in Table 10.

The majority of these malwares are lifestyle, social and gaming applications that lure users into installing them by pretending to offer some specialised features. However, after installation, they immediately change their names and icons to make them difficult to discover and uninstall. In addition, they serve intrusive ads to users by abusing WebView. As these apps use their own framework to load ads, they may place additional payloads on infected devices. Based on the number of downloads and ratings, a large number of users have already been infected. In this case, Google's responsibility is to promptly enhance security measures to detect and prevent malicious software in time, thereby reducing the threat to users.

Summary: *MalScan successfully identifies 498 zero-day malware from one million Google-Play apps and 102 of which were not reported as malware by the existing tool. In addition, 114 of the 498 zero-day malware detected by MalScan have been removed from the Google-Play app market, which further demonstrates the effectiveness of our detection.*

5 Discussion

5.1 Threats to Validity

Android malware detection lacks a unified dataset, and to ensure the effectiveness of the detection, we further expand the dataset based on our original work by randomly selecting over 200,000 tagged apps from *AndroZoo* as our dataset. In addition, for the given dataset we randomly divide the training set and the test set in a ratio of 9:1. To reduce the impact of training set selection on classification model training, we present all experimental outcomes through a 10-fold cross-validation conducted on our dataset. Furthermore, due to fluctuating machine states, the time overhead computation is susceptible to mistakes. We take the mean value after performing the measurement numerous times to confirm the reliability and validity of the findings in order to reduce the risk.

5.2 Why does Malscan perform better?

Firstly, in terms of scalability, *MalScan* effectively captures the semantic features of the function call graph of an application by treating it as a social network and employing centrality analysis based on social network theory. This approach demonstrates greater time efficiency compared to graph-based methods such as *MaMaDroid*, *HomDroid*, and *MSDroid*, enabling it to more effectively handle large and complex applications. Furthermore, the focus on the centrality analysis of only 426 critical APIs significantly reduces the time consumption associated with API tracking.

Secondly, regarding accuracy, our emphasis on the 426 key sensitive APIs mitigates the risk of model overfitting, thereby enhancing the precision and recall of the model. Building on existing methodologies, we also incorporate three additional centrality analysis methods and four machine learning algorithms for malware identification. Finally, by introducing ensemble learning to integrate the prediction results of multiple classifiers, we construct a more robust and comprehensive supervised model. This strategy effectively enhances detection capabilities, giving MalScan a significant advantage in the field of malware detection.

5.3 Lessons

Through our research, we discover that random forest is the quickest among the selected seven machine learning algorithms, while xgboost performs the best. We also observe that out of the seven centrality measures we selected, degree centrality analysis is the fastest, while pagerank centrality is the most effective. For the single classification model, the models generated by pagerank centrality and xgboost classification algorithms perform best in detection effectiveness and anti-confusion experiments; in terms of anti-evolution, each single classifier performs inconsistently across different datasets; in scalable experiments, classification using degree centrality analysis and the random forest algorithm takes the least time, with an average of 1.6s to complete the detection of an app. *Malscan*'s integrated learning algorithm outperforms the single classifier model in terms of effective-

TABLE 10: Category distribution of zero-day malware detected by *MalScan*

Category	# Apps	# Downloads	Size(MB)			Rating			# Ratings
			Avg.	Max	Min	Avg.	Max	Min	
Travel & Local	14	595,110	46.96	86.56	11.92	4.30	4.30	4.30	5,570
Shopping	19	1,751,560	47.79	92.64	11.64	4.27	5.00	3.30	7,934
Health	15	1,055,100	63.98	106.61	25.27	2.90	2.90	2.90	23,000
Browser	20	1,237,230	28.23	80.95	4.79	4.37	4.50	4.20	47,532
Educational	12	427,000	66.80	172.07	11.02	4.63	4.90	4.50	8,080
Finance	46	20,831,655	52.56	103.84	8.91	3.54	5.00	1.60	231,914
Lifestyle	76	114,434,750	36.12	182.86	7.32	4.12	5.00	2.70	13,171,537
Social	42	120,248,605	61.25	91.46	14.13	4.04	4.50	3.20	3,192,660
Books & Reference	22	37,303,000	36.87	95.29	13.16	4.39	4.80	3.70	1,212,784
Productivity	9	500,516,110	28.41	80.01	6.51	3.63	4.60	1.80	8,921,482
Music & Audio	39	112,673,060	27.68	87.85	11.18	4.40	5.00	3.70	9,198,210
Game	49	64,650,560	44.58	147.65	3.34	4.00	4.70	2.50	6,910,456
News & Magazines	21	1,222,650	40.06	87.41	8.55	4.13	5.00	3.60	10,955
Total	384	976,946,390	44.71	182.86	3.34	4.06	5.00	1.60	42,942,114

ness and robustness, but it is more time-consuming than the single classifier, taking 6.23s on average to detect an app.

According to the experimental results, if the testers want to ensure certain effectiveness and relatively fast detection, we suggest choosing the model trained by pangrank centrality analysis and xgboost machine learning algorithm for detection, which takes 1.63s on average to get the detection results. For more accurate results, the ensemble learning model of *Malscan* is chosen which takes 6.23s on average.

5.4 Limitation and Future Work

In our research, we thoroughly choose seven distinct centrality measures and seven machine-learning techniques to classify Android apps. Although the robustness of *MalScan* against evolution is limited, its efficiency in malware scanning positions it as a primary line of defense. By sieving through the majority of malicious software, it paves the way for the utilization of other computationally intensive and more robust methods as a secondary layer of defense. This strategic approach enables us to conserve valuable time and resources. Furthermore, given that the majority of Android malware detection systems are closed source, we only compared *MalScan* against six state-of-the-art systems. In our future work, we intend to conduct in-depth comparative analyses involving a broader spectrum of systems.

6 Related Work

Depending on the type of information extracted, the large number of malware detection methods currently proposed can be divided into two groups: syntax-based and semantics-based.

6.1 Syntax-based Methods

Syntax-based approaches [48], [49], [7], [8], [9], [10], [11], [50], [51], [52], [53] enable efficient Android malware detection, but the absence of semantic and contextual messages makes the accuracy of the detection results difficult to guarantee.

For instance, [7], [11], [52] treat permission control as an entry point for Android malware detection, and high-risk permissions are considered as characteristics of malware for detection. Note that *LinRegDroid* [52] significantly improves classification performance by creating different classifiers using the bagging method in ensemble learning.

However, it is difficult to guarantee the accuracy of software detection with a single feature. In response to this issue, *Drebin* [8] employs comprehensive static analysis to extract the maximum number of features from applications. These features are subsequently embedded in a vector space for malicious software detection. Alazab *etal.* [50] selects the most valuable API calls by three grouping strategies (ambiguous group, risky group, and disruptive group) and classifies applications as benign or malicious by examining the distribution of permissions and API call frequency. *Famd* [51] constructs the original feature set by extracting permissions and Dalvik opcode sequences, and obtains dimensionality reduction features to detect malware by *fast correlation-based filter* (FCBF) algorithm. *MFDroid* [53] combine permissions, API calls, and opcodes to form a feature set and train multiple base classifiers using seven classification algorithms. The ensemble learning algorithm then integrates the prediction outcomes of the various base classifiers to get the final classification outcomes. However, the feature extraction of syntax-based methods only treats the source code as text and performs the extraction of specific strings. These methods do not account for the semantic information of the program and they are susceptible to evasion through syntax feature attacks [54], [55].

6.2 Semantics-based Methods

To uphold a strong capability in the detection of Android malware, semantic-based approaches [12], [13], [15], [56], [57], [58], [59], [60], [61], [62], [63], [64], [65], [16], [66], [67] extract the semantics of different types of causal programs through program analysis. Compared to syntax-based approaches, more information significantly improves the accuracy of detection. but also suffers from high time overhead and difficulty in scaling.

For example, *DroidSIFT* [12] extracts features from the weighted contextual API dependency graph to counter transformation attacks. *IntDroid* [67] view the function call graph as a social network and analyze the centrality to filter out the central nodes. The average intimacies between these central nodes and sensitive API calls are then calculated to capture the semantic properties of the graph. *GENDroid* [66] achieves improved accuracy by integrating diverse graph-based classification methods with a majority voting approach. *MaMaDroid* [15] extracts API call sequences from the call graph to establish a Markov chain model and conducts feature extraction for detection. This method demonstrates greater adaptability to API variations and enhances resilience against the evolving landscape of Android malware. Nonetheless, it also come with certain limitations: firstly, it is susceptible to evasion by custom packages that appear similar to Android, Google, or Java packages [54];

secondly, due to the extensive extraction of feature sets and call graphs, considerable memory is required during the detection process.

6.3 Differences From Previous Version

The currently proposed methods all exhibit inherent limitations, making it challenging to achieve rapid malware detection while ensuring rich semantic information. Our prior work, *Malscan*, introduces a graph-based lightweight approach and achieves notable results, but it still presents some drawbacks. Building upon the foundation of prior research, this paper has undertaken enhancements in the following aspects:

Firstly, the previous work selects more than 20,000 sensitive APIs as features to characterize malicious behavior. However, upon conducting further analysis of the resultant feature vectors, we discover that only a part of these sensitive APIs hold significant importance. A multitude of irrelevant APIs not only waste computational resources but also undermine detection efficacy. To provide a precise representation of malicious behavior, we reduce the feature dimension and choose 426 most representative security-related sensitive APIs [35] as features.

Secondly, in our previous work, we confine our selections to a mere 15,285 benign apps and 15,430 malicious apps spanning the years 2011 to 2018 for our experimental dataset. The limited dataset result in the trained model lacking stability and reliability, potentially leading to biased predictions and an inability to accurately capture various scenarios. Additionally, considering the persistent evolution of malicious software, outdated datasets may struggle to precisely identify and capture the latest malicious behaviors, consequently elevating the probabilities of both false positives and false negatives. Hence, we significantly expand our data collection efforts by accumulating a dataset comprising 101,913 benign applications and 109,945 malicious software spanning the period from 2017 to 2021, aiming to enhance detection performance and bolster the credibility of our experiments.

Thirdly, the limitations in the robustness and effectiveness of *MalScan* stem from its narrow focus on just three machine learning techniques and four centrality analysis methods. Consequently, we undertake algorithmic enrichment, ultimately embracing a diverse spectrum encompassing seven centrality analysis techniques and seven machine learning algorithms for the identification of malicious software. We feed the feature vectors generated from distinct centrality analyses into distinct machine learning models, culminating in the creation of 49 distinct weak classifiers, each emphasizing different aspects. Moreover, we introduce the concept of ensemble learning, leveraging a DNN model to integrate predictions from multiple classifiers into a comprehensive and exceptionally strong supervised model. This approach ensures that even if one weak classifier makes an inaccurate prediction, the collaborative effect of others corrects the error.

Finally, to provide a more comprehensive assessment of our proposed malicious software detection system's performance, we incorporate anti-obfuscation experiments and case study experiments into the framework of our previous experiments. On one hand, malware authors frequently employ a variety of obfuscation techniques (*e.g.,* altering code structure, naming conventions) to veil their code and thwart static analysis and detection efforts. Consequently, we select 12 distinct obfuscators from *Obfuscapk* to obfuscate the same dataset samples, followed by conducting malware detection. This methodology enables us to evaluate the system's resilience and adaptability against obfuscation techniques. On the other hand, practical applications demand that malware detection methods adapt to diverse application scenarios and evolving malware variations. Thus, we conduct detection and research on millions of applications from the Google Play Store to authenticate the method's utility and feasibility in real-world settings. Through the analysis results, we finally discover 498 zero-day malware, and they are downloaded more than 97 million times. This means that there is a good chance that millions of users have been infected, causing financial losses. We have reported these malware to Google, hope they can deal with them as soon as possible.

7 Conclusion

Building upon previous work *MalScan*, we propose an enhanced, graph-based lightweight approach for detecting malicious software. For a given test sample, a succinct function call graph is initially extracted, followed by the application of seven centrality analysis algorithms and seven machine learning classification algorithms, generating 49 corresponding labels. Subsequently, a DNN model integrates all labels to produce the final detection outcome. We comprehensively evaluate *MalScan* using the dataset of 213,532 applications sourced from *AndroZoo*. Experimental results demonstrate that *MalScan* can achieve a high accuracy rate of 98%, effectively identifying Android malware within an average processing time of 6.23 seconds. This represents a significant speed improvement of over 20 times compared to two advanced methods, namely *MaMaDroid* and *Drebin*. Additionally, we conduct a statistical study on a million applications from the Google Play Store and identify 498 zero-day malware. This underscores the feasibility of *MalScan* for comprehensive mobile malware scanning across the market.

Acknowledgements

We would thank the anonymous reviewers for their insightful comments to improve the quality of the paper. This work is supported by the Key Program of National Science Foundation of China under Grant No. 62172168.

References

[1] "The mobile malware threat landscape in 2022," https://securelist.com/mobile-threat-report-2022/108844/, 2022.

[2] B. Anderson, D. Quist, J. Neil, C. Storlie, and T. Lane, "Graph-based malware detection using dynamic analysis," *Journal in computer Virology*, vol. 7, pp. 247–258, 2011.

[3] W.-C. Wu and S.-H. Hung, "Droiddolphin: a dynamic android malware detection framework using big data and machine learning," in *Proceedings of the 2014 conference on research in adaptive and convergent systems*, 2014, pp. 247–252.

[4] B. Amos, H. Turner, and J. White, "Applying machine learning classifiers to dynamic android malware detection at scale," in *2013 9th international wireless communications and mobile computing conference (IWCMC)*. IEEE, 2013, pp. 1666–1671.

[5] P. Feng, J. Ma, C. Sun, X. Xu, and Y. Ma, "A novel dynamic android malware detection system with ensemble learning," *IEEE Access*, vol. 6, pp. 30 996–31 011, 2018.

[6] H. Long, Z. Tian, and Y. Liu, "Detecting android malware based on dynamic feature sequence and attention mechanism," in *2021 IEEE 5th International Conference on Cryptography, Security and Privacy (CSP)*. IEEE, 2021, pp. 129–133.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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*, 2014, pp. 1105–1116.

[13] 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.

[14] G. Suarez-Tangil and G. Stringhini, "Eight years of rider measurement in the android malware ecosystem," *IEEE Transactions on Dependable and Secure Computing*, 2020.

[15] 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.

[16] Y. Bai, S. Chen, Z. Xing, and X. Li, "Argusdroid: detecting android malware variants by mining permission-api knowledge graph," *Science China Information Sciences*, vol. 66, no. 9, pp. 1–19, 2023.

[17] Y. Wu, X. Li, D. Zou, W. Yang, X. Zhang, and H. Jin, "Malscan: Fast market-wide mobile malware scanning by social-network centrality analysis," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 139–150.

[18] 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.

[19] Y. Wu, D. Zou, W. Yang, X. Li, and H. Jin, "Homdroid: detecting android covert malware by social-network homophily analysis," in *Proceedings of the 30th acm sigsoft international symposium on software testing and analysis*, 2021, pp. 216–229.

[20] M. M. Alani, A. Mashatan, and A. Miri, "Xmal: A lightweight memory-based explainable obfuscated-malware detector," *Computers & Security*, vol. 133, p. 103409, 2023.

[21] H. Li, S. Zhou, W. Yuan, X. Luo, C. Gao, and S. Chen, "Robust android malware detection against adversarial example attacks," in *Proceedings of the Web Conference 2021*, 2021, pp. 3603–3612.

[22] Y. He, Y. Liu, L. Wu, Z. Yang, K. Ren, and Z. Qin, "Msdroid: Identifying malicious snippets for android malware detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 3, pp. 2025–2039, 2022.

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

[24] 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.

[25] 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.

[26] 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.

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

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

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

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

[31] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer networks and ISDN systems*, 1998.

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

[33] 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.

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

[35] L. Gong, Z. Li, F. Qian, Z. Zhang, Q. A. Chen, Z. Qian, H. Lin, and Y. Liu, "Experiences of landing machine learning onto market-scale mobile malware detection," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020.

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

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

[38] M. E. Newman, "The mathematics of networks," *The new palgrave encyclopedia of economics*, 2008.

[39] J. M. Kleinberg, "Authoritative sources in a hyperlinked environment," *Journal of the ACM (JACM)*, 1999.

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

[41] "Socialnetwork," https://en.wikipedia.org/wiki/Social_network/, 2019.

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

[43] "Tensors and dynamic neural networks in python with strong gpu acceleration (pytorch)," https://pytorch.org, 2021.

[44] S. Aonzo, G. C. Georgiu, L. Verderame, and A. Merlo, "Obfuscapk: An open-source black-box obfuscation tool for android apps," *SoftwareX*, vol. 11, p. 100403, 2020.

[45] C. Gao, G. Huang, H. Li, B. Wu, Y. Wu, and W. Yuan, "A comprehensive study of learning-based android malware detectors under challenging environments," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, 2024, pp. 12:1–12:13.

[46] J. Liu, J. Zeng, F. Pierazzi, L. Cavallaro, and Z. Liang, "Unraveling the key of machine learning solutions for android malware detection," *CoRR*, vol. abs/2402.02953, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2402.02953

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

[48] 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.

[49] 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.

[50] M. Alazab, M. Alazab, A. Shalaginov, A. Mesleh, and A. Awajan, "Intelligent mobile malware detection using permission requests and api calls," *Future Generation Computer Systems*, vol. 107, pp. 509–521, 2020.

[51] H. Bai, N. Xie, X. Di, and Q. Ye, "Famd: A fast multifeature android malware detection framework, design, and implementation," *IEEE Access*, vol. 8, pp. 194 729–194 740, 2020.

[52] D. Ö. Şahın, S. Akleylek, and E. Kiliç, "Linregdroid: detection of android malware using multiple linear regression models-based classifiers," *IEEE Access*, vol. 10, pp. 14 246–14 259, 2022.

[53] X. Wang, L. Zhang, K. Zhao, X. Ding, and M. Yu, "Mfdroid: A stacking ensemble learning framework for android malware detection," *Sensors*, vol. 22, no. 7, p. 2597, 2022.

[54] 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 evad-

ing machine-learning detection," *IEEE Transactions on Information Forensics and Security*, 2019.
[55] 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.
[56] 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.
[57] 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.
[58] 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.
[59] 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.
[60] 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.
[61] 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.
[62] 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.
[63] 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.
[64] 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.
[65] 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.
[66] S. Badhani and S. K. Muttoo, "Gendroid-a graph-based ensemble classifier for detecting android malware," *International Journal of Information and Computer Security*, vol. 18, no. 3-4, pp. 327–347, 2022.
[67] D. Zou, Y. Wu, S. Yang, A. Chauhan, W. Yang, J. Zhong, S. Dou, and H. Jin, "Intdroid: Android malware detection based on api intimacy analysis," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 3, pp. 1–32, 2021.

Yueming Wu received the Ph.D. degree in the School of Cyber Science and Engineering at Huazhong University of Science and Technology, Wuhan, China, in 2021. He is currently an associate professor in the School of Cyber Science and Engineering at Huazhong University of Science and Technology, Wuhan, China. His primary research interests lie in malware analysis and vulnerability analysis.

Wenqi Suo received the master degree in the School of Cyber Science and Engineering at Huazhong University of Science and Technology, Wuhan, China, in 2024. Her primary research interests lie in malware analysis and vulnerability analysis.

Siyue Feng received the bachelor degree in University of Electronic Science and Technology of China (UESTC), in 2021. She is currently a Ph.D. candidate in the School of Cyber Science and Engineering at Huazhong University of Science and Technology, Wuhan, China. Her primary research interests lie in vulnerability analysis.

Deqing Zou received the Ph.D. degree at Huazhong University of Science and Technology (HUST), in 2004. He is currently a professor of School of Cyber Science and Engineering, Huazhong University of Science and Technology (HUST), Wuhan, China. His main research interests include system security, trusted computing, virtualization and cloud security.

Wei Yang received his Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign and his bachelor degree from Shanghai Jiao Tong University. He was a visiting researcher in University of California, Berkeley. His research interests are in software engineering and security. His current primary projects relate to Mobile Security, Software engineering/Security for Machine Learning, Intelligent SE/Security, and IoT/Blockchain Security.

Yang Liu received the Ph.D. degree from NUS and MIT, in 2010. He started his postdoctoral work in NUS and MIT. In 2012, he joined Nanyang Technological University (NTU). He is currently a Full Professor and the Director of the Cybersecurity Laboratory, NTU. He specializes in software verification, security, and software engineering. By now, he has more than 400 publications in top tier conferences and journals.

Hai Jin received the Ph.D. degree in computer engineering from Huazhong University of Science and Technology (HUST), Wuhan, China, in 1994. He is a Cheung Kung Scholars Chair Professor of computer science and engineering at HUST in China. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. He is the chief scientist of ChinaGrid, the largest grid computing project in China, and the chief scientists of National 973 Basic Research Program Project of Virtualization Technology of Computing System, and Cloud Security. He is a fellow of the IEEE, a fellow of the CCF, and a member of the ACM. He has co-authored 22 books and published over 700 research papers. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security.