IntDroid: Android Malware Detection Based on API Intimacy Analysis

DEQING ZOU, YUEMING WU, and SIRU YANG, Huazhong University of Science and Technology, China
ANKI CHAUHAN and WEI YANG, University of Texas at Dallas, USA
JIANGYING ZHONG, SHIHAN DOU, and HAI JIN, Huazhong University of Science and Technology, China

Android, the most popular mobile operating system, has attracted millions of users around the world. Meanwhile, the number of new Android malware instances has grown exponentially in recent years. On the one hand, existing Android malware detection systems have shown that distilling the program semantics into a graph representation and detecting malicious programs by conducting graph matching are able to achieve high accuracy on detecting Android malware. However, these traditional graph-based approaches always perform expensive program analysis and suffer from low scalability on malware detection. On the other hand, because of the high scalability of social network analysis, it has been applied to complete large-scale malware detection. However, the social-network-analysis-based method only considers simple semantic information (i.e., centrality) for achieving market-wide mobile malware scanning, which may limit the detection effectiveness when benign apps show some similar behaviors as malware.

In this article, we aim to combine the high accuracy of traditional graph-based method with the high scalability of social-network-analysis–based method for Android malware detection. Instead of using traditional heavyweight static analysis, we treat function call graphs of apps as complex social networks and apply social-network–based centrality analysis to unearth the central nodes within call graphs. After obtaining the central nodes, the average intimacies between sensitive API calls and central nodes are computed to represent the semantic features of the graphs. We implement our approach in a tool called *IntDroid* and evaluate it on

D. Zou and Y. Wu contributed equally to this research.

This work is supported by the Key Program of National Science Foundation of China under Grant No. U1936211, by the Shenzhen Fundamental Research Program under Grant No. JCYJ20170413114215614 and the Key-Area Research and Development Program of Guangdong Province under Grant No. 2019B010139001.

Authors' addresses: D. Zou is 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, and also with Shenzhen Huazhong University of Science and Technology Research Institute, Shenzhen, 518057, China; email: deqingzou@hust.edu.cn. Y. Wu (corresponding author) is 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; email: wuyueming@hust.edu.cn; A. Chauhan and W. Yang are with University of Texas at Dallas, Dallsa, USA; emails: {axc170043, wei.yang}@utdallas.edu; S. Yang, J. Zhong, S. Dou, and H. Jin are 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; emails: {yangsiru, eileen_zjy, shihandou, hjin}@hust.edu.cn.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2021 Association for Computing Machinery.

1049-331X/2021/05-ART39 $15.00
https://doi.org/10.1145/3442588

a dataset of 3,988 benign samples and 4,265 malicious samples. Experimental results show that *IntDroid* is capable of detecting Android malware with an F-measure of 97.1% while maintaining a True-positive Rate of 99.1%. Although the scalability is not as fast as a social-network-analysis–based method (i.e., *MalScan*), compared to a traditional graph-based method, *IntDroid* is more than six times faster than *MaMaDroid*. Moreover, in a corpus of apps collected from GooglePlay market, *IntDroid* is able to identify 28 zero-day malware that can evade detection of existing tools, one of which has been downloaded and installed by more than ten million users. This app has also been flagged as malware by six anti-virus scanners in VirusTotal, one of which is *Symantec Mobile Insight*.

CCS Concepts: • **Security and privacy → Malware and its mitigation**;

Additional Key Words and Phrases: Android malware, API intimacy, social network, centrality

ACM Reference format:
Deqing Zou, Yueming Wu, Siru Yang, Anki Chauhan, Wei Yang, Jiangying Zhong, Shihan Dou, and Hai Jin. 2021. IntDroid: Android Malware Detection Based on API Intimacy Analysis. *ACM Trans. Softw. Eng. Methodol.* 30, 3, Article 39 (May 2021), 32 pages.
https://doi.org/10.1145/3442588

1 INTRODUCTION

In the second quarter of 2018, Google's Android further extended its lead over Apple's iOS, securing 88 percent market share to 11.9 percent share for iOS [4]. The explosive growth of Android devices and applications (apps) has spurred the growth of Android malware. Millions of apps have been installed by Android users around the world from various app markets. Up to the end of the third quarter of 2018, the number of new malicious apps had an increase of over 40 percent compared to the same period the previous year [3]. Correspondingly, this upsurge in Android malware has resulted in a strong enthusiasm to develop methods for detecting malware automatically.

Generally speaking, existing mobile malware detection approaches can be classified as either syntax-based methods [9, 12, 36, 46, 50, 53, 60] or semantics-based methods [10, 16, 24, 28, 43, 56–58]. Syntax-based techniques, for example, some methods [46, 53] focus on permissions requested by apps and build models for malware detection. However, malware can perform malicious activities without any permissions [29]. Moreover, benign apps may request more permissions than necessary, which can also cause a high false-positive rate [20]. To address this, *Drebin* [12] uses an extensive static analysis to obtain as many features as possible, which consist of both permissions and API calls. However, it can be easily evaded by obfuscation because of the lack of structural and contextual information of the program behaviors. To overcome the challenge, several systems have been proposed to focus on distilling an app's program semantics into a graph representation and perform graph matching to detect malware. Empirical studies [24, 28, 58] have shown the high effectiveness of these graph-based techniques on Android malware detection. Nevertheless, graph matching is typically time-consuming, because a graph often contains thousands of nodes. For instance, on average, the analysis time on an app of *DroidSIFT* [58] and *Apposcopy* [24] is 175.8 s and 275 s, respectively. In other words, these graph-based techniques suffer from low scalability on detecting mobile malware.

To achieve more scalable malware detection, an abstraction-based approach has been proposed, namely, *MaMaDroid* [43], which leverages the abstracted sequences obtained from a call graph to model the app's behaviors. Specifically, it builds a Markov chain model to represent the transitions between method calls, these Markov chains stand for multiple pairwise invocation relationships performed by an app and are used to extract features to complete the classification. The abstraction of method calls and Markov chain modeling in *MaMaDroid* can achieve the robustness and

scalability on detecting mobile malware. However, several design choices may limit the effectiveness of *MaMaDroid*. First, one-order Markov chains (i.e., pairwise invocations) cannot fully reflect dependencies among method calls, thus the approach lacks the key information to distinguish some of the malicious apps from benign ones. Additionally, the coarser-granularity information (i.e., package-level information instead of method-level information) may not accurately differentiate benign apps with malware. For instance, *android.telephony.TelephonyManager.getDeviceId()* and *android.telephony.SmsManager.sendTextMessage()* are both abstracted in *android.telephony* package while their usages and levels of sensitivity are completely different. Such design choices are understandable, because using whole-graph analysis (instead of only considering pairwise invocations) and finer-granularity information (i.e., method-level information) may incur higher costs, thus making the large-scale malware scanning infeasible.

To address the above issues, we present *MalScan* in our prior work [54] to complete market-wide mobile malware scanning. *MalScan* regards the function call graph of an app as a complex social network and apply social-network-centrality analysis on sensitive API calls to represent graph semantics for malware detection. However, only using the centrality of sensitive API calls to conduct malware detection may cause some false positives when benign apps show some similar behaviors as malware by invoking sensitive API calls. In such case, the centrality of certain sensitive API calls in benign apps may be almost the same as in some malware. For example, some social apps (e.g., Tiktok) require to access users' location for presenting location-specific news or videos and read users' address book for recommending new friends. In the situation, the centrality of sensitive API call *LocationManager.getLastLocation()* may be almost the same as some malware, which can cause a false positive.

In conclusion, on the one hand, traditional graph-analysis–based methods [24, 28, 43, 58] can achieve high effectiveness on Android malware detection because of the consideration of different types of program semantics. However, the efficiency of these methods is not ideal, since a graph often contains thousands of nodes, resulting in low scalability on malware detection. On the other hand, social-network-analysis–based method [54] is able to complete market-wide Android malware scanning because of the high scalability of social network analysis[1]. However, *MalScan* [54] only considers the centrality of sensitive API calls, such simple consideration may limit the detection effectiveness when apps show some similar behaviors as malware by invoking sensitive API calls. Therefore, we raise a research question:

Is there any way to combine the high effectiveness of traditional graph-analysis–based method with the high scalability of social-network-analysis–based method for Android malware detection?

In this article, we aim to address the raised research question. To achieve the combination and balance between traditional graph-analysis–based method and social-network-analysis–based method, we first leverage social-network-centrality analysis on the whole graph to excavate the most important nodes (i.e., central nodes). Then traditional graph analysis is applied to compute the intimacy between these nodes and sensitive API calls as the semantic features, which provide more effective graph details to distinguish malware from benign apps. Our key insight is derived from an observation from daily-life social networks. We have found that a person in different social networks may present different intimacies (i.e., communication frequencies) between she and the central person in corresponding social networks. In the context of malware detection, due to the different inherent goals of benign apps (e.g., delivering utility) and malware (e.g., maximizing profits, prolonging lifetime) [57], an API method may have different communication frequencies with central nodes in different function call networks.

[1]Social network analysis can process networks with millions of nodes (e.g., Twitter, Facebook).

Specifically, to maintain the program semantics, we first extract the function call graph of an app based on static analysis. Given a call graph, we then apply centrality analysis to unearth the central nodes within the graph. The concepts of centrality were first proposed in social network analysis whose original objective is to quantify the importance of a vertex in the network. We leverage the analysis result of method calls' centralities to find nodes with high centrality (i.e., central nodes). Moreover, as malware always invoke some sensitive API calls to perform malicious activities, we pay attention to these API calls. Therefore, after obtaining the central nodes within the graph, we perform intimacy analysis to compute the average intimacies between sensitive API calls and central nodes. Our definition of intimacy between two nodes within a call graph consists of two influencing factors: (1) the number of reachable paths and (2) the average path distance. In other words, the higher the number of reachable paths and the shorter the average path distance, the higher the intimacy between the two nodes. These calculated intimacies are used as features and fed into a machine learning classifier to train a model and perform classification on a newly given feature vector.

We implement a prototype system, *IntDroid*, and evaluate it using 3,988 benign samples and 4,265 malicious samples. Experimental results show that *IntDroid* is capable of detecting Android malware with an F-measure of 97.1% while the True-positive Rate is able to maintain 99.1%. As for scalability, *IntDroid* is not as fast as social-network-analysis–based method (i.e., *MalScan* [54]) because of the process of intimacy analysis, however, compared to a state-of-the-art traditional graph-based method (i.e., *MaMaDroid* [43]), the time overhead of *IntDroid* is more than six times less than it. We also examine the ability of *IntDroid* on detecting zero-day malware. Specifically, in a corpus of apps collected from GooglePlay market, *IntDroid* is able to identify 28 zero-day malware that can evade detection of existing tools, one of which has been downloaded and installed by more than ten million users. This app has also been flagged as malware by six anti-virus scanners in VirusTotal, one of which is *Symantec Mobile Insight*.

In summary, this article makes the following contributions:

- We propose a novel method to perform detection on Android malware by analyzing the intimacies between sensitive API calls and central nodes within function call graphs.
- We design and implement a prototype system, *IntDroid*, to combine the high effectiveness of traditional graph-analysis–based method with the high scalability of social-network-analysis–based method to detect Android malware.
- We conduct evaluations using 3,988 benign samples and 4,265 malicious samples. Experimental results show that *IntDroid* is capable of detecting Android malware with an F-measure of 97.1% and a True-positive Rate of 99.1%. Moreover, compared to a traditional graph-based method, *IntDroid* is more than six times faster than *MaMaDroid* [43].

Article organization. The remainder of the article is organized as follows. Section 2 presents our motivation. Section 3 shows the definitions. Section 4 introduces our system. Section 5 reports the experimental results. Section 6 presents the extensive comparison with *MalScan*. Section 7 discusses the future work and limitations. Section 8 describes the related work. Section 9 concludes the present article.

2 MOTIVATION

To understand the key insight, we examine the API-call network of a benign app and a malicious app. The benign app is a mobile security app for a safer Android experience while the malicious app allows a trace of lost mobile phones.

We analyzed the API calls for both apps and created function call graphs. The function call graph of these apps can be treated as an API-call network. The nodes represent functions and the edges

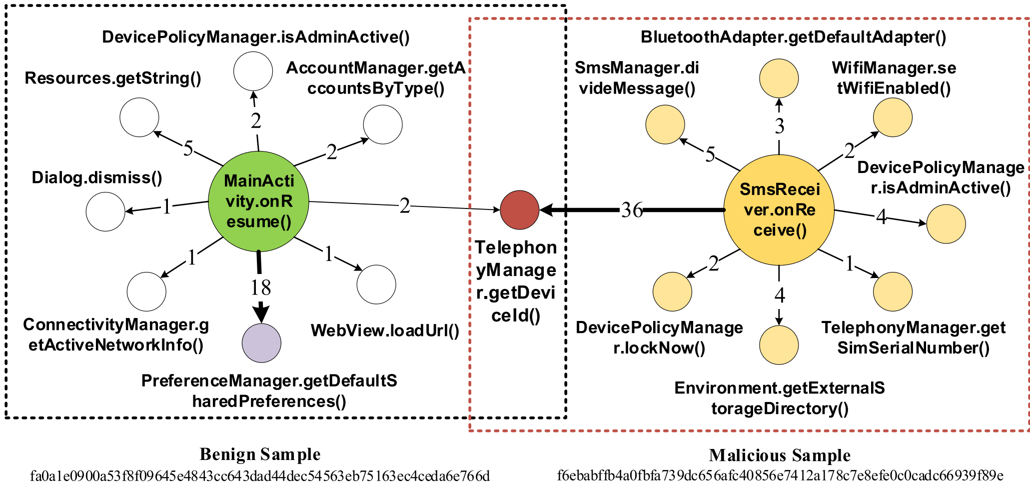

Fig. 1. The relationships of two assumable networks, which are a benign app API-call network and a malicious app API-call network. The number on an edge represents the frequency of communication between the two nodes. The higher the number, the more frequent are the calls.

Table 1. The Frequency of Communication between *PreferenceManager.getDefaultSharedPreferences()*, *TelephonyManager.getDeviceId()* and *MainActivity.onResume()*, *SmsReceiver.onReceive()* in Figure 1

API Calls	*MainActivity.onResume()*	*SmsReceiver.onReceive()*
PreferenceManager.getDefaultSharedPreferences()	18 (high)	0 (low)
TelephonyManager.getDeviceId()	2 (low)	36 (high)

represent communications between them. The direction of the edge is to help distinguish the caller and callee in the graph. For example, in Figure 1, an edge from node *MainActivity.onResume()* to node *PreferenceManager.getDefaultSharedPreferences()* represents a call path from function *MainActivity.onResume()* to function *PreferenceManager.getDefaultSharedPreferences()*. For each API-call network, we select a central node based on node degree.

As shown in Figure 1, the benign app network is centered around node *MainActivity.onResume()* with outward edges to *PreferenceManager.AccountManager()*, and so on. The malicious app network is centered around node *SmsReceiver.onReceive()* with outward edges to *TelephonyManager. DevicePolicyManager()*, and so on. Functions *MainActivity.onResume()* and *SmsReceiver.onReceive()* are lifecycle methods, as a result they have high degree. App developers override these functions for specific implementations, which usually involve many calls to other API calls. Many of these API calls fall under sensitive methods [13], in which case, we refer to them as sensitive API calls. The weight of an edge in Figure 1 represents the frequency of communication between the two nodes. The higher the weight, the more frequent are the calls from the center node to the API.

Table 1 presents the communication frequency between nodes *PreferenceManager. getDefaultSharedPreferences()*, *TelephonyManager.getDeviceId()* and *MainActivity.onResume()*, *SmsReceiver.onReceive()* from Figure 1, respectively. If we define intimacy as the frequency of communication between two functions, then from the results presented in Figure 1 and Table 1, we can see that, due to the different objectives of *PreferenceManager.getDefaultSharedPreferences()* and *TelephonyManager.getDeviceId()*, the intimacies between them and center nodes are different. Based on the observation, we raise a question:

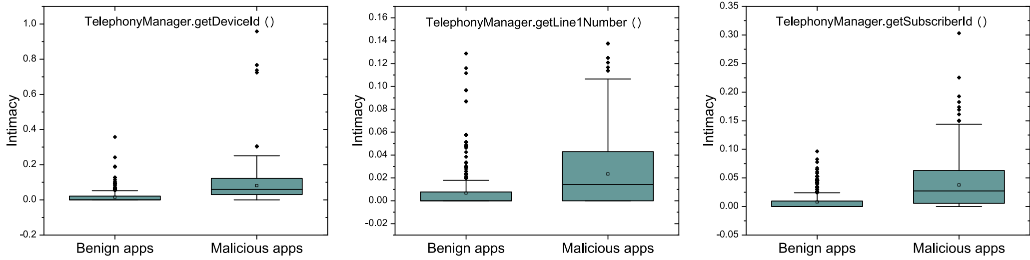

Fig. 2. The average intimacies between sensitive API calls and central nodes in benign apps and malicious apps.

Do the intimacies between sensitive API calls and central nodes in malicious apps vary from the intimacies between sensitive API calls and central nodes in benign apps?

To answer the proposed question, we randomly select 500 benign apps and 500 malicious apps from AndroZoo [11]. Then the function call graphs are extracted based on static analysis. Given a call graph, we select the nodes with top 1% degree as central nodes within a call graph. After obtaining the central nodes in a call graph, we then compute the *intimacy* between sensitive API calls and central nodes. As aforementioned, we simply define the *intimacy* as the communication frequency between two persons in a social network. Therefore, to define a more precise *intimacy* between two nodes in a function call graph, we select the following two influencing factors: (1) the number of reachable paths and (2) the average path distance. In other words, the higher the number of reachable paths and the shorter the average path distance, the higher the intimacy between the two nodes. A more detailed real-world example is presented in Section 4.

API calls with the prefix *android.telephony.TelephonyManager.get* from a list of security-sensitive methods [13] are selected as our test objects. After computing all the intimacies between a sensitive API call and central nodes within a call graph, we then collect the average value of these intimacies. As a matter of fact, due to the small size of our randomly selected dataset, which consists of only 500 benign apps and 500 malicious apps, some sensitive API calls do not appear in these apps. In such case, the average intimacy values are all zero, therefore, we only consider API calls that are invoked by both benign apps and malicious apps. Finally, the total number of satisfied API calls with the prefix *android.telephony.TelephonyManager.get* in PScout [13] is 14. Due to the limitation of the page, we only show a portion of our results. Figure 2 shows the average intimacy distributions of three sensitive API calls: (1) *TelephonyManager.getDeviceID()* can get your phone's *International Mobile Equipment Identity* (IMEI), (2) *TelephonyManager.getLine1Number()* can obtain your phone number, and (3) *TelephonyManager.getSubscriberld()* can gain your phone's *International Mobile Subscriber Identification Number* (IMSI). Results in Figure 2 indicate that there are differences in the average intimacies between sensitive API calls and the central nodes within a call graph in benign apps and malicious apps.

To obtain more determinate results, we first perform *Shapiro-Wilk test* [51] to check whether the average intimacies are normally distributed or not. After obtaining the test result, we find that these average intimacies between sensitive API calls and central nodes do not fit the normal distribution. Therefore, we adopt a non-parametric test (i.e., *Mann-Whitney U test* [41]), which is robust against deviation from normality to commence our research. The null hypothesis of *Mann-Whitney U test* is that the two populations have same means and the alternative hypothesis is that the means of two populations are different.

The type I error rate or significance level is the probability of rejecting the null hypothesis given that it is true. The null hypothesis is rejected if the calculated p-value is less than a pre-determined

Table 2. P-values of Average Intimacies between 14 API Calls with the Prefix *android.telephony.TelephonyManager.get* from PScout [13] and Central Nodes in Benign Apps and Malicious Apps

API Calls	P-values
TelephonyManager.getSimOperator()	0.054473
TelephonyManager.getSimState()	0.014292
TelephonyManager.getNetworkType()	0.004563
TelephonyManager.getNetworkCountryIso()	0.000743
TelephonyManager.getSimOperatorName()	3.02E-05
TelephonyManager.getPhoneType()	6.81E-18
TelephonyManager.getSimSerialNumber()	8.43E-29
TelephonyManager.getLine1Number()	5.36E-31
TelephonyManager.getDeviceID()	3.59E-36
TelephonyManager.getSimCountryIso()	6.09E-47
TelephonyManager.getNetworkOperator()	1.36E-55
TelephonyManager.getSubscriberId()	5.43E-63
TelephonyManager.getCellLocation()	9.75E-97
TelephonyManager.getNetworkOperatorName()	5.6E-118

threshold value α, which is referred to as the level of significance. Usually, the significance level is set to be 0.05, implying that it is acceptable to have a 5% probability of incorrectly rejecting the true null hypothesis. We experiment with these API calls to examine the difference of API intimacy between benign apps and malicious apps by using *Mann-Whitney U test*. Table 2 presents the p-values of 14 API calls with the prefix *android.telephony.TelephonyManager.get* on average intimacies between benign apps and malicious apps. As shown in Table 2, we can see that most of the p-values are less than 0.05. In other words, as for most of API calls in Table 2, we can reject the null hypothesis.

Therefore, based on the observation, we propose and develop an automatic Android malware detection system by analyzing the average intimacies between sensitive API calls and central nodes within a function call graph.

3 DEFINITIONS

Before introducing our proposed method, we first describe the formal definitions that we use throughout the article.

3.1 Centrality

Centrality concepts were first developed in social network analysis, which quantify the importance of a node in the network. Centrality measures are very useful for network analysis, and many studies have been proposed to use centrality measures in different areas, such as biological network [34], co-authorship network [39], transportation network [30], criminal network [18], affiliation network [22], and so on. There has been proposed several definitions of centrality in a social network, for example:

Definition 1. Degree centrality [26] of a node is the fraction of nodes it is connected to. The degree centrality values are normalized by dividing by the maximum possible degree in a graph

N-1 where N is the number of nodes in the graph:

$$x_i = \frac{deg(i)}{N-1}.$$

Note that $deg(i)$ is the degree of node i.

Definition 2. Katz centrality [35] computes the centrality for a node based on the centrality of its neighbors. The *katz centrality* for node i is

$$x_i = \alpha \sum_j A_{ij} x_j + \beta.$$

Note that A is the adjacency matrix of the graph G with eigenvalues λ. The parameter β controls the initial centrality and

$$\alpha < \frac{1}{\lambda_{max}}.$$

katz centrality computes the relative influence of a node within a graph by measuring the number of the immediate neighbors (first degree nodes) and also all other nodes in the graph that connect to the node under consideration through these immediate neighbors.

Definition 3. Closeness centrality [26] indicates how close a node is to all other nodes in the network. It is calculated as the average of the shortest path length from the node to every other nodes in the graph. The smaller the average shortest distance of a node, the greater the closeness centrality of the node. In other words, the average shortest distance and the corresponding closeness centrality are negatively correlated:

$$x_i = \frac{N-1}{\sum_{i \neq j} d(i,j)}.$$

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

Definition 4. Harmonic centrality [42] reverses the sum and reciprocal operations in the definition of closeness centrality:

$$x_i = \frac{\sum_{i \neq j} \frac{1}{d(i,j)}}{N-1}.$$

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

3.2 Intimacy

On the one hand, if there are more reachable paths between two functions, then the "communication" between these two functions can be considered frequent. On the other hand, if the distance between two functions is shorter, then the "communication" between them can be considered easy. In this article, if the "communication" between two functions is frequent and easy, then they will be treated as a close pair.

Definition 5. Given a function call graph $G = (V, E)$, and two functions $a, b \in V$, then the *intimacy* between a and b is defined as

$$intimacy(a,b) = \frac{n}{ad(a,b)+1}.$$

Note that n denotes the number of reachable paths between a and b, *ad(a, b)* denotes the average distance of these reachable paths. For instance, suppose that there are two reachable paths between a and b: $a \rightarrow p \rightarrow q \rightarrow b$ and $a \rightarrow m \rightarrow b$. Then the number of reachable paths

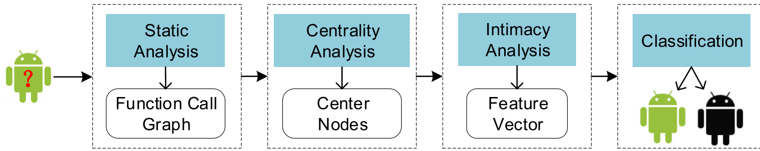

Fig. 3. System overview of *IntDroid*.

n is 2 and the average distance $ad(a, b)$ is (3+2)/2=2.5. Therefore, the intimacy between a and b can be computed as 2/(2.5+1)=0.57. Moreover, there is a special case when the central node is a sensitive API call. In such case, the average distance between the API call and the central node will be 0, and this is the reason why the denominator plus 1 in the definition of intimacy.

4 SYSTEM ARCHITECTURE

In this section, we introduce *IntDroid*, an automatic Android malware detection system based on intimacy analysis between sensitive API calls and central function calls within a function call graph. Algorithm 1 presents the whole procedures of our system.

4.1 Overview

As shown in Figure 3, *IntDroid* consists of four main phases: *Static Analysis*, *Centrality Analysis*, *Intimacy Analysis*, and *Classification*.

- ***Static Analysis:*** This phase aims at extracting the function call graph of an app based on static analysis, in which, each node is a function that can be an API call or a user-defined function.
- ***Centrality Analysis:*** After obtaining the function call graph, we then calculate the centralities of all nodes within the call graph. Nodes with top $n\%$[2] centrality will be selected as central nodes.
- ***Intimacy Analysis:*** Next, we compute the average intimacies between sensitive API calls and central nodes within the function call graph. The output of this phase is the feature vector.
- ***Classification:*** In the final phase, given the feature vector, we can accurately and efficiently classify the app as either benign or malicious by using a machine learning classifier.

4.2 Static Analysis

In this article, we aim to combine the effectiveness of graph-based method with the scalability of social-network-analysis–based method. Therefore, we extract a succinct function call graph by performing low-cost program analysis (e.g., context- and flow-insensitive analysis) on given APK files. Specifically, we implement the static analysis based on an Android reverse engineering tool, Androguard [19].

To better illustrate the different phases involved in our system, we choose a real-world malware sample.[3] Figure 4 shows the sample's function call graph, in which, each node is an API call or a user-defined function. The number of nodes and edges are 140 and 251, respectively.

[2] The value of n is 1 to 8 in our system.
[3] 2af4c588b447963118fd0a8a984438f64898efb0abd01aa6c65dad88d95c7880.

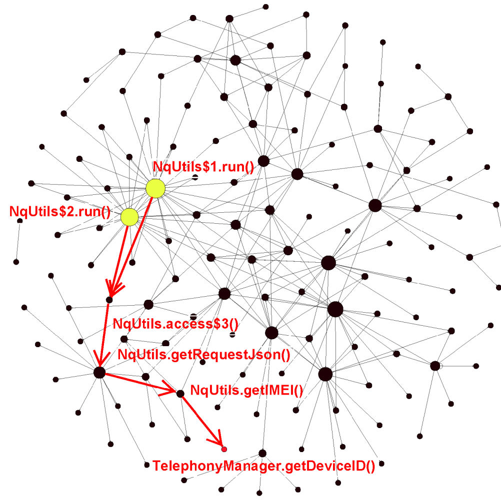

Fig. 4. Function call graph of a real-world malicious app (com.qiacal.paobe).

ALGORITHM 1: Extract the average intimacy between sensitive API calls and central nodes within a call graph

Require: A: An APK file; t: The type of centrality measure for excavating the central nodes; n: Nodes with top $n\%$ centrality will be selected as central nodes; S: The list of sensitive API calls.

Ensure: AI: The average intimacy between sensitive API calls and central nodes.

1: *CG* ← *extractCallGraph(A)*
2: *V* ← *obtainNodes(CG)*
3: **for** each $v \in V$ **do**
4: *centrality* ← *computeCentrality(CG, v, t)*
5: *Centralities.add(centrality)*
6: **end for**
7: **for** each $v \in V$ **do**
8: *ranking* ← *computeRanking(Centralities, v)*
9: **if** ranking/len(V) $\leq n\%$ **then**
10: *CenterNodes.add(v)*
11: **end if**
12: **end for**
13: **for** each $s \in S$ **do**
14: **for** each $c \in CenterNodes$ **do**
15: *i* ← *computeIntimacy(CG, s, c)*
16: *I.add(i)*
17: **end for**
18: *ai* ← *computeAverageIntimacy(I)*
19: *AI.add(ai)*
20: **end for**
21: **return** *AI*

4.3 Centrality Analysis

As aforementioned, we regard the function call graph as a social network. Therefore, in this phase, our objective is to dig out the most important "persons" in a call graph social network. This phase corresponds to Lines 2–12 in Algorithm 1.

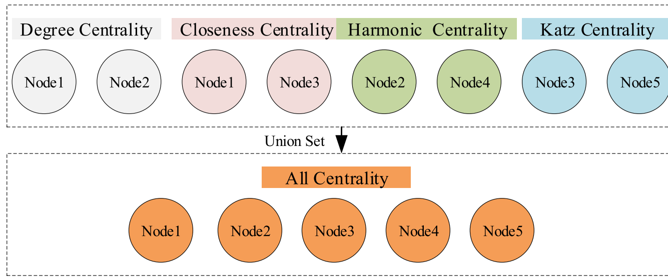

Fig. 5. Construction of all centrality by computing the union set of four individual results of central nodes.

As centrality measures can indicate the importance of a node within a network, we perform centrality analysis to select central nodes in a call graph. Given a call graph, we compute the centralities of all nodes in the graph. Nodes with top $n\%$ centralities will be treated as central nodes. To conduct more comprehensive experiments, we take a total of eight different values of n, which are 1, 2, 3, 4, 5, 6, 7, and 8. Moreover, as for centrality measures, we select degree centrality, *katz* centrality, closeness centrality, and harmonic centrality to commence our experiments. Additionally, the importance of a vertex is generally to be measured by using multiple centralities, since different centralities measure the importance of a vertex from different aspects. Therefore, for the completeness of our research, we construct other two centralities by integrating the four individual centrality measures. The one is average centrality by computing the average value of the former four centrality measures and the other is all centrality by incorporating the central nodes obtained by four individual centrality measures. As Figure 5 shows, suppose that we select node1 and node2 as the central nodes in Figure 4 after degree centrality analysis. Similarly, the central nodes are (node1, node3), (node2, node4), and (node3, node5) after closeness centrality, harmonic centrality, and *katz* centrality analysis, respectively. Then the union set of these four results of central nodes is computed as the central nodes of all centrality, which are node1, node2, node3, node4, and node5.

As shown in Figure 4, we first compute the degree centralities of all nodes. To present more clearly in Figure 4, we use degree centrality as the weight of a node size (i.e., the greater the degree centrality, the larger the node). The number of nodes of the function call graph in Figure 4 is 140, therefore, we choose the nodes with top 2[4] degree centrality as central nodes, namely, *NqUtils$1.run()* and *NqUtils$2.run()*. We mark these two nodes as yellow to distinguish them from other nodes in Figure 4.

4.4 Intimacy Analysis

Since API calls are used by the Android apps to access operating system functionality and system resources, they can be used as representations of the behaviors of Android apps. Particularly, Android malware usually invokes some security-related API calls to perform malicious activities. For instance, *getDeviceID()* can get your phone's IMEI and *getLine1Number()* can obtain your phone number. Therefore, to characterize malicious behaviors, we focus on these security-related API calls, namely, sensitive API calls on the basis of the result reported by PScout [13], which consists of 21,986 sensitive API calls. Lines 13–21 in Algorithm 1 shows intimacy analysis.

After computing all the intimacies between a sensitive API call and central nodes, we use the average value of these intimacies as the corresponding feature of this sensitive API call. For instance, we select *NqUtils$1.run()* and *NqUtils$2.run()* as central nodes after degree centrality analysis

[4] $\lceil 140/100 \rceil = 2$.

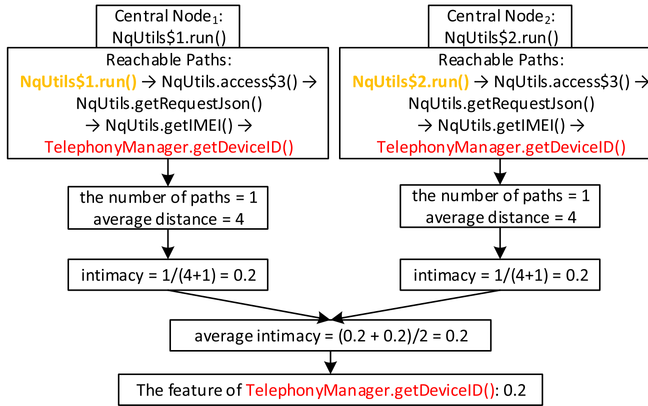

Fig. 6. Detailed procedures to compute the feature (i.e., average intimacy) of sensitive API call *TelephonyManager.getDeviceID()* in Figure 4.

and mark them as yellow in Figure 4. We choose a sensitive API call *TelephonyManager.getDeviceID()* as an example to illustrate the detail steps of *Intimacy Analysis*. As shown in Figure 6, we first perform reachability analysis to obtain the reachable paths between *TelephonyManager.getDeviceID()* and central nodes (i.e., *NqUtils$1.run()* and *NqUtils$2.run()*). Results in Figure 4 show that there is one path between *TelephonyManager.getDeviceID()* and *NqUtils$1.run()*: *NqUtils$1.run()* → *NqUtils.access$3()* → *NqUtils.getRequestJson()* → *NqUtils.getIMEI()* → *TelephonyManager.getDeviceID()*. That is, the number of reachable paths is one (i.e., n=1) and the average distance is four (i.e., ad=4). Therefore, the intimacy between *TelephonyManager.getDeviceID()* and *NqUtils$1.run()* is 1/(4+1)=0.2. Similarly, there is also one path between *TelephonyManager.getDeviceID()* and another central node *NqUtils$2.run()*, and the intimacy between *TelephonyManager.getDeviceID()* and *NqUtils$2.run()* is also 1/(4+1)=0.2. Then the average value of these two intimacies can be calculated as (0.2+0.2)/2=0.2, in other words, the feature of *TelephonyManager.getDeviceID()* is 0.2. In addition, other sensitive API calls that do not appear in the function call graph are represented as 0 in the feature vector.

4.5 Classification

Our final phase focuses on classification, i.e., labeling apps as either benign or malicious. For this purpose, we select three different classification algorithms: *1-Nearest Neighbor* (1-NN), *3-Nearest Neighbor* (3-NN), and *Random Forest* (RF) to complete the classification. Feature vectors extracted from a training dataset are fed into a learning model to train a classifier and then performing classification on a testing dataset.

5 EVALUATIONS

In this section, we aim to answer the following research questions:

- *RQ1: What is the effectiveness of IntDroid on detecting Android malware from different aspects?*
- *RQ2: What is the effectiveness of IntDroid compared to other state-of-the-art Android malware detection methods?*
- *RQ3: What is the runtime overhead of IntDroid on detecting Android malware?*
- *RQ4: Can IntDroid detect zero-day malware?*

Table 3. Summary of the Dataset Used in Our Experiments

Category	#Apps	Average Size (MB)
Benign apps	3,988	3.45
Malicious apps	4,265	3.97
Total	8,253	3.72

Table 4. Descriptions of the Used Metrics in Our Experiments

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

5.1 Datasets and Metrics

Our dataset used to evaluate *IntDroid* includes 8,253 samples that are available in github,[5] by this researchers can conduct reproducible experiments. We crawled these APK files from Andro-Zoo [11], which currently contains over ten million APK files and each of which has been detected by several different anti-virus products in VirusTotal [7]. We leverage the detection reports to filter and generate our dataset. An APK file is considered benign only if all its reports classify it as non-malicious. As for malicious samples, we adopt the selection method in *Drebin* [12], that is, an app is collected when it is flagged as malicious by more than two anti-virus scanners whose purpose is to generate a more accurate dataset. Our final dataset includes 3,988 benign apps and 4,265 malicious apps. Table 3 lists the summary of our dataset. The largest benign sample size is 45.81 MB, and the smallest benign sample size is 67 KB. As for malware, the largest sample size is 69.69 MB while the smallest sample size is 54 KB.

To evaluate *IntDroid*, we conduct experiments by performing 10-fold cross validations using the dataset, which means that the dataset is partitioned into ten subsets, each time we pick one subset as our testing set, and the rest nine subsets as training set. We repeat this ten times and report the average as the final results. The metrics used to measure the effectiveness of *IntDroid* are shown in Table 4.

5.2 Detection Effectiveness

We first evaluate how effective *IntDroid* is on detecting Android malware. To this end, we perform 10-fold cross validations on our dataset. Particularly, we evaluate the effectiveness of *IntDroid* from the following three aspects:

[5] https://github.com/IntDroid/IntDroid.

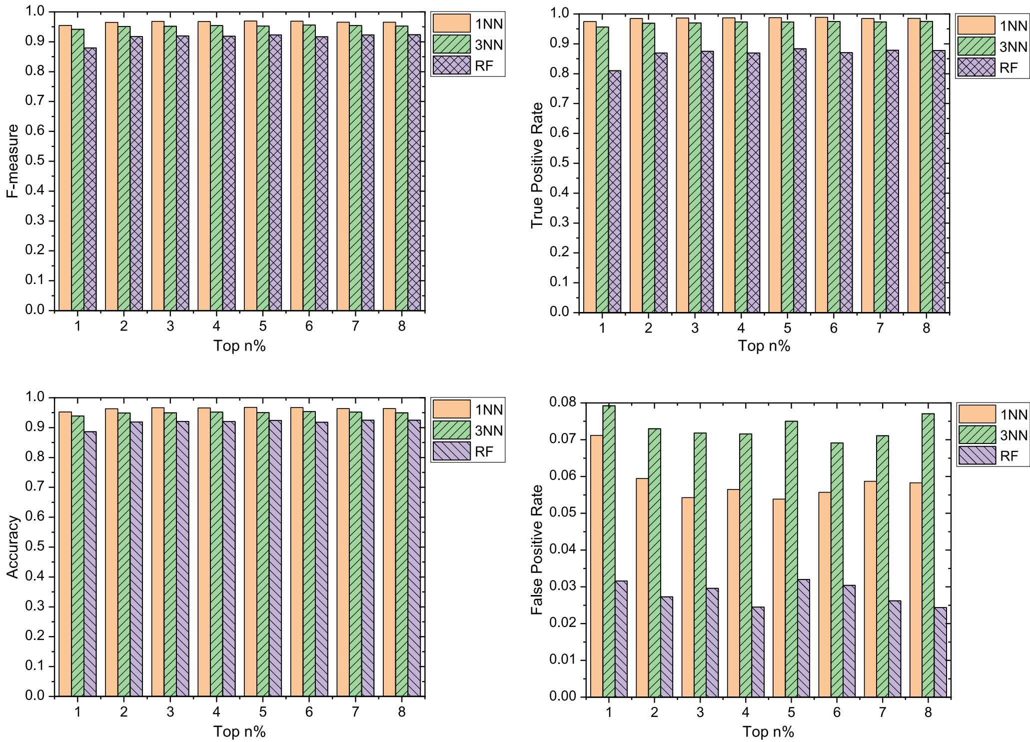

Fig. 7. F-measure, Accuracy, *True-positive Rate* (TPR), and *False-positive Rate* (FPR) of *IntDroid* on detecting Android malware by using degree centrality with different classification models.

- *Different classification models: 1NN, 3NN, and Random Forest.*
- *Different centrality measures: degree centrality, closeness centrality, harmonic centrality, katz centrality, average centrality, and all centrality.*
- *Nodes with different top n% centrality to select as central nodes: 1, 2, 3, 4, 5, 6, 7, and 8.*

5.2.1 Different Classification Models. We first conduct a series of experiments using different machine learning algorithms as discussed in Section 4.5. Specifically, we run *IntDroid* using *1-Nearest Neighbor* (1NN), *3-Nearest Neighbor* (3NN), and *Random Forest* (RF). These three classifiers are implemented by using a python library scikit-learn [6]. For the RF, we adopt the default parameters to commence our experiments.[6] Figure 7 presents the F-measure, Accuracy, *True-positive Rate* (TPR), and *False-positive Rate* (FPR) achieved by *IntDroid* when we choose degree centrality to pick out the central nodes within call graphs. The results in Figure 7 indicate that 1NN is able to maintain the best effectiveness on all experiments in terms of F-measure, TPR, and Accuracy. For instance, when we select nodes with top 3% degree centralities as central nodes, the F-measure of *IntDroid* is 96.8% with 1NN while is 95.2% and 92.0% by using 3NN and RF, respectively. As for TPR, when *IntDroid* uses 1NN for classification, it can achieve better performance than uses 3NN and RF. As shown in Figure 7, TPRs of *IntDroid* with 1NN range from 97.4% to 98.9% while range from 81.0% to 88.3% with RF. As for FPR, *IntDroid* is able to detect more benign apps when we use RF to train our classifier, however, the TPR is very low, which indicates that more malicious samples are being misclassified as benign apps. As a matter of fact, it is very important to maintain

[6] More detailed information of parameters is available on the official website: https://scikit-learn.org/.

Table 5. F-measure, Accuracy, *True-positive Rate* (TPR), and *False-positive Rate* (FPR) of *IntDroid* on Detecting Android Malware by Using Different Centralities with 1NN

Top *n*%	1		2		3		4		5		6		7		8	
Metrics	F1	TPR	F1	TPR	F1	TPR	F1	TPR	F1	TPR	F1	TPR	F1	TPR	F1	TPR
Degree	95.5	**97.4**	96.5	**98.5**	96.8	98.6	**96.8**	**98.7**	**96.9**	**98.7**	96.9	**98.9**	96.6	**98.5**	**96.6**	**98.5**
Closeness	88.6	89.5	92.0	93.8	92.9	95.6	93.0	95.7	93.1	95.5	93.6	95.8	93.6	95.6	93.8	95.5
Harmonic	88.3	90.0	91.7	93.9	92.7	95.1	93.2	95.5	93.8	96.3	93.4	95.6	93.9	96.0	94.4	96.1
Katz	90.4	93.0	92.0	94.5	93.2	95.2	93.7	95.6	94.1	96.2	94.3	96.1	94.7	96.5	94.8	97.0
Average	88.1	89.9	91.7	94.0	92.7	95.0	93.2	95.5	93.6	96.2	93.6	95.8	94.0	96.3	94.2	95.8
All	**95.6**	**97.4**	**96.6**	**98.5**	**97.1**	**99.1**	96.7	98.5	96.6	98.2	**97.0**	98.7	**96.7**	98.4	96.5	98.2
Metrics	A	FPR	A	FPR	A	FPR	A	FPR	A	FPR	A	FPR	A	FPR	A	FPR
Degree	95.2	7.1	96.3	5.9	96.7	**5.4**	**96.6**	5.6	**96.8**	5.4	96.7	5.6	96.4	5.9	**96.4**	5.8
Closeness	88.1	13.4	91.5	10.9	92.4	11.0	92.5	10.9	92.7	10.3	93.2	9.7	93.3	9.3	93.5	8.6
Harmonic	87.6	14.8	91.2	11.8	92.2	10.8	92.8	10.1	93.4	9.7	93.0	9.7	93.6	9.0	94.1	8.1
Katz	89.8	13.6	91.5	11.6	92.8	9.8	93.4	8.9	93.7	8.9	93.9	8.4	94.4	7.8	94.5	8.1
Average	87.4	15.3	91.2	11.8	92.3	10.7	92.8	10.1	93.3	9.8	93.3	9.5	93.7	9.1	93.9	8.2
All	**95.3**	**6.8**	**96.4**	**5.7**	**96.9**	**5.4**	96.5	**5.5**	96.5	**5.3**	**96.8**	**5.2**	**96.5**	**5.5**	96.3	**5.7**

a high TPR for Android malware detection, because low TPR signifies more malicious samples being misclassified as benign samples, and these misclassified malicious samples can still spread malicious activities. When users install these evasive malware, their private data may be stolen by attackers, which may cause different levels of economic losses. In conclusion, *IntDroid* is able to obtain better results when we adopt 1NN to train a classifier and use it to detect Android malware.

5.2.2 Different Centrality Measures. To test the effectiveness of *IntDroid* on detecting Android malware with using different centralities to select central nodes, we first conduct four experiments by adopting the following four centralities: degree centrality, closeness centrality, harmonic centrality, and *katz* centrality. In addition, it is general to measure the importance of a vertex in a social network by combining multiple centrality measures. Therefore, we add an average centrality experiment by using the average value of the former four individual centralities to pick out central nodes within call graphs. Moreover, we also construct another experiment (i.e., all centrality) by computing the union set of the former four individual results of central nodes (Figure 5).

Table 5 and Figure 8 present the experimental results of *IntDroid* with 1NN, which include F-measure, Accuracy, TPR, and FPR for each experiment. Results in Table 5 and Figure 8 indicate that *IntDroid* can obtain better effectiveness when we select degree centrality and all centrality to dig out central nodes. For instance, the F-measures of *IntDroid* are 96.8% and 97.1% when the centrality measures are degree centrality and all centrality, while they are 92.9%, 92.7%, 93.2%, and 92.7% when we choose closeness centrality, harmonic centrality, *katz* centrality, and average centrality to pick out central nodes within call graphs. As for TPR, as shown in Table 5 and Figure 8, *IntDroid* is able to maintain a TPR of 99.1% when we adopt all centrality to extract central nodes. In conclusion, *IntDroid* can achieve better effectiveness when we adopt degree centrality and all centrality to unearth central nodes within call graphs.

5.2.3 Different Top n*%.* The larger *n*, the more central nodes, resulting in greater runtime overheads (Section 5.4). Therefore, we only choose eight different values of *n* and exclude the larger numbers. As shown in Figure 8, F-measure, Accuracy, TPR, and FPR vary according to the value of *n*. As for degree centrality and all centrality, the detection results change slowly as *n* increases.

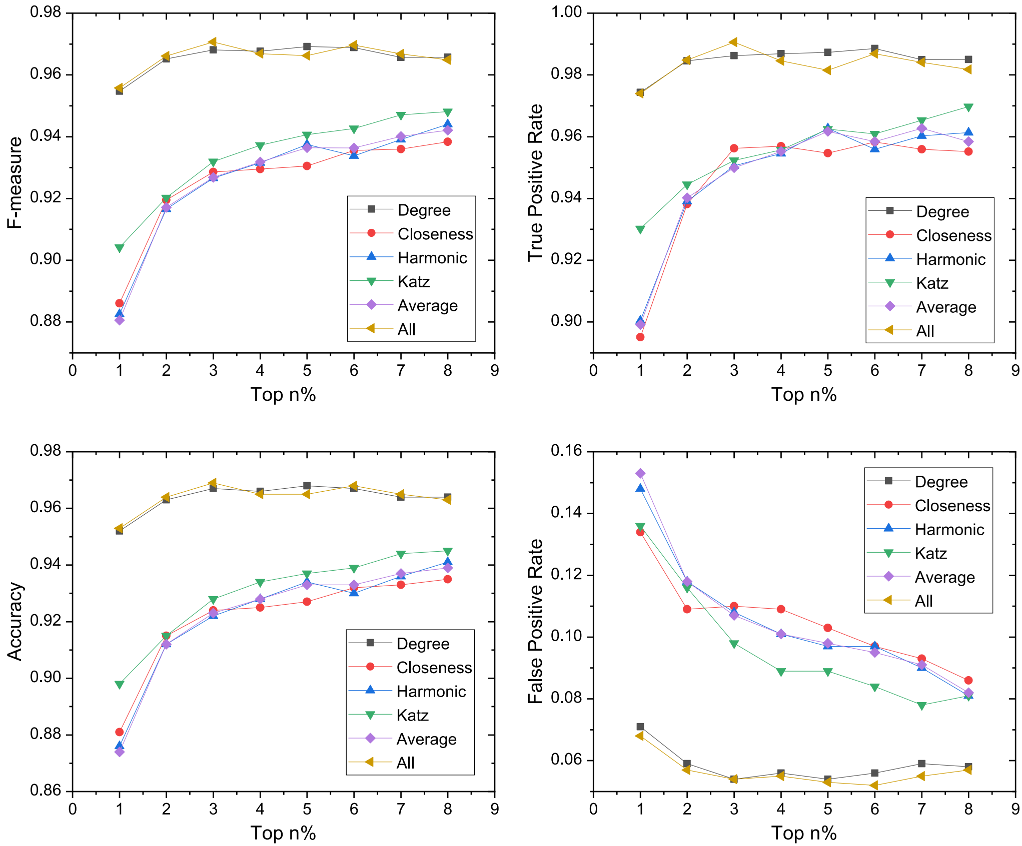

Fig. 8. F-measure, Accuracy, *True-positive Rate* (TPR), and *False-positive Rate* (FPR) of *IntDroid* on detecting Android malware by using different centralities with 1NN.

As for other selected centrality measures, F-measures, Accuracy, and TPR are generally positively correlated to the value of n and FPR is generally negatively correlated to n. In addition, *IntDroid* can always maintain better effectiveness whatever n is when we select degree centrality or all centrality to extract central nodes within call graphs.

5.3 Comparison with Prior Work

In this section, we compare *IntDroid* with 30 anti-virus scanners in VirusTotal [7] and three state-of-the-art Android malware detection approaches: one permission-based approach (i.e., *PerDroid*[7] [53]), one traditional graph-based method (i.e., *MaMaDroid* [43]), and one social-network-analysis–based system (i.e., *MalScan* [54]). To show more clearly in figure, we only present the best result[8] of *IntDroid* from our all 48 experimental results in Table 5.

5.3.1 With Anti-Virus Scanners. As discussed before, our dataset derives from AndroZoo [11] and each of APK file has been detected by different anti-virus products in VirusTotal [7]. We leverage the detection reports to generate our dataset. An APK file is considered benign only if all

[7]For more convenient discussion, we call the system [53] as *PerDroid*, because it is a permission-based method.

[8]*IntDroid* is able to maintain the best performance when we select nodes with top 3% all centralities as central nodes and apply 1NN to train a model for malware detection.

Table 6. The Names of 30 Selected Anti-virus Scanners in VirusTotal

ID	Scanners	ID	Scanners	ID	Scanners
AV1	SymantecMobileInsight	AV11	Babable	AV21	Trustlook
AV2	ESET-NOD32	AV12	McAfee	AV22	MicroWorld-eScan
AV3	Avira	AV13	WhiteArmor	AV23	Ad-Aware
AV4	F-Secure	AV14	CAT-QuickHeal	AV24	Emsisoft
AV5	AVware	AV15	Cyren	AV25	VIPRE
AV6	Fortinet	AV16	F-Prot	AV26	DrWeb
AV7	Sophos	AV17	AVG	AV27	Comodo
AV8	AhnLab-V3	AV18	Cynet	AV28	MAX
AV9	NANO-Antivirus	AV19	GData	AV29	Arcabit
AV10	Ikarus	AV20	BitDefender	AV30	Baidu-International

Table 7. Detection Rates of 30 Anti-virus Scanners and *IntDroid* on Detecting 4,265 Malicious Apps in Our Dataset

Scanners	AV1	AV2	AV3	AV4	AV5	AV6	AV7	AV8	AV9	AV10	
TPR	0.957	0.916	0.911	0.860	0.858	0.846	0.836	0.825	0.813	0.780	
Scanners	AV11	AV12	AV13	AV14	AV15	AV16	AV17	AV18	AV19	AV20	
TPR	0.765	0.745	0.718	0.702	0.687	0.672	0.672	0.600	0.597	0.593	
Scanners	AV21	AV22	AV23	AV24	AV25	AV26	AV27	AV28	AV29	AV30	IntDroid
TPR	0.576	0.561	0.552	0.550	0.550	0.517	0.496	0.484	0.469	0.454	**0.991**

its reports classify it as non-malicious. In other words, benign apps in our dataset are all within the range that anti-virus scanners can detect. Therefore, in this subsection, we focus on the detection rate of 4,265 malware samples in our dataset. Specifically, we upload these 4,265 malware samples to VirusTotal to generate the detection reports. After obtaining all reports, we show the top 30 TPRs of all anti-virus scanners in Table 7 and the names of these 30 scanners are presented in Table 6.

From the results in Table 7, we can see that the TPRs of these anti-virus scanners vary considerably. For instance, *SymantecMobileInsight* can detect over 95% malware in our dataset while *Baidu-International* is able to detect only 45% of the 4,265 malicious samples. As for *IntDroid*, the TPR is encouraging, since it can discover over 99% malware in our dataset, which is the highest among all the scanners in VirusTotal. Furthermore, we also investigate the overlap of malware detected by *IntDroid* and these 30 anti-virus scanners. We find that although *IntDroid* can detect more malware than these 30 anti-virus scanners, seven false negatives of *IntDroid* (i.e., misclassified malware by *IntDroid*) are correctly detected by certain scanners. We then manually analyze these malware and the result shows that all of them are grayware. A grayware is an unwanted app that is not necessarily malicious but can cause performance issues, as well as security risks, when left unaddressed.[9] In other words, *IntDroid* can detect malware misclassified by 30 scanners and these 30 scanners can also detect malware misclassified by *IntDroid*. Therefore, we can combine *IntDroid* with other anti-virus scanners in VirusTotal to achieve a more complete malware detection.

5.3.2 With PerDroid. Wang et al. [53] proposed an approach for Android malware detection based on risky permissions, which are security-aware features that provide a mechanism of access control to core facilities of the mobile system. They apply three feature ranking methods (i.e.,

[9] https://www.logixconsulting.com/2019/12/24/what-is-grayware-2.

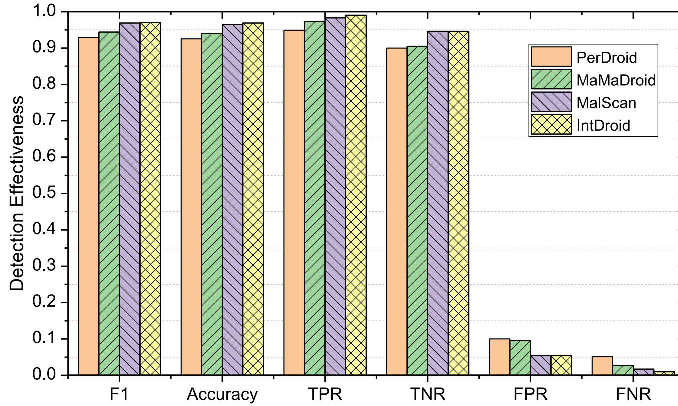

Fig. 9. F-measure, Accuracy, *True-positive Rate* (TPR), *True-negative Rate* (TNR), *False-positive Rate* (FPR), and *False-negative Rate* (FNR) of *IntDroid*, *PerDroid*, *MaMaDroid*, and *MalScan* on detecting Android malware.

mutual information, correlation coefficient, and T-test) to rank Android individual permissions with respect to their risk. Then the top risky permissions are used as features for malware detection. We leverage the risky rank of permissions reported in their paper and use the top 88 risky permissions presented in their public website [1]. Specifically, we employ Androguard [19] to construct the features and feed them to three machine learning classifiers, namely, 1NN, 3NN, and RF. Through the experimental results on our 8,253 samples, we find that 3NN is able to maintain the best effectiveness of these three classifiers.

Figure 9 presents the comparative results of *IntDroid* and *PerDroid* [53], such results indicate that our proposed approach is more effective than permission-based method. For instance, the F-measure and Accuracy are 92.9% and 92.5% while *IntDroid* can achieve 97.1% and 96.9%, respectively. This observation is mainly due to the lack of program semantics of permission-based method.

5.3.3 With MaMaDroid. We also compare *IntDroid* with a state-of-the-art graph-based approach, namely, *MaMaDroid* [43], which leverages the sequences of abstracted function calls obtained from a call graph and uses it to extract features to conduct classification. These abstracted sequences are used to build a Markov chain model to represent the transmission probabilities between functions. Specifically, we use *MaMaDroid*'s open-source code in their website [2] for its abstraction, modeling, and feature extraction. In an effort to complete the classification, we implement three classifiers (i.e., 1NN, 3NN, and RF) as described in their paper, which are not provided in their source code. Comparative results of *IntDroid* and *MaMaDroid* in terms of F-measure, Accuracy, TPR, TNR, FPR, and FNR are given in Figure 9.

From results in Figure 9, we can see that *MaMaDroid* outperforms permission-based method, this happens because of the reserve of program semantics in *MaMaDroid* (i.e., it distills the program semantics into a graph representation) while permission-based method ignores them. In addition, the comparative results also demonstrate that *IntDroid* is capable of detecting more malware and achieving better effectiveness than *MaMaDroid*. The F-measure and TPR are able to maintain 97.1% and 99.1% with *IntDroid* while they are 94.4% and 97.3% with *MaMaDroid*. This is mainly due to the fact that the abstraction of API calls can cause some false positives. For instance, two completely different API calls *TelephonyManager.getDeviceId()* and *SmsManager.sendTextMessage()* can be abstracted into the same package, which is *android.telephony*.

5.3.4 With MalScan. To achieve market-wide mobile malware scanning, *MalScan* [54] regards the function call graph of an app as a complex social network and apply centrality analysis on sensitive API calls to extract semantic features of the graph. Then these centrality values are used to train a model for Android malware detection. As described in *MalScan* [54], we can see that the average detection effectiveness of *MalScan* is able to maintain the best when concatenate centrality[10] is used to train a 1NN model for detecting malware. Therefore, we extract the concatenate centrality of sensitive API calls within a call graph to train a 1NN model for completing the comparative experiment.

From the results presented in Figure 9, we see that *MalScan* can achieve better effectiveness than permission-based method because of the consideration of graph semantics of an app. Moreover, *IntDroid* is capable of maintaining slightly better effectiveness than *MalScan*. For instance, the TPR is 98.3% with *MalScan* while it is 99.1% when we use *IntDroid* for classification. This happens because only using centrality of sensitive API calls to conduct malware detection may cause some false positives and false negatives when benign apps show some similar behaviors as malware by invoking sensitive API calls. For example, some social apps (e.g., Tiktok) require to access users' location for presenting location-specific news or videos and read users' address book for recommending new friends. In such case, the centrality of sensitive API call *LocationManager.getLastLocation()* may be almost the same as some malware. However, when we use intimacy between sensitive API calls and central nodes to detect malware, the central nodes of these benign apps and malware are different, resulting in different intimacies between the same sensitive API call and central nodes in different apps. In other words, although the centrality values of sensitive API calls are similar, the intimacy between the API call and central nodes can be different, since the central nodes are different in benign apps and malware. By this, we can detect more malware with lower false negatives.

5.4 Runtime Overhead

In this section, we perform a comprehensive evaluation on runtime overhead of *IntDroid* by using our 8,253 samples (i.e., 3,988 benign samples and 4,265 malicious samples). The average number of nodes and edges of these 8,253 samples are 4,779 and 10,159, respectively. Given a new app, *IntDroid* consists of four main steps to complete the classification: (1) Function Call Graph Extraction, (2) Centrality Analysis, (3) Intimacy Analysis, and (4) Classification.

5.4.1 Function Call Graph Extraction. The first step of *IntDroid* is to extract the function call graph for a given APK file. Figure 10 presents the runtime overheads of call graph extraction on our dataset, for more than 99% APK files we can obtain call graphs in 2 s. On average, it is required 0.37 s to construct a call graph for a given APK file.

5.4.2 Centrality Analysis. The second step of *IntDroid* is to conduct centrality analysis to unearth the central nodes within a function call graph. As shown in Table 8 and Figure 11, the average runtime overheads vary according to different centrality measures. It is obvious that the running time of average centrality and all centrality is around the sum of four individual centralities, which is reasonable, because these two centralities are the combination of degree centrality, closeness centrality, harmonic centrality, and Katz centrality.

5.4.3 Intimacy Analysis. The third step of *IntDroid* is to perform intimacy analysis for extracting the feature vector. Table 9 and Figure 11 present the average runtime overheads of *IntDroid* in this step, such results indicate that the average running time of intimacy analysis is generally positive

[10] More detailed descriptions are in *MalScan* [54].

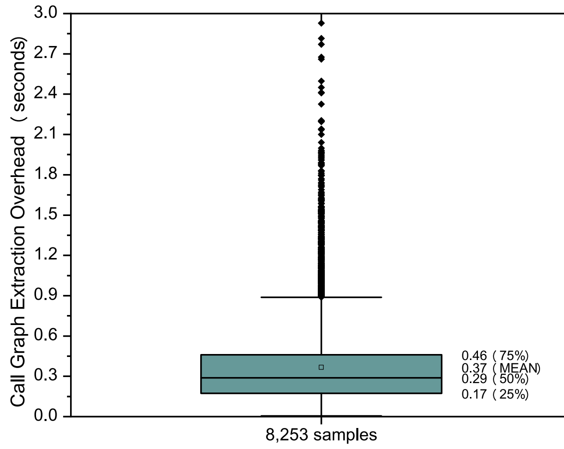

Fig. 10. Runtime overheads of call graph extraction (seconds).

Table 8. Average Runtime Overheads of Centrality Analysis (seconds)

Top *n*%	1	2	3	4	5	6	7	8
Degree	0.41	0.40	0.40	0.40	0.40	0.40	0.40	0.40
Closeness	0.67	0.66	0.65	0.65	0.66	0.65	0.65	0.65
Harmonic	0.81	0.80	0.80	0.79	0.79	0.78	0.79	0.79
Katz	0.80	0.78	0.78	0.77	0.76	0.76	0.76	0.76
Average	2.52	2.53	2.50	2.51	2.48	2.47	2.47	2.48
All	2.43	2.40	2.40	2.40	2.39	2.39	2.42	2.40

Table 9. Average Runtime Overheads of Intimacy Analysis (seconds)

Top *n*%	1	2	3	4	5	6	7	8
Degree	4.61	11.50	17.24	21.46	25.12	29.42	32.53	36.42
Closeness	1.17	2.39	3.93	6.04	8.28	10.45	12.47	14.17
Harmonic	1.14	2.40	3.92	5.68	7.83	10.25	12.20	13.86
Katz	1.16	2.40	4.02	5.53	7.10	8.61	10.03	11.36
Average	1.15	2.44	3.91	5.67	7.90	10.12	12.17	14.38
All	5.32	13.52	20.98	27.38	33.72	39.58	45.06	50.40

related to the top *n*%. This observation is mainly due to the fact that the larger *n*, the more central nodes, resulting in more runtime overheads to extract the average intimacy between sensitive API calls and central nodes.

5.4.4 Classification. The last step of *IntDroid* is to train classifiers by using feature vectors obtained from intimacy analysis step and perform classification on a testing dataset. As shown in Table 10 and Figure 11, it consumes less than 0.02 s to complete the classification. The runtime overhead of classification is the least expensive of four steps.

Table 11 and Figure 11 present the total average runtime overheads of *IntDroid* to analyze a given APK file. When we select nodes with top 1% degree centrality as central nodes and conduct

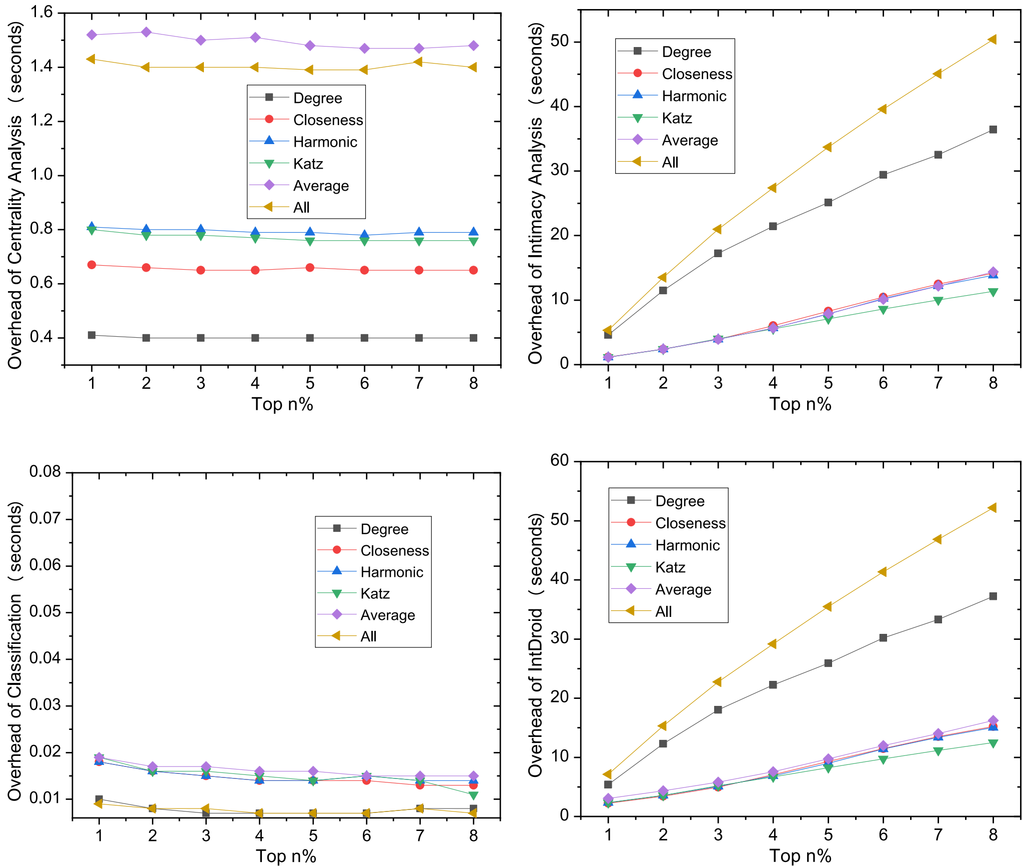

Fig. 11. Runtime overhead of *IntDroid* on different phases.

Table 10. Average Runtime Overheads of Classification with 1NN (seconds)

Top *n*%	1	2	3	4	5	6	7	8
Degree	0.010	0.008	0.007	0.007	0.007	0.007	0.008	0.008
Closeness	0.018	0.016	0.015	0.014	0.014	0.014	0.013	0.013
Harmonic	0.018	0.016	0.015	0.014	0.014	0.015	0.014	0.014
Katz	0.019	0.016	0.016	0.015	0.014	0.015	0.014	0.011
Average	0.019	0.017	0.017	0.016	0.016	0.015	0.015	0.015
All	0.009	0.008	0.008	0.007	0.007	0.007	0.008	0.007

intimacy analysis to produce a feature vector for classification, the total runtime overhead is only 5.4 s on average, while the TPR is able to maintain 97.4%.

On the one hand, *PerDroid* [53] is a permission-based method and its effectiveness on malware detection is lower than both *MaMaDroid* [43] and *MalScan* [54]. On the other hand, *IntDroid* is the combination of traditional graph-analysis–based method with social-network-analysis–based method. Therefore, we mainly focus on the runtime overhead of *IntDroid* compared to traditional graph-analysis–based method (i.e., *MaMaDroid*) and social-network-analysis–based approach (i.e., *MalScan*). As for *MaMaDroid*, Table 12 shows that it needs to take about 33.83 s to finish a

Table 11. Total Average Runtime Overheads of *IntDroid* to Complete Analyzing a Given APK File (seconds)

Top n%	1	2	3	4	5	6	7	8
Degree	5.40	12.28	18.02	22.24	25.90	30.20	33.31	37.20
Closeness	2.23	3.43	4.96	7.07	9.32	11.48	13.50	15.20
Harmonic	2.34	3.58	5.10	6.85	9.00	11.41	13.37	15.03
Katz	2.35	3.56	5.18	6.68	8.24	9.75	11.17	12.50
Average	4.06	5.36	6.80	8.56	10.76	12.97	15.02	17.24
All	8.13	16.30	23.76	30.16	36.49	42.35	47.86	53.18

Table 12. Comparative Runtime Overhead of *MaMaDroid*, *MalScan*, and *IntDroid*

Tools	*MaMaDroid*	*MalScan*	*IntDroid*	
			Degree_1	All_3
RunTime Overhead (s)	33.83	2.75	5.40	23.76

classification on average. Although the runtime overhead of *MaMaDroid* can be almost the same as *IntDroid*. However, it requires more than 7 times as much memory as *IntDroid* to perform classification. *MaMaDroid* requires 63.7 GB of memory to complete the classification on our 8,253 samples, while *IntDroid* only requires 8.5 GB. This happens because the dimension of feature vector of *MaMaDroid* is 115,600 while is 21,986 for our proposed method. Moreover, because of the *intimacy analysis* of *IntDroid*, its detection efficiency is lower than *MalScan*. For instance, when we treat nodes with top 1% degree centralities as central nodes to train a classifier by using 1NN algorithm, it takes about 5.4 s to analyze a given app for *IntDroid* while *MalScan* only consumes 2.7 s to complete the whole analysis when we extract the concatenate centrality as the feature to train a 1NN classifier. However, *IntDroid* and *MalScan* can be complementary because of the higher effectiveness of *IntDroid*. In other words, *MalScan* can be used as the first line of defense to filter most of malware, then *IntDroid* can be applied as the second line of defense to excavate more malware. In this way, we can save longer time and resources. For example, suppose that given 10,000 new apps, we can use *MalScan* to detect these apps first. Then benign apps classified by *MalScan* can be fed into *IntDroid* to be deeply analyzed for discovering more malicious apps. By this, we can save longer times compared to analyzing these 10,000 apps only using *IntDroid*. In practice, both *IntDroid* and *MalScan* can not achieve 100% of detection accuracy. Therefore, after collecting all detected malware samples, we can upload them to VirusTotal for generating more detailed reports to assist analysts to filter benign apps from these malware.

5.5 Detection of Zero-Day Malware

In this subsection, we would like to check the ability of *IntDroid* on discovering real-world malware. To achieve this goal, we treat nodes with top 3% all centralities as central nodes and leverage our 8,253 samples to train a classifier by using 1NN algorithm. Next, we crawl 5,000 apps from GooglePlay market and feed them to the trained 1NN classifier. Among these apps, *IntDroid* reports 32 of them as being malicious. To validate whether these 32 apps are indeed malware or not, we upload them to VirusTotal [7] to analyze each of them. Among these 32 samples, 24 of them are reported as malicious by at least two anti-virus scanners. To further study the behaviors of these 24 samples, we upload them to a state-of-the-art sandbox [8], which combines static and dynamic

analysis for reporting detailed risky behaviors. Through the results, we observe that all of them collect user's private information (e.g., IMSI, IMEI, phone number, and contacts) and then send them to the network or write them into a file. Additionally, seven of them execute shellcode to complete more risky activities. After analyzing these samples, we then conduct an investigation (i.e., obtaining more detailed information from GooglePlay official website) about them, results show that one of which has been downloaded and installed by more than ten million users. This app has also been flagged as malware by six anti-virus scanners in VirusTotal [7], one of which is *Symantec Mobile Insight*. In reality, three apps are flagged as malware by only one scanner in VirusTotal. Similarly, we also upload them to the sandbox [8] and the behavioral reports show that they are not necessarily malicious and are identified as one type of grayware (i.e., adware).

Of the remaining five apps, we manually inspect them. Our manual checks show that one of these five apps is actually malware as it contains highly suspicious behaviors. The app is a music downloader, which contains six dangerous permissions and reads battery and memory information of the device. Moreover, the app collects many sensitive data (i.e., Android ID, serial number, IMSI, IMEI, phone number, contacts, emails, and location) and even can execute shellcode. In conclusion, *IntDroid* is able to discover 28 zero-day malware[11] among 5,000 GooglePlay apps, one of them has been downloaded and installed by more than 10 million users, and one of them is not reported as malware by existing tools [7].

6 EXTENSIVE COMPARISON WITH MALSCAN

Because the most similar system of *IntDroid* is *MalScan*, therefore, to validate the higher effectiveness of *IntDroid*, we present an extensive comparison with *MalScan* in this section. From the experimental results presented in Section 5, we are able to maintain the best effectiveness when we select nodes with top 3% all centralities as central nodes and apply 1NN to train a model for malware detection. We choose these three parameters[12] to commence our comparison with *MalScan*.

6.1 Malware Detection in the Twilight Zone

As aforementioned, a grayware sample is an unwanted app that is not necessarily malicious but can cause performance issues, as well as security risks, when left unaddressed. According to a recent report,[13] grayware can do plenty of damage even if they are not actively malicious. Since the malicious behaviors of grayware are not obvious, they are more difficult to be distinguished from benign apps. In Euphony [38], authors find that apps flagged by at least five anti-virus scanners in VirusTotal exhibit more obvious malicious behaviors. In other words, apps that are detected by one to five scanners are highly likely in the twilight zone between malicious and benign functionality.

In this part, we pay attention to detect malware in the twilight zone (e.g., grayware) with *MalScan* and *IntDroid*. Specifically, we randomly download 3,000 malware samples that are reported as malicious by one to five scanners in VirusTotal. As for benign apps, we use 3,988 benign samples in Section 5 as our dataset. After collecting 3,988 benign apps and 3,000 malicious samples, we begin to examine the capability of *MalScan* and *IntDroid* on detecting malware in the twilight zone.

Table 13 presents the comparative results of *MalScan* and *IntDroid*. The TPR of *MalScan* is 91.2%, which means that *MalScan* can only detect 91.2% of malware that are in the twilight zone. However, as for *IntDroid*, the TPR is able to maintain 95.1%, which is greater than 91.2%. Such result indicates that *IntDroid* performs better than *MalScan* on detecting malware in the twilight zone.

[11] More detailed behaviors and information are available in the website: https://github.com/IntDroid/IntDroid.
[12] Classification model: 1NN, Centrality measure: all centrality, and $N = 3$.
[13] https://solutionsreview.com/endpoint-security/grayware-can-defend/.

Table 13. Detection Effectiveness of *MalScan* and *IntDroid* on Detecting Malware in the Twilight Zone

Systems	Accuracy	Precision	Recall	F1	TPR	TNR	FPR	FNR
MalScan	0.930	0.925	0.912	0.918	0.912	0.945	0.055	0.088
IntDroid	0.961	0.957	0.951	0.954	0.951	0.968	0.032	0.049

Although *IntDroid* can achieve 96.1% of accuracy when detecting our collected dataset, it still can not detect certain malware in the twilight zone. We then manually inspect these misclassified malware and observe that most of them are adware. The most common reason for adware is to collect information about users for the purpose of making advertising dollars. In practice, some benign apps may use ad libraries to access users' private information and even monitor users' behavior to push suitable ads for profit. Behaviors of these ad libraries and adware are similar, resulting a wrong classification.

6.2 Malware Detection in Different Families

There is another important aspect (i.e., the balance of malware families in the experimental dataset [12, 48]) that should be considered when we test the detection effectiveness of a malware detection method. Suppose that the number of samples of several malware families is much larger than of other families, then the detection ability of the trained model will mainly depend on these families. In other words, a malware detection method should all maintain high effectiveness for different families. In this subsection, we conduct a comparative experiment to evaluate the effectiveness of *MalScan* and *IntDroid* on detecting malware in different families. Specifically, we select the 20 largest malware families in AndroZoo [11] as our test objects. Some malicious APK files in AndroZoo [11] are labeled into corresponding families by using method in Reference [33]. We randomly download 1,000 samples for each family, and the total number of downloaded samples is 20,000. The family names and the average size of samples for each family can be found in Table 14.

We leverage our 3,988 benign samples used in Section 5 and these 20,000 malware samples to commence our study. In reality, we totally conduct 20 experiments for 20 families. In other words, the dataset in each experiment including 3,988 benign apps and 1,000 malicious apps for each family. We repeat this 20 times for 20 families. All the researches are performed by using 10-fold cross validations. The detection effectiveness of *MalScan* and *IntDroid* for each family is illustrated in Figure 12.

From the results in Figure 12, we can see that *IntDroid* is capable of maintaining better effectiveness than *MalScan* on detecting malware in all these 20 families. For example, when detecting malware in admogo family, *MalScan* can only detect 95.5% of malware while *IntDroid* can achieve 98.8% of TPR. Moreover, *IntDroid* is able to maintain above 94% of detection ratio for all 20 malware families. In particular, there are seven families with a TPR exceeding 98%, and one family with a TPR greater than 99%. The average detection ratio of *IntDroid* is 96.9% when testing on these 23,988 samples, such high effectiveness suggests that *IntDroid* is suitable for the detection of malware in different families.

6.3 Malware Detection in Unknown Families

Along with the constant evolution of Android, many new malicious samples are created by attackers for evading existing malware detectors [52]. Some of these samples may not belong to any existing malware families. When the app is from a new malware family, malware detectors may produce a false negative because of a lack of discriminative features to label it as a malware. Therefore, it is important to assess the ability of a malware detection method on distinguishing

Table 14. The Average Size (MB) of Top 20 Families in Our Dataset

Family	Average Size (MB)	#Samples
admogo	2.61	1,000
adwo	2.64	1,000
airpush	2.48	1,000
appsgeyser	0.53	1,000
artemis	3.41	1,000
deng	4.37	1,000
domob	1.95	1,000
droidkungfu	1.94	1,000
feiwo	2.21	1,000
gingermaster	4.11	1,000
ginmaster	2.36	1,000
kuguo	3.38	1,000
leadbolt	2.15	1,000
plankton	2.15	1,000
smspay	5.84	1,000
startapp	2.69	1,000
waps	2.47	1,000
wapsx	2.41	1,000
wooboo	1.30	1,000
youmi	2.11	1,000
Total	2.66	20,000

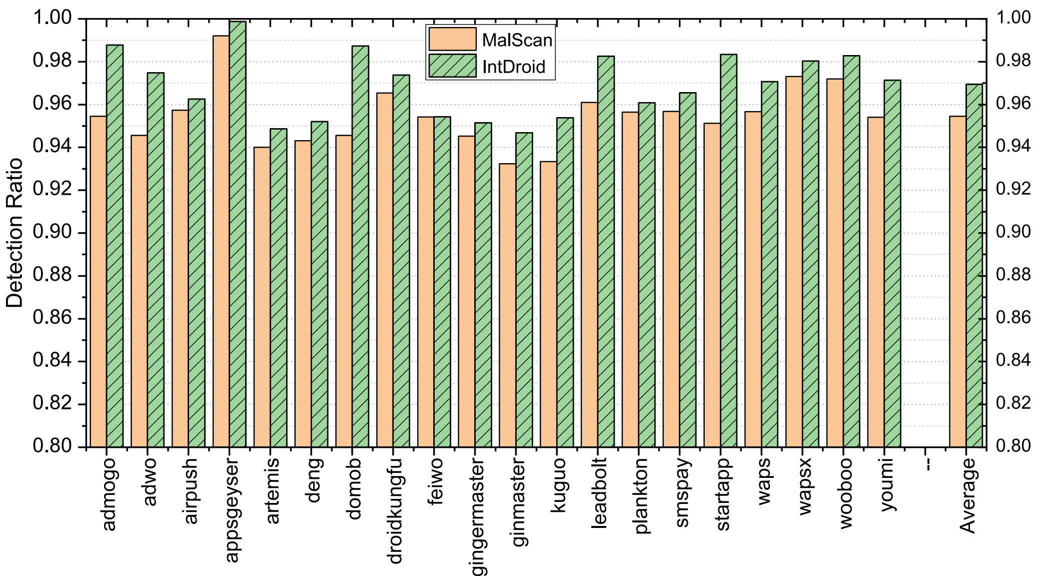

Fig. 12. Detection effectiveness of *MalScan* and *IntDroid* in 20 different malware families.

Table 15. Detection Effectiveness of *MalScan* and *IntDroid* on Detecting Malware in Unknown Families

families	0 Samples Available		10 Samples Available		100 Samples Available	
	MalScan	*IntDroid*	*MalScan*	*IntDroid*	*MalScan*	*IntDroid*
admogo	0.982	0.994	0.984	0.994	0.987	0.999
adwo	0.975	0.988	0.979	0.989	0.981	0.99
airpush	0.912	0.929	0.921	0.941	0.921	0.941
appsgeyser	1	1	1	1	1	1
artemis	0.855	0.877	0.867	0.917	0.896	0.947
deng	0.857	0.884	0.865	0.888	0.901	0.919
domob	0.961	0.977	0.965	0.978	0.972	0.984
droidkungfu	0.959	0.979	0.963	0.979	0.963	0.979
feiwo	0.876	0.89	0.876	0.89	0.909	0.938
gingermaster	0.949	0.957	0.949	0.957	0.949	0.957
ginmaster	0.889	0.902	0.89	0.904	0.904	0.909
kuguo	0.945	0.954	0.949	0.955	0.951	0.961
leadbolt	0.921	0.934	0.925	0.944	0.936	0.951
plankton	0.88	0.88	0.88	0.88	0.921	0.946
smspay	0.914	0.925	0.916	0.93	0.924	0.96
startapp	0.801	0.814	0.831	0.849	0.869	0.926
waps	0.977	0.983	0.977	0.983	0.977	0.984
wapsx	0.961	0.981	0.961	0.981	0.961	0.983
wooboo	0.931	0.957	0.939	0.975	0.958	0.983
youmi	0.966	0.973	0.966	0.973	0.966	0.973
Average	0.926	0.939	0.930	0.945	0.942	0.962

new malware families. In this subsection, we conduct a comparative experiment to examine the effectiveness of *MalScan* and *IntDroid* on detecting malware in unknown families.

Our dataset in this subsection consists of 3,988 benign samples in Section 5 and 20,000 malicious samples in Section 6.2. We totally conduct three experiments where we limit the number of training samples of a particular family. In our first experiment, we do not provide any samples of the family in our training set. Suppose that *admogo* family is a new family, then the training set will be made up of 3,988 benign samples and 19,000 malicious samples, including 0 *admogo* family samples. To simulate the starting spread of a new family, we commence our second experiment, we first randomly select 10 samples of the family, then these selected samples are put into the training set. The number of malicious training samples in this experiment is 19,010, which includes 10 new family samples. As a new family of malware proliferates, the number of available samples will increase. In our final experiment, we assume that the number of new family samples obtained is 100 and put these 100 samples into our training set. We conduct these three experiments for each family and the detection results are presented in Table 15.

As shown in Table 15, similar to the results in Figure 12, *IntDroid* performs better than *MalScan* on detecting malware in all new families. When the number of available samples for a new family is 0, 10, and 100, the average TPR of *MalScan* is 92.6%, 93.0%, and 94.2%, respectively. However, they are all smaller than TPRs achieved by *IntDroid*. Furthermore, without any samples of a new family for training, the TPR of *IntDroid* is still able to maintain above 80% for 20 different families. Moreover, 11 families can achieve above 95% of TPR and even one family (i.e., *appsgeyser*) can be detected perfectly (i.e., 100% accuracy). Such results suggest that *IntDroid* is capable of detecting malware from a new family with good effectiveness. When the number of available samples of a

new family increases, the TPR of *IntDroid* can achieve better effectiveness. For instance, the TPR of *IntDroid* on detecting *startapp* family malware is 81.4% if no *startapp* samples are available for training, while are 84.9% and 92.6% when the available training samples increase to 10 and 100, respectively. Such results are reasonable, because members of certain families often show similar behaviors and are just repackaged malware with some slight modifications. The more the number of available samples, the more variations of a family can be detected.

7 DISCUSSIONS AND LIMITATIONS

7.1 Threats to Validity

Threats to External Validity. Using a limited number of apps in Section 5 poses an external validity threat. These limited apps may not be representative of the entire market. We conduct other detailed comparative experiments in Section 6 to mitigate the threat by using 20,000 malicious apps from 20 families.

Threats to Internal Validity. The distribution of intimacy values may cause some inaccuracies when we conclude the level of the differences between benign apps and malicious apps. We mitigate the threat by adopting a non-parametric test (i.e., Mann-Whitney) to study the difference of these intimacy values. In addition, inaccuracies in logging the runtime overheads are inevitable due to the different machine running states (e.g., different CPU usages). The threat is mitigated by reporting the average runtime overhead after collecting all runtime overheads three times. Moreover, selecting central nodes according to the simply sorted percentage of node centrality may cause some inaccuracies. To mitigate the threat, we select eight thresholds (i.e., 1% to 8%) for performing more complete experiments to find out the ideal threshold for malware detection. Finally, there may be some false positives in detecting zero-day malware from Google Play market. We upload these samples to a sandbox that combines static with dynamic analysis for generating detailed risky reports to mitigate the threat.

7.2 False Positives and False Negatives

False Positives. After deeply analyzing the false positives caused by *IntDroid*, we find that the most common reason is the use of ad libraries in benign apps. These ad libraries need to access private information and even monitor users' behavior to push suitable ads for profit. Behaviors of these ad libraries may be misclassified as malware, resulting in false positives.

False Negatives. As for false negatives of *IntDroid*, we observe that most of them are in the twilight zone, that is, grayware. In other words, malicious codes of these samples do not exhibit obvious malicious behaviors or are deeply disguised. Adware is one type of grayware and is unwanted software designed to throw advertisements up on your screen. Since most of them are not performing any clearly malicious activities, *IntDroid* can not flag them as malware correctly.

7.3 Discussions

In our work, we totally select four individual centrality measures and construct another two centrality measures to excavate central nodes for Android malware detection. Through the experimental results in Section 5, we see that degree centrality can achieve the best performance among four individual centrality measures (i.e., degree centrality, katz centrality, closeness centrality, and harmonic centrality). Moreover, when we use the combined central nodes (i.e., all centrality) to detect malware, the performance is almost the same as using degree centrality.

We conduct a simple study to examine the cosine distance between the feature vectors obtained by using degree centrality and all centrality. We choose n=1 (i.e., nodes with top 1% centrality will be selected as central nodes) as our parameter. For feature vectors of 3,988 benign samples between

degree centrality and all centrality, the similarity of 3,921 samples (3,921/3,988=98.32%) is greater than 90%, 3,185 samples (3185/3988=79.86%) have a similarity greater than 99%, and the similarity of 548 samples are 100%. In addition, as for our 4,265 malicious samples, the similarity of 4,249 samples (4,249/4,265=99.62%) is greater than 90%, 3,818 samples (3,818/4,265=89.52%) have a similarity greater than 99%, and the similarity of 394 samples are 100%. Such results indicate that degree centrality may be the best candidate to extract central nodes for malware detection. In our future work, we will test the capability of more different centrality measures on detecting Android malware.

In addition, since most Android malware detection systems are closed source, we only compare *IntDroid* with three open-source systems [43, 53, 54]. We will conduct a detailed comparative analysis on more systems in our future work.

7.4 Limitations

Similar to any empirical approach, *IntDroid* suffers from several limitations, which are listed below.

(1) Sensitive API calls: Our method depends on intimacy analysis between sensitive API calls and central nodes within a call graph. We use the newest version of sensitive API calls mapped by PScout [13]. However, it may be partially outdated. Some incorrect and missing API calls may cause some false positives and false negatives on malware detection. Sensitive API calls need to be updated by using PScout [13] on the latest Android version.

(2) Encryption: Using obfuscation techniques to protect Android apps is very popular. Considering that *IntDroid* analyzes the function call graph to extract the features, it is resilient to several typical local obfuscation techniques [32], such as renaming of the user-defined functions and packages. However, it is vulnerable to certain obfuscation techniques, such as encryption (e.g., *APK Protect* [5]). These encryption packers can protect apps by using encryption techniques to hide the actual Dex code. To address this limitation, we can use some unpacker systems [55, 59] to recover the actual Dex files, then static analysis can be applied to extract the call graph.

(3) Call Graph Extraction: To maintain high efficiency of *IntDroid* on malware detection, we conduct simple static analysis to extract a succinct function call graph by using Androguard [19]. In reality, many apps use reflection technique [47] to call sensitive methods, in which case, we may miss the call relationships between these methods. To be resilient to reflection, we can use an open-source tool, *DroidRA* [37], to conduct reflection analysis on our dataset to identify methods that use reflection for each app. Then the missing edges can be added into the call graph, where caller nodes are methods that use reflection and callee nodes are reflected methods. Moreover, function call graph extracted by Androguard [19] is a context- and flow-insensitive call graph. We ignore these information for achieving high efficiency to conduct malware detection. It is dilemmatic for us to maintain high scalability if we perform expensive program analysis for considering the context and flow information of a call graph. However, our experimental results show that the succinct call graph is enough for us to perform effective malware detection.

8 RELATED WORK

There has been proposed many approaches on Android malware detection that can be classified into two main categories: syntax-based and semantics-based.

8.1 Syntax-based Android Malware Detection

Syntax-based methods [9, 12, 21, 31, 36, 46, 50, 53, 60] ignore the semantics of app code to achieve efficient Android malware detection. For example, Wang et al. [53] focus on the permissions requested by apps to detect Android malware. It scans the manifest file to collect the list of all permissions, and then apply several feature ranking methods to rank them with respect to the risk. After obtaining the ranking of all analyzed permissions, permissions with top risks will be

considered as risky permissions and are used as features to detect malware. These risky permissions can provide a mechanism of access control to core facilities of the mobile system, so they can be represented as a type of apps' behavior. Similar to Wang et al. [53], Huang et al. [31] also extract permissions and several easy-to-retrieve features (e.g., the number of files with a ".so" extension filename) from APK files to detect malware. In practice, due to the limited precision, they conclude that their method can be used as a quick filter to identify malware, and then more advanced techniques should be applied to achieve more accurate malware detection. In other words, because of the lack of program semantics, permission-based approaches suffer from low effectiveness on detecting Android malware. To mitigate the issue, *Drebin* [12] uses an extensive static analysis to extract as many features as possible from both manifest and disassembled code, and embeds them in a joint vector space to detect malware. Feature sets in the manifest consists of hardware features, requested permissions, app components, and filtered intents while includes restricted API calls, used permissions, suspicious API calls, and network addresses in app disassembled code. However, it only searches for the presence of particular strings, rather than considers the program semantics. So it can be easily evaded by attacks on syntax features [17].

8.2 Semantics-based Android Malware Detection

To maintain high effectiveness on detecting Android malware, researchers [10, 14, 20, 23–25, 27, 28, 40, 43–44, 45, 49, 56–58] conduct program analysis to extract different types of app semantics. For example, *MassVet* [16] builds a view graph to describe an app with a reasonably complicated UI structure. To ensure the high scalability on graph matching, *MassVet* applies a similarity comparison algorithm that appeared in their former work [15] to the analysis of recovered view graph. It has validated the high efficiency and scalability on mobile malware detection, however, the original purpose of *MassVet* is to detect repackaged malware. It can cause a false negative when the app is a new malware. *DroidSIFT* [58] extracts the weighted contextual API dependency graph to solve the malware deformation problem based on static analysis. *Apposcopy* [24] utilizes static taint analysis to form a new program representation called *Inter-Component Call Graph* and use it to detect malware. However, both *DroidSIFT* [58] and *Apposcopy* [24] suffer from heavy runtime overhead. As reported in their papers, they consume an average of 175.8 s and 275 s to analyze an app, respectively. *SMART* [44] constructs semantic models of Android malware based on *Deterministic Symbolic Automaton*, which can capture common malicious behaviors of malware families. It consists of two main phases to complete malware detection. The first phase is offline model learning, which consumes an average of 72.5 s for clone detection and 167.5 s for clone differencing and DSA generation. The second phase is online malware detection and classification where ML-based malware detection takes 13.4 s on average while it takes 105.9 s on average to classify malware into corresponding families. *MaMaDroid* [43] leverages the sequences of abstracted function calls obtained from a call graph to build a behavioral model and uses it to extract features to conduct classification. One limitation of this method is that it can be easily evaded by the self-defined packages that look similar to Android's, Google's, or Java's packages [17], another is that it requires a considerable amount of memory on classification because of its large features [43].

8.3 Differences from *MalScan*

The most similar work as *IntDroid* is our prior work, *MalScan* [54], which considers the function call graph as a complex social network and apply social-network–based centrality analysis on sensitive API calls to represent the graph semantics for classification. However, only using the centrality of sensitive API calls to conduct malware detection may cause some false positives when benign apps show some similar behaviors as malware by invoking sensitive API calls. In such case,

the centrality of certain sensitive API calls in benign apps may be almost the same as in some malware. However, when we use intimacy (defined in Section 3.2) between sensitive API calls and central nodes to detect malware, the central nodes of these benign apps and malware are different, resulting in different intimacies between the same sensitive API call and central nodes in different apps. In other words, although the centrality values of sensitive API calls are similar, the intimacy between the API call and central nodes in benign apps and malicious apps can be different, since the central nodes are different. By this, we can detect more malware with lower false positives. In reality, because of the computation of intimacy between sensitive API calls and central nodes, the efficiency of *IntDroid* is lower than *MalScan*. Therefore, *MalScan* can be used as the first line of defense to filter most of malware, then *IntDroid* can be applied as the second line of defense to discover more malware. In this way, we can achieve more efficient malware detection and save more resources.

9 CONCLUSION

In this article, we present a novel approach to detect Android malware based on intimacy analysis between sensitive API calls and central nodes within function call graphs. To avoid heavyweight graph matching overheads, we treat a function call graph as a complex social network and conduct centrality analysis to unearth central nodes. We implement an automatic system, *IntDroid*, and the extensive evaluations show that our proposed system is able to maintain high accuracy and scalability on Android malware detection.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments to improve the quality of the article.

REFERENCES

[1] 2014. Permission-based method. Retrieved from http://infosec.bjtu.edu.cn/wangwei/?page_id=85/.
[2] 2017. MaMaDroid. Retrieved from https://bitbucket.org/gianluca_students/mamadroid_code/.
[3] 2018. Cyber attacks on Android devices on the rise. Retrieved from https://www.gdatasoftware.com/blog/2018/11/31255-cyber-attacks-on-android-devices-on-the-rise/.
[4] 2018. Worldwide Smartphone Sales to End Users by Operating System in 2Q18. Retrieved from https://www.gartner.com/en/newsroom/press-releases/2018-08-28-gartner-says-huawei-secured-no-2-worldwide-smartphone-vendor-spot-surpassing-apple-in-second-quarter/.
[5] 2019. APK Protect—Provide Android APK Encryption and Protection. Retrieved from https://sourceforge.net/projects/apkprotect/.
[6] 2019. scikit-learn. Retrieved from https://scikit-learn.org/.
[7] 2019. VirusTotal—Free online virus, malware and URL scanner. Retrieved from https://www.virustotal.com/.
[8] 2020. SanDroid—An automatic Android application analysis system. Retrieved from http://sanddroid.xjtu.edu.cn/.
[9] Yousra Aafer, Wenliang Du, and Heng Yin. 2013. 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)*.
[10] Joey Allen, Matthew Landen, Sanya Chaba, Yang Ji, Simon Pak Ho Chung, and Wenke Lee. 2018. Improving accuracy of android malware detection with lightweight contextual awareness. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC'18)*.
[11] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th Working Conference on Mining Software Repositories (MSR'16)*.
[12] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. 2014. Drebin: Effective and explainable detection of android malware in your pocket. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS'14)*.
[13] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. Pscout: Analyzing the Android permission specification. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'12)*.

[14] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2015. Mining apps for abnormal usage of sensitive data. In *Proceedings of the 37th International Conference on Software Engineering (ICSE'15)*.

[15] Kai Chen, Peng Liu, and Yingjun Zhang. 2014. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*.

[16] Kai Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. 2015. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security'15)*.

[17] Xiao Chen, Chaoran Li, Derui Wang, Sheng Wen, Jun Zhang, Surya Nepal, Yang Xiang, and Kui Ren. 2018. Android HIV: A study of repackaging malware for evading machine-learning detection. *IEEE Trans. Info. Forens. Secur.* (2018).

[18] Nigel Coles. 2001. It's not what you know-It's who you know that counts. Analysing serious crime groups as social networks. *Brit. J. Criminol.* (2001).

[19] Anthony Desnos et al. 2011. Androguard. Retrieved from https://github.com/androguard/androguard.

[20] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2014. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Trans. Comput. Syst.* (2014).

[21] Ming Fan, Xiapu Luo, Jun Liu, Chunyin Nong, Qinghua Zheng, and Ting Liu. 2019. CTDroid: Leveraging a corpus of technical blogs for android malware analysis. *IEEE Trans. Reliabil.* (2019).

[22] Katherine Faust. 1997. Centrality in affiliation networks. *Soc. Netw.* (1997).

[23] Ruitao Feng, Sen Chen, Xiaofei Xie, Lei Ma, Guozhu Meng, Yang Liu, and Shang-Wei Lin. 2019. Mobidroid: A performance-sensitive malware detection system on mobile platform. In *Proceedings of the 24th International Conference on Engineering of Complex Computer Systems (ICECCS'19)*.

[24] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 2014. 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)*.

[25] Yu Feng, Osbert Bastani, Ruben Martins, Isil Dillig, and Saswat Anand. 2016. Automated synthesis of semantic malware signatures using maximum satisfiability. Retrieved from https://arXiv:1608.06254.

[26] Linton C. Freeman. 1978. Centrality in social networks conceptual clarification. *Soc. Netw.* (1978).

[27] Joshua Garcia, Mahmoud Hammad, and Sam Malek. 2018. Lightweight, obfuscation-resilient detection and family identification of Android malware. *ACM Trans. Softw. Eng. Methodol.* (2018).

[28] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. 2013. Structural detection of android malware using embedded call graphs. In *Proceedings of the ACM Workshop on Artificial Intelligence and Security (WAIS'13)*.

[29] Michael C. Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. 2012. Systematic detection of capability leaks in stock android smartphones. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS'12)*.

[30] Roger Guimera, Stefano Mossa, Adrian Turtschi, and L. A. Nunes Amaral. 2005. The worldwide air transportation network: Anomalous centrality, community structure, and cities' global roles. *Proc. Natl. Acad. Sci. U.S.A.* (2005).

[31] Chun-Ying Huang, Yi-Ting Tsai, and Chung-Han Hsu. 2013. Performance evaluation on permission-based detection for android malware. In *Advances in Intelligent Systems and Applications*.

[32] Heqing Huang, Sencun Zhu, Peng Liu, and Dinghao Wu. 2013. A framework for evaluating mobile app repackaging detection algorithms. In *Proceedings of the 2013 International Conference on Trust and Trustworthy Computing (ICTTC'13)*.

[33] Médéric Hurier, Guillermo Suarez-Tangil, Santanu Kumar Dash, Tegawendé F. Bissyandé, Yves Le Traon, Jacques Klein, and Lorenzo Cavallaro. 2017. Euphony: Harmonious unification of cacophonous anti-virus vendor labels for Android malware. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR'17)*.

[34] Hawoong Jeong, Sean P. Mason, A.-L. Barabási, and Zoltan N. Oltvai. 2001. Lethality and centrality in protein networks. *Nature* (2001).

[35] Leo Katz. 1953. A new status index derived from sociometric analysis. *Psychometrika* (1953).

[36] Jin Li, Lichao Sun, Qiben Yan, Zhiqiang Li, Witawas Srisa-an, and Heng Ye. 2018. Significant permission identification for machine-learning-based android malware detection. *IEEE Trans. Industr. Info.* (2018).

[37] Li Li, Tegawendé F. Bissyandé, Damien Octeau, and Jacques Klein. 2016. Droidra: Taming reflection to support whole-program analysis of android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'16)*.

[38] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor Van Der Veen, and Christian Platzer. 2014. Andrubis–1,000,000 apps later: A view on current Android malware behaviors. In *Proceedings of the International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS'14)*.

[39] Xiaoming Liu, Johan Bollen, Michael L. Nelson, and Herbert Van de Sompel. 2005. Co-authorship networks in the digital library research community. *Info. Process. Manage.* (2005).

[40] Aravind Machiry, Nilo Redini, Eric Gustafson, Yanick Fratantonio, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. 2018. Using loops for malware classification resilient to feature-unaware perturbations. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC'18).*

[41] Henry B. Mann and Donald R. Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *Ann. Math. Stat.* (1947).

[42] Massimo Marchiori and Vito Latora. 2000. Harmony in the small-world. *Physica A: Stat. Mech. Appl.* (2000).

[43] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. 2017. MAMADROID: Detecting android malware by building markov chains of behavioral models. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS'17).*

[44] Guozhu Meng, Yinxing Xue, Zhengzi Xu, Yang Liu, Jie Zhang, and Annamalai Narayanan. 2016. Semantic modelling of android malware for effective malware comprehension, detection, and classification. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'16).*

[45] Annamalai Narayanan, Mahinthan Chandramohan, Lihui Chen, and Yang Liu. 2018. A multi-view context-aware approach to Android malware detection and malicious code localization. *Empir. Softw. Eng.* (2018).

[46] Hao Peng, Chris Gates, Bhaskar Sarma, Ninghui Li, Yuan Qi, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. 2012. Using probabilistic generative models for ranking risks of android apps. In *Proceedings of the ACM Conference on Computer and communications security (CCS'12).*

[47] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. 2013. Catch me if you can: Evaluating android anti-malware against transformation attacks. *IEEE Trans. Info. Forensics Secur.* (2013).

[48] Christian Rossow, Christian J. Dietrich, Chris Grier, Christian Kreibich, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten Van Steen. 2012. Prudent practices for designing malware experiments: Status quo and outlook. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'12).*

[49] Andrea Saracino, Daniele Sgandurra, Gianluca Dini, and Fabio Martinelli. 2018. Madam: Effective and efficient behavior-based android malware detection and prevention. *IEEE Trans. Depend. Secure Comput.* (2018).

[50] Bhaskar Pratim Sarma, Ninghui Li, Chris Gates, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. 2012. Android permissions: A perspective combining risks and benefits. In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies (ACMT'12).*

[51] Samuel Sanford Shapiro and Martin B. Wilk. 1965. An analysis of variance test for normality (complete samples). *Biometrika* (1965).

[52] Guillermo Suarez-Tangil and Gianluca Stringhini. 2018. Eight years of rider measurement in the android malware ecosystem: Evolution and lessons learned. Retrieved from https://arXiv:1801.08115.

[53] Wei Wang, Xing Wang, Dawei Feng, Jiqiang Liu, Zhen Han, and Xiangliang Zhang. 2014. Exploring permission-induced risk in android applications for malicious application detection. *IEEE Trans. Info. Forens. Secur.* (2014).

[54] Yueming Wu, Xiaodi Li, Deqing Zou, Wei Yang, Xin Zhang, and Hai Jin. 2019. MalScan: Fast market-wide mobile malware scanning by social-network centrality analysis. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE'19).*

[55] Lei Xue, Xiapu Luo, Le Yu, Shuai Wang, and Dinghao Wu. 2017. Adaptive unpacking of Android apps. In *Proceedings of the International Conference on Software Engineering (ICSE'17).*

[56] Wei Yang, Mukul Prasad, and Tao Xie. 2018. Enmobile: Entity-based characterization and analysis of mobile malware. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18).*

[57] Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie, and William Enck. 2015. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *Proceedings of the 37th International Conference on Software Engineering (ICSE'15).*

[58] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. 2014. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS'14).*

[59] Yueqian Zhang, Xiapu Luo, and Haoyang Yin. 2015. Dexhunter: Toward extracting hidden code from packed android applications. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS'15).*

[60] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. 2012. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS'12).*

Received February 2020; revised December 2020; accepted December 2020