HomDroid: Detecting Android Covert Malware by Social-Network Homophily Analysis

Yueming Wu[*†]
Huazhong University of Science and Technology
China
wuyueming@hust.edu.cn

Deqing Zou[*†‡§]
Huazhong University of Science and Technology
China
deqingzou@hust.edu.cn

Wei Yang
University of Texas at Dallas
United States
wei.yang@utdallas.edu

Xiang Li[*†]
Huazhong University of Science and Technology
China
xianglilxlx@hust.edu.cn

Hai Jin[†¶]
Huazhong University of Science and Technology
China
hjin@hust.edu.cn

ABSTRACT

Android has become the most popular mobile operating system. Correspondingly, an increasing number of Android malware has been developed and spread to steal users' private information. There exists one type of malware whose benign behaviors are developed to camouflage malicious behaviors. The malicious component occupies a small part of the entire code of the application (app for short), and the malicious part is strongly coupled with the benign part. In this case, the malware may cause false negatives when malware detectors extract features from the entire apps to conduct classification because the malicious features of these apps may be hidden among benign features. Moreover, some previous work aims to divide the entire app into several parts to discover the malicious part. However, the premise of these methods to commence app partition is that the connections between the normal part and the malicious part are weak (*e.g.,* repackaged malware).

In this paper, we call this type of malware as *Android covert malware* and generate the first dataset of covert malware. To detect covert malware samples, we first conduct static analysis to extract the function call graphs. Through the deep analysis on call graphs, we observe that although the correlations between the normal part and the malicious part in these graphs are high, the degree of these correlations has a unique range of distribution. Based on the observation, we design a novel system, *HomDroid*, to detect covert malware by analyzing the homophily of call graphs. We identify the ideal threshold of correlation to distinguish the normal part and the malicious part based on the evaluation results on a dataset of 4,840 benign apps and 3,385 covert malicious apps. According to our evaluation results, *HomDroid* is capable of detecting 96.8% of covert malware while the False Negative Rates of another four state-of-the-art systems (*i.e., PerDroid*, *Drebin*, *MaMaDroid*, and *IntDroid*) are 30.7%, 16.3%, 15.2%, and 10.4%, respectively.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**.

KEYWORDS

Android, Covert Malware, Social Network, Homophily Analysis

ACM Reference Format:
Yueming Wu, Deqing Zou, Wei Yang, Xiang Li, and Hai Jin. 2021. HomDroid: Detecting Android Covert Malware by Social-Network Homophily Analysis. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21), July 11–17, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3460319.3464833

1 INTRODUCTION

The widespread use of smartphones has led to a rapid increase in the number of mobile malware. Especially in recent years, mobile phones based on the Android operating system have occupied a dominant position in the smartphone market [3, 4], and the Android platform has become the target of choice for attackers due to its huge market share and open-source features. To hide the malicious behaviors, attackers have created certain stealthy malware samples [6–8]. For example, *DEFENSOR ID* [6] removed all potential malicious functionalities but one (*i.e.,* abusing Accessibility Service) to hide its maliciousness. However, as long as this malware is triggered, it can secretly wipe out the victim's bank account or cryptocurrency wallet and take over their email or social media accounts, which may cause huge economic losses [5]. Therefore, accurate analysis of malware behavior characteristics is urgently needed to remove malware from users' daily life.

[*]Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, HUST, Wuhan, 430074, China
[†]National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, HUST, Wuhan, 430074, China
[‡]Shenzhen HUST Research Institute, Shenzhen, 518057, China
[§]Corresponding author
[¶]Cluster and Grid Computing Lab, School of Computer Science and Technology, HUST, Wuhan, 430074, China

ISSTA '21, July 11–17, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8459-9/21/07…$15.00
https://doi.org/10.1145/3460319.3464833

In general, existing mobile malware detection approaches can be classified into two categories, namely syntax-based approaches [17, 20, 46, 51, 55, 62] and semantics-based systems [18, 22, 31, 41, 42, 58–60]. As for syntax-based methods, they ignore the program semantics of apps to achieve high efficient Android malware detection. For instance, some methods [46, 55] only consider permissions requested by apps and extract these permissions from manifest to construct feature vectors. Nevertheless, malware can spread malicious activities without any permissions [35]. To complete more effective Android malware detection, many approaches [31, 32, 60] distill the program semantics into graph representations and detect malware by analyzing generated graphs. These graph-based techniques can indeed achieve high accuracy on malware detection. For example, *MaMaDroid* [42] first obtains all sequences from an extracted call graph and then abstracts them into their corresponding packages. After abstraction, it establishes a Markov chain model to represent the transition probabilities between these packages. The Markov chain model is used to construct feature vectors to detect Android malware. The empirical results [42] have demonstrated the effectiveness of *MaMaDroid* on general malware detection. However, when malicious code accounts for a small part of the entire code of an app, the feature vectors obtained from the normal part and the whole call graph can be highly similar. In this case, *MaMaDroid* will misclassify the malware as a benign app.

To address the issue, some studies [36, 54] intent to partition the entire app to discover the suspicious part to achieve more accurate malware detection. However, these approaches are designed to detect only repackaged Android malware. A repackaged malware is constructed by disassembling benign apps, adding malicious codes, and then reassembling them as new apps. In other words, as discussed in their papers [36, 54], the connections between malicious code and normal code in a repackage malware are expected to be weak because the malicious part is constructed by injection. Therefore, if the malicious code is strongly coupled with normal code, methods [36, 54] may not achieve high accuracy because it is not easy to divide them into different parts. In short, when the malicious code accounts for a small part (*e.g.,* less than 2%) of a malware sample and the connections between normal code and malicious code are strong, prior approaches [36, 42, 54] may cause high false negatives.

In this paper, we call this type of malware as *Android covert malware* and generate the first covert malware dataset (*i.e.,* 3,358 covert malware) by analyzing 100,000 malicious apps' call graphs. In particular, a malicious sample is considered as a covert malware must meet the following two conditions: 1) Nodes in the malicious part occupy a small part (*e.g.,* 2%) of all nodes in the entire call graph; and 2) The normal part and the malicious part are highly correlated. Since this type of malware is highly concealed, it may be possible to detect them by performing advanced static analysis or dynamic analysis. However, the time-cost of these precise analysis methods [58, 59] are expensive (*e.g.,* 291 seconds for *AppContext* [59] to analyze an app), making it difficult to filter and discover covert malware samples from large-scale real apps. To tackle the above issues, we aim to propose a novel approach to detect covert malware efficiently and effectively. Specifically, we mainly address two challenges:

- *Challenge 1: How to identify the malicious code in a covert malware sample?*
- *Challenge 2: How to extract light-weight semantic features to achieve accurate behavior characteristics?*

The first key insight of *HomDroid* is the observation from a deep analysis of correlations between the normal part and the malicious part in covert call graphs, that is, although the correlation between the two parts is high, it has a range of distribution. Specifically, to address the first challenge, we first perform simple static analysis (*i.e.,* flow- and context-insensitive analysis) to distill the program semantics of an app into a function call graph. Then the generated graph is divided into certain communities (*i.e.,* subgraph) by community detection. As malware always performs malicious behaviors by invoking sensitive API calls, therefore, we mainly focus on sensitive API calls in this paper. In other words, communities that do not contain any sensitive API calls will be integrated into a *benign community*, and the rest are *sensitive communities*. For each sensitive community, we perform homophily analysis to obtain the coupling between it and the benign community. If the coupling is above a predetermined threshold[1], the sensitive community will be considered benign. Otherwise, it is treated as a suspicious part of the app. After obtaining all suspicious parts of the app, we integrate them into a subgraph, called *suspicious subgraph*.

The second key insight of *HomDroid* is that triads in social-network-analysis can represent different network structural properties of a network and the extraction of triads from a network is a lightweight task. Specifically, to address the second challenge, we extract two types of feature sets from the suspicious subgraph. We first consider the occurrence of sensitive API calls which are highly correlated with malicious operations. In addition, to maintain the graph details, we collect the ratio of the number of sensitive triads to the total number of triads within the suspicious subgraph. By this, we can achieve semantic and efficient Android covert malware detection.

We develop an automatic system, *HomDroid*, and evaluate it on a dataset of 8,198 apps including 4,840 benign apps and 3,358 covert malware. Compared to four state-of-the-art Android malware detection methods (*i.e., PerDroid* [55], *Drebin* [20], *MaMaDroid* [42], and *IntDroid* [63]), *HomDroid* is able to detect 96.8% of covert malware while the False Negative Rates of *PerDroid*, *Drebin*, *MaMaDroid*, and *IntDroid* are 30.7%, 16.3%, 15.2%, and 10.4%, respectively. Furthermore, as for runtime overhead of *HomDroid*, it consumes about 13.4 seconds to complete the whole analysis of an app in our dataset. Such result indicates that *HomDroid* is extremely efficient than methods that perform complex program analysis (*e.g.,* 291 seconds for *AppContext* [59], 20 minutes for *EnMobile* [58], and 275 seconds for *Apposcopy* [31]).

In summary, this paper makes the following contributions:

- We built the first dataset [13] of Android covert malware and propose a novel technique to discover the most suspicious part of a covert malware by analyzing the homophily of a call graph.
- We implement a prototype system, *HomDroid*, a novel and automatic system that can accurately detect Android covert malware.

[1] In this paper, we select a total of five thresholds: 1, 2, 3, 4, and 5.

- We conduct evaluations using 4,840 benign samples and 3,358 covert malicious samples. Experimental results show that *HomDroid* is capable of detecting Android covert malware with a False Negative Rate of 3.2% while are 30.7%, 16.3%, 15.2%, and 10.4% for four comparative systems (*i.e., PerDroid* [55], *Drebin* [20], *MaMaDroid* [42], and *IntDroid* [63]).

Paper organization. The remainder of the paper is organized as follows. Section 2 presents our motivation. Section 3 introduces our system. Section 4 reports the experimental results. Section 5 discusses the future work and limitations. Section 6 describes the related work. Section 7 concludes the present paper.

2 MOTIVATION

2.1 A Simplified Example

To better illustrate the key insight of our approach, we present a simplified example at first. This example (*i.e., com.cpssw*) is an app that pushes notifications about the scores of users' favorite teams. However, it collects private data such as *International Mobile Equipment Identity* (IMEI), writes them into files, and sends them to a remote server. The normal behavior of this malware is to bring notifications to users, thus it needs to connect to the Internet to complete this purpose. Meanwhile, the malicious activity is to send data to the Internet which can cause privacy leaks. In other words, the normal code and malicious code of this malware both need to connect to the Internet to achieve their corresponding goals.

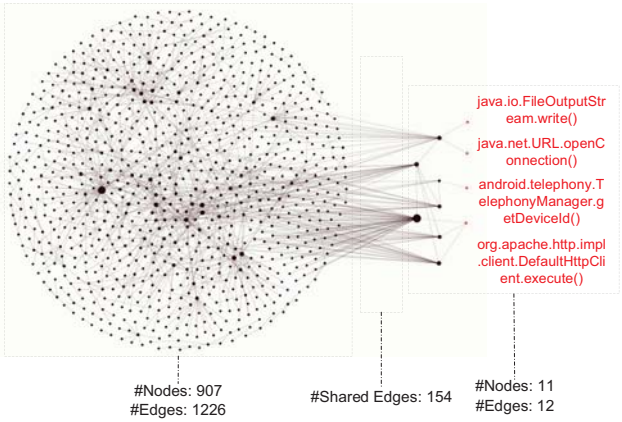

Figure 1: The function call graph of a real malware (com.cpssw)

Because Android apps use API calls to access operating system functionality and system resources, and malware samples always invoke sensitive API calls to perform malicious activities. Therefore, we can leverage sensitive API calls to characterize the malicious behaviors of an app. To obtain the sensitive API calls invoked by the malware example, we upload it to an Android apps analysis system [14] to analyze and generate a detailed behavioral report. Through the report, we find that this malware mainly invokes four sensitive API calls (*i.e., FileOutputStream.write(), URL.openConnection(), TelephonyManager.getDeviceId()*, and *DefaultHttpClient.execute()*) to complete its maliciousness. In other words, the malicious part of the malware can be characterized by these sensitive API calls and their correlative parent nodes.

Figure 1 shows the function call graph of the malware example where the total number of nodes and edges are 918 and 1,392, respectively. It consists of two parts, the left part is made up of normal nodes while the right part includes sensitive API calls and their correlative parent nodes. As shown in Figure 1, the number of nodes of the malicious part is 11 which accounts for about 1.2% of all nodes of the entire call graph. Moreover, the proportion of edges in the malicious part is also low, which is less than 1% of all edges of the whole app. In this case, we may cause a false negative when we extract features from the entire app to conduct classification since the malicious features may be hidden under normal behaviors. In practice, the cosine similarity of two feature vectors obtained from the normal part and the whole call graph by *MaMaDroid* [42] is more than 95%, and it is misclassified by *MaMaDroid* [42] in our experiments.

Some prior studies [36, 54] intent to partition the entire app to discover the suspicious part to achieve a more accurate malware detection. However, these approaches are designed to detect repackaged Android malware and the connections between injected malicious code and legitimate part are expected to be weak. In this malware, the number of edges shared by the normal part and malicious part is 154 which is more than 12 times greater than the total number of edges in the malicious part. To obtain a more determinate result, we regard the call graph in Figure 1 as a social network and conduct social-network-analysis to research the homophily of the network. Homophily is the tendency of individuals to associate and bond with similar others, as in the proverb 'birds of a feather flock together' [43] (Details are in Section 3.3). Suppose that a network consists of two subnetworks, high homophily of the network means that the correlation between these two subnetworks is low. After analyzing Figure 1, we find that the homophily of Figure 1 is not distinct, in other words, the connections between the normal part and malicious part are strong, making it difficult to be handled by methods in prior studies [36, 54].

In conclusion, when the malicious code accounts for a small part of an entire malicious app and the connection between normal code and malicious code is strong, prior approaches [36, 42, 54] may cause high false negatives since the malicious behaviors can be hidden under the normal codes. In this paper, we call this type of malware as *Android covert malware* and construct the first dataset of these malware samples.

2.2 Covert Malware Dataset Construction

To generate the dataset of Android covert malware, we first randomly download about 100,000 malicious apps from AndroZoo [19] which is a growing collection of Android apps collected from several sources, including the official Google Play app market and several third-party Android app markets (*e.g.,* AppChina [12]). After obtaining the samples, we perform static analysis to extract the function call graphs of these apps. As API calls are used by the Android apps to access operating system functionality and system resources, they can be used as representations of the behaviors of Android apps. Moreover, Android malware usually invokes some

sensitive API calls to perform malicious activities. Therefore, we assume these sensitive API calls and their correlated parent nodes as the malicious part and the rest as the normal part. Particularly, we focus on the newest version of API calls set in PScout [21], which is the largest collection of sensitive API calls (*i.e.,* 21,986 API calls).

After dividing the graph into the normal part and malicious part, we build the covert malware dataset according to the following steps: 1) We first conduct statistical analysis to obtain the proportion of malicious nodes in all nodes; 2) We then perform homophily analysis to collect the correlation between the normal part and the malicious part; 3) Next, malware samples with low correlation between the normal part and the malicious part are be omitted; 4) As for the remaining malware, we divide them into six categories according to the proportion: [0-1%), [1%-2%), [2%-3%), [3%-4%), [4%-5%), and greater than 5%; 5) To find which category may be more covert, we randomly select 100 samples from each category and upload them to VirusTotal [16]; 6) After analyzing the scanning results, we find that malware with a proportion of less than 2% is less likely to be detected as malware. Therefore, we pay more attention to these malware samples with a proportion of less than 2% (*i.e.,* 4,321 samples); 7) To further verify whether these samples contain disguised behaviors or not, we conduct dynamic analysis to obtain accurate behaviors. Specifically, we use some state-of-the-art security tools (*e.g., AppCritique* [10] and *Sandroid* [14]) to generate detailed behavior reports, and then manually analyze these reports to check whether sensitive API calls invoked by malicious behaviors are also used by normal behaviors. If there are many such cases, then we consider the malware to be covert. After in-depth manual analysis, we finally obtain 3,358 covert malware samples.

3 SYSTEM ARCHITECTURE

In this section, we propose a novel system *HomDroid*. We further illustrate how *HomDroid* excavates the most suspicious part of an app and extracts semantic features from the suspicious part to detect Android covert malware.

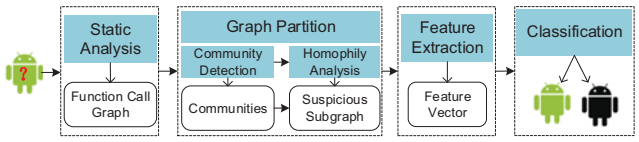

Figure 2: System overview of *HomDroid*

3.1 Overview

As shown in Figure 2, *HomDroid* consists of four main phases: *Static Analysis*, *Graph Partition*, *Feature Extraction*, and *Classification*.

- ***Static Analysis:*** Given an app, we first conduct static analysis to obtain the function call graph where each node is a function that can be an API call or a user-defined function.
- ***Graph Partition:*** After generating the call graph, we then divide it into certain subgraphs by community detection and discover the most suspicious part by homophily analysis. The output of this phase is the most suspicious subgraph.
- ***Feature Extraction:*** Next, two types of feature sets are collected from the suspicious subgraph, including the appearance of sensitive API calls and the ratio of the number of sensitive triads to the total number of triads within the subgraph.
- ***Classification:*** In our final phase, given a feature vector, we can accurately flag it as either benign or malicious by using a trained machine learning classifier.

3.2 Static Analysis

To achieve high accuracy malware detection, we consider maintaining the program semantics of an APK file. In other words, we conduct light-weight static analysis to distill the semantics of an app into a graph representation. More specifically, we implement our static analysis to extract the function call graph of an app based on an Android reverse engineering tool, *Androguard* [28].

3.3 Graph Partition

In this phase, we pay attention to discover the most suspicious part of a function call graph. As shown in Figure 2, our *Graph Partition* phase is composed of two steps which are *Community Detection* and *Homophily Analysis*.

3.3.1 Community Detection. In the study of complex networks, a network is said to have community structure if the nodes of the network can be easily grouped into sets of nodes such that each set of nodes is densely connected internally. As for an Android app, it is made up of certain specific modules and each module completes different functionality. Nodes in one module should be closely connected because they are designed to implement the same functionality in cooperation. Furthermore, a previous study [47] has demonstrated that a software call graph can be treated as a network with community structures. Therefore, in this subsection, we perform community detection to divide a function call graph into certain communities (*i.e.,* subgraphs).

We implement four widely used community detection algorithms (*i.e., infomap* [50], *label propagation* [48], *multi level* [24], and *leading eigenvector* [44]) to check which one is better for us to detect malware. More specifically, we first download 500 benign apps and 500 malicious apps from AndroZoo [19] as our test dataset. Then function call graphs of these apps are extracted by our static analysis. After obtaining 1,000 call graphs, we conduct community detection on them and record the detection results including the values of modularity Q [45] of these call graphs and the runtime overheads of different community detection algorithms. Modularity Q is a metric that can quantify the quality of a detected community structure of a network. The value of Q is between 0 to 1, Q is 0 means that the network has no community structure. On the contrary, if Q is close to 1, the network may have an ideal community structure. According to a previous study [45], the values of Q typically fall in the range from about 0.3 to 0.7 and higher values are rare.

Figure 3 presents the community detection results on our 1,000 randomly downloaded apps (*i.e.,* 500 benign apps and 500 malicious apps). On the one hand, the average values of modularity Q of communities generated by *informap*, *label propagation*, *multi level*, and *leading eigenvector* are 0.60, 0.52, 0.66, and 0.54, respectively.

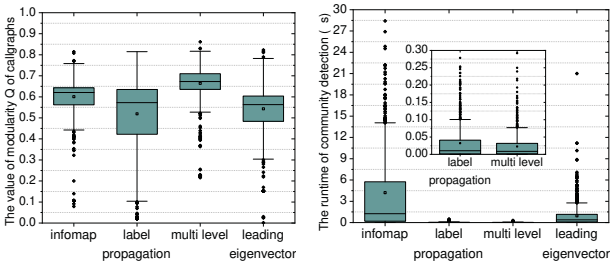

Figure 3: The values of modularity Q of 1,000 call graphs and the runtime overheads of four different community detection algorithms (*i.e., infomap* [50], *label propagation* [48], *multi level* [24], and *leading eigenvector* [44])

Such result indicates that communities obtained by *multi level* have better community structure than the other three algorithms. On the other hand, as for the runtime of community detection, *multi level* consumes the least runtime overhead which means that it is faster than *informap*, *label propagation*, and *leading eigenvector*. Therefore, by comprehensively considering the modularity Q and the runtime of community detection, we finally decide to choose *multi level* as our community detection algorithm.

3.3.2 Homophily Analysis. Homophily is the tendency of individuals to associate and bond with similar others. Homophily has been investigated in many network analysis studies [27, 30, 33, 53]. In this subsection, we use a simple example to better illustrate how to quantify the homophily of a social network.

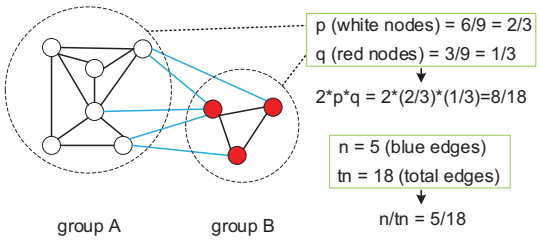

Figure 4: An assumed social network which consists of two groups

Figure 4 presents the friendships between students in two groups and connections between them indicate that they are friends. For simplicity, we use white nodes and red nodes to represent students in groups A and B, respectively. Suppose that the proportions of white nodes and red nodes in Figure 4 are p and q, respectively. Because whether the color of a node is red or blue is an independent random process, the probability that a node is white is p and the probability of red is q. Then the probability that two nodes on an edge are different can be computed as $p*q$ (*i.e.,* the color of the left node is white while is red for the right node) + $q*p$ (*i.e.,* the color of the left node is red while is white for the right node) = $2*p*q$. Moreover, if the total number of edges in Figure 4 is tn and the number of edges between groups A and B is n. Then the proportion of edges with nodes of different colors can be calculated as n/tn.

After obtaining the probability that two nodes are different $2*p*q$ and the proportion of edges with two nodes of different colors n/tn, we can analyze the homophily of the social network in Figure 4 by comparing the value of $2*p*q$ and n/tn. In other words, when n/tn is less than or equal to $2*p*q$, the homophily of this social network will be considered high and the correlation between groups A and B is low. On the contrary, if n/tn is greater than $2*p*q$, the social network will have low homophily while the correlation between groups A and B is high.

In Figure 4, the proportion of white nodes and red nodes are 2/3 and 1/3, respectively. The number of edges with nodes of different colors is 5 and the total number of edges is 18. In other words, p, q, t, and tn in Figure 4 are 2/3, 1/3, 5, and 18, respectively. Therefore, we can claim that the homophily of the social network in Figure 4 is high because 5/18 (*i.e., n/tn*) is less than 8/18 (*i.e., 2*p*q*).

As stated earlier, the purpose of this phase is to discover the most suspicious part of a given function call graph. In reality, Android malware samples usually invoke some sensitive API calls to spread malicious activities. For example, *getDeviceID()* can get your phone's IMEI and *getLine1Number()* can obtain your phone number. To achieve efficient covert malware detection, we only consider a small part of sensitive API calls. Specifically, we choose sensitive API calls reported in a recent work [34] as our objectives which composes of three different API call sets. The first API call set is the top 260 API calls with the highest correlation with malware, the second API call set is 112 API calls that relate to restrictive permissions, and the third API call set is 70 API calls that are relevant to sensitive operations. In final, 426 sensitive API calls are obtained by computing the union set of these three API call sets. However, benign apps may also invoke several sensitive API calls to complete some functionalities (*e.g.,* push functionality). For instance, some social apps may require to access users' location for presenting location-specific news or videos. In the situation they also need to invoke a sensitive API call *LocationManager.getLastLocation()*. Therefore, in an effort to achieve more accurate malicious behavior characteristics, we perform homophily analysis to deeply analyze and extract the most suspicious part of a call graph.

Specifically, we define the coupling between two graphs a and b by computing the quotient of n/tn and $2*p*q$:

$$c(a,b) = \frac{\frac{s}{e_a+e_b}}{2*\frac{n_a}{n_a+n_b}*\frac{n_b}{n_a+n_b}}$$

Note that the number of nodes and edges in graph a are n_a and e_a while are n_b and e_b in graph b. Furthermore, the number of edges shared by graph a and graph b is s.

Given an APK file, we first extract the function call graph by static analysis, then the call graph is divided into certain communities (*i.e.,* subgraphs) by community detection. Communities that do not contain any sensitive API calls will be integrated into a *benign community*, and the rest are *sensitive communities*. For each sensitive community, we perform homophily analysis to obtain the coupling between it and the benign community. If the coupling is above a threshold, the sensitive community should be considered benign. Otherwise, it is treated as a suspicious part of the app. After obtaining all suspicious parts of the app, we integrate them into a subgraph which is the most suspicious subgraph of the function call graph. In this subsection, the purpose of our homophily analysis is

to filter the normal parts in sensitive communities (*i.e.,* subgraph) to generate a more accurate suspicious subgraph.

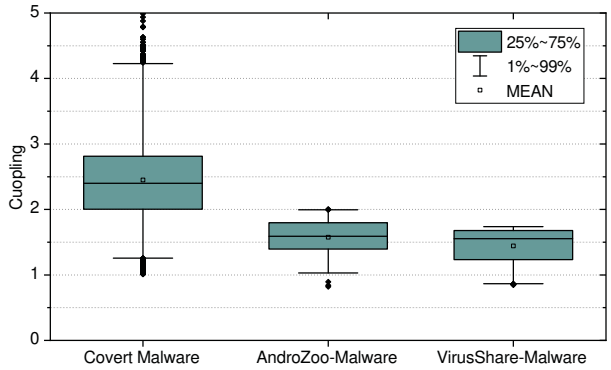

Figure 5: The coupling values between the normal part and the malicious part in Android covert malware and general malware

In order to select suitable thresholds of coupling, we first conduct a simple study. Specifically, we consider 426 sensitive API calls and their correlated nodes as the malicious part and the rest nodes are the normal part. Then the values of coupling of 3,358 covert malware are extracted. Moreover, to verify the assumption that the connections between the malicious part and the normal part of covert malware are higher than that of general malware. We also randomly download 3,000 malware samples from AndroZoo [19] and VirusShare [11], respectively. As aforementioned, we discover 3,358 covert malware from 100,000 malware samples, the percentage is only 3.358%, which means that most of the malware in our randomly downloaded samples are general malware. Figure 5 presents the coupling of 3,358 Android covert malware and our randomly downloaded samples from AndroZoo [19] and VirusShare [11]. Through the results in Figure 5, we observe that the coupling values between the malicious part and the normal part of covert malware are higher than that of most of general malware, and almost all values of coupling in covert malware are between 1 to 5. Therefore, we finally choose five thresholds of coupling in this paper, which are 1, 2, 3, 4, and 5.

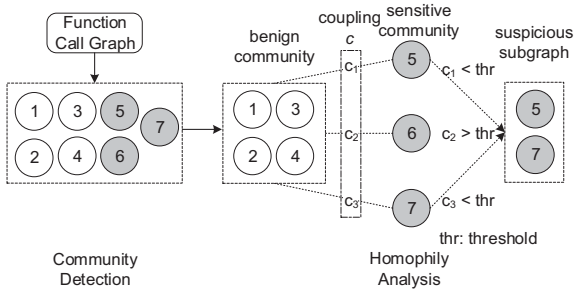

Figure 6: An example of community detection and homophily analysis

To better illustrate the different steps involved in community detection and homophily analysis, we present an example in Figure 6. Suppose that a function call graph is partitioned into seven communities and communities 1-4 do not contain any sensitive API calls. After generating the benign community by integrating communities 1-4, we perform homophily analysis to compute the coupling between sensitive communities 5-7 and the benign community one by one. The analysis results show that the coupling between community 6 and the benign community is less than a given threshold while is greater than the threshold for communities 5 and 7. Therefore, we only consider communities 5 and 7 as suspicious parts and integrate them into a subgraph to represent the most suspicious part of the function call graph.

3.4 Feature Extraction

After obtaining the most suspicious subgraph of a call graph, we then extract features from the subgraph. More specifically, we collect two types of feature sets including the appearance of sensitive API calls and the ratio of the number of sensitive triads to the total number of triads within the subgraph.

Our first type of feature set focuses on the occurrence of sensitive API calls. As aforementioned, we select 426 most suspicious sensitive API calls [34] as our concerned objectives. Given the most suspicious subgraph, we check whether the nodes in the subgraph contain any sensitive API calls. If sensitive API calls appear in the subgraph, the value of the corresponding feature will be one, otherwise, it is zero. The dimension of our first type of feature vector is the total number of sensitive API calls, which is 426.

Furthermore, to maintain the graph semantics of the subgraph, we also consider the graph structure details to construct our second feature set. In social network analysis, there are 16 different types [23] of triads in a network. In practice, these triads can represent different network structure properties of a network. Therefore, we select a total of six types of triads from the 16 types [23] since sensitive API calls are always being invoked by other functions to perform malicious activities in Android malware.

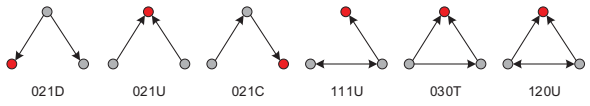

Figure 7: The selected six types of sensitive triads where the red nodes are sensitive API calls

In Figure 7, we present our selected six types of triads which can represent different graph structure details. For example, a sensitive triad of *021U* shows that two normal functions invoke a same sensitive API call while *021C* indicates that one normal function f_1 first invokes another normal function f_2 and then normal function f_2 invokes a sensitive API call. By extracting these sensitive triads from the suspicious subgraph, we are able to characterize malicious behaviors to achieve semantic Android malware detection. More specifically, we first extract all these six types of triads from a given suspicious subgraph. For each type of triad, we then collect sensitive triads by checking whether a triad contains any sensitive API call or not. After obtaining all sensitive triads and all triads, the ratio

of the number of sensitive triads to the total number of triads for each sensitive API call will be calculated as the features.

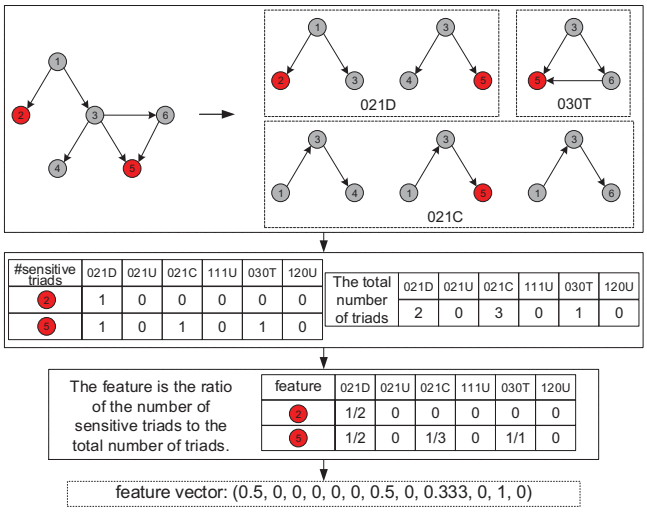

Figure 8: An example to illustrate how we construct the feature vector about the ratio of the number of sensitive triads to the total number of triads

To better describe the construction of the features of sensitive triads, we present an example in Figure 8. Assuming that a suspicious subgraph contains six nodes, and two of them are sensitive API calls. After collecting all the triads from the subgraph, we find that there are four sensitive triads including two sensitive triads of *021D*, one sensitive triad of *021C*, and one sensitive triad of *030T*. In addition, the total number of six types (*i.e., 021D*, *021U*, *021C*, *111U*, *030T*, and *120U*) of triads in the subgraph are two, zero, three, zero, one, and zero, respectively. For each sensitive API call, we compute the ratio of the number of corresponding sensitive triads to the total number of triads for each type of triad. For example, the number of triads of *021C* of sensitive API call **5** is one and the total number of triads of *021C* in the subgraph is three, then the feature of sensitive API call **5** for *021C* will be calculated as 1/3. Moreover, if the subgraph does not contain sensitive triads of certain types, the corresponding features will be set zero directly. Finally, we can obtain a feature vector whose dimension is the number of sensitive API calls multiply by six (*i.e.,* six types of sensitive triads). In *HomDroid*, the number of selected sensitive API calls is 426, then the dimension of the feature vector of sensitive triads is 426∗6=2,556.

After extracting the mentioned two types of features, we concatenate them as our final feature vector whose dimension is 2,982.

3.5 Classification

Given extracted feature vectors, our final phase focuses on training a classifier first and then uses it to detect Android covert malware. More specifically, in order to check the ability of *HomDroid* on detecting covert malware with different classifiers, we implement six classification algorithms (*i.e.,* 1-Nearest Neighbor, 3-Nearest Neighbor, Random Forest, Decision Tree, SVM, and Logistic Regression) by using a python library scikit-learn [15]. Parameters in these algorithms are default parameters in scikit-learn [15], we leverage them to commence our evaluations and all the experimental results are presented in the following section.

4 EVALUATIONS

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

- *RQ1: How effective is HomDroid on detecting Android covert malware in various setups?*
- *RQ2: How does HomDroid perform compared to other state-of-the-art Android malware detection systems?*
- *RQ3: What is the runtime overhead of HomDroid on detecting Android malware?*

4.1 Dataset and Metrics

Table 1: Summary of the dataset used in our experiments

Category	#Apps	Average Size (MB)
Benign apps	4,840	2.7
Covert malware	3,358	3.2

As aforementioned, we obtain 3,358 Android covert malware by analyzing and filtering 100,000 malicious apps from AndroZoo [19]. In order to evaluate the effectiveness of *HomDroid* on detecting Android covert malware, we also crawl benign apps from AndroZoo. Our final dataset consists of 4,840 benign apps and 3,358 covert malware, and the average sizes (in Table 1) of these benign apps and covert malware are 2.7 MB and 3.2 MB, respectively. We leverage this dataset to commence our evaluations and the experimental results are reported in the following subsections. Note that all experiments are conducted by performing 10-fold cross-validations, which means that the dataset is partitioned into 10 subsets, each time we pick one subset as our testing set and the rest nine subsets as training set. We repeat this 10 times and report the average as our final results.

In addition, we adopt several widely used metrics to measure the experimental results of *HomDroid*. These metrics are presented in Table 2, in which *True Positive Rate* (TPR) and *False Negative*

Table 2: 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)

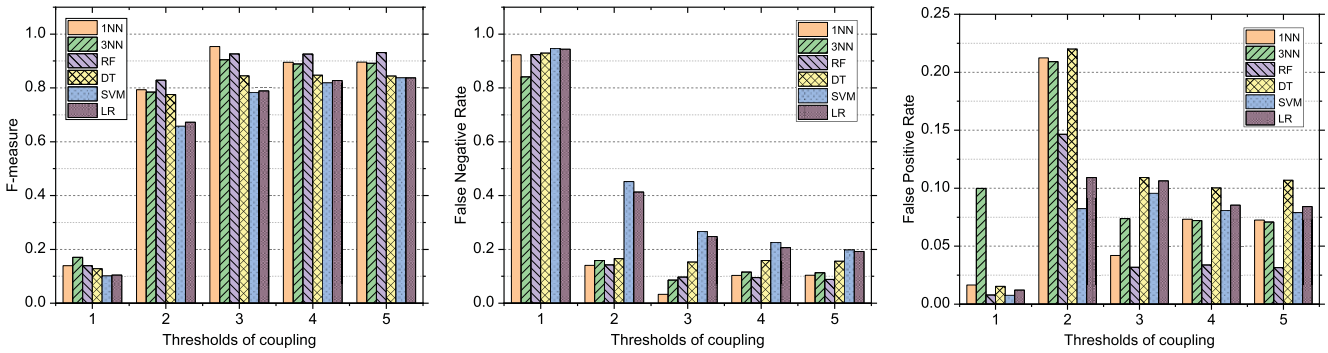

Figure 9: F-measure, FNR, and FPR of *HomDroid* on detecting Android covert malware with different classification models

Rate (FNR) present the effectiveness on detecting malicious samples while *True Negative Rate* (TNR) and *False Positive Rate* (FPR) show the ability on benign samples detection. In our experiments, we report both FNR and FPR to see how *HomDroid* performs on classifying both malicious and benign samples. Moreover, we also report F-measure and Accuracy for presenting the overall detection effectiveness of *HomDroid*.

4.2 Detection Effectiveness

In this phase, we conduct several experiments to examine the ability of *HomDroid* on Android covert malware detection by using the collected dataset in Table 1. Specifically, we evaluate the effectiveness of *HomDroid* from the following two aspects:

- Different classification models: *1-Nearest Neighbor* (1NN), *3-Nearest Neighbor* (3NN), *Random Forest* (RF), *Decision Tree* (DT), *Support Vector Machine* (SVM), and *Logistic Regression* (LR).
- Different thresholds of coupling: 1, 2, 3, 4, and 5.

4.2.1 Different Classification Models. As aforementioned, we implement six different machine learning algorithms by using a python library scikit-learn [15]. These implemented classifiers (*i.e.,* 1NN, 3NN, RF, DT, SVM, and LR) are used to evaluate the effectiveness of *HomDroid* on detecting Android covert malware. Figure 9 reports the experimental results including the F-measure, Accuracy, *False Negative Rate* (FNR), and *False Positive Rate* (FPR) achieved by *HomDroid* with different classifiers. From the results in Figure 9, we can see that classifier 1NN is able to achieve the best overall effectiveness than the other five classifiers. For example, when we select the threshold of coupling as 3, the F-measure of *HomDroid* is the highest among all F-measures in Figure 9 when we adopt 1NN to detect malware. The F-measure is 95.3% while is 90.4%, 92.6%, 84.5%, 78.3%, and 78.9% for 3NN, RF, DT, SVM, and LR, respectively.

As for the ability of *HomDroid* on detecting covert malware, the FNR of *HomDroid* is lowest when we choose the threshold of coupling as 3 and 1NN as our final classifier. In such case, *HomDroid* only misclassifies about 3.2% of covert malware as benign apps, which can demonstrate that *HomDroid* has high effectiveness on detecting Android covert malware. As regards the effectiveness of *HomDroid* on distinguishing benign samples, RF is able to maintain the lowest false positives than 1NN, 3NN, DT, SVM, and LR no matter which threshold of coupling is selected. However, the results of RF are not as good as 1NN in terms of F-measure, Accuracy, and FNR when we choose 3 as the threshold of coupling. As a matter of fact, it is crucial to maintain a low FNR for Android malware detection because high FNR signifies more malicious samples being misclassified as benign samples, and these misclassified malicious samples can still spread malicious activities. When users install these covert malware samples, their private data may be stolen by attackers, which may cause different levels of economic losses.

In conclusion, *HomDroid* can obtain better effectiveness when we adopt 1NN to train a classifier and use it to detect Android covert malware.

Table 3: F-measure, Accuracy, FNR, and FPR of *HomDroid* on detecting Android covert malware with 1NN

Thresholds	F-measure	Accuracy	FNR	FPR
1	13.9%	61.3%	92.3%	1.7%
2	79.3%	81.7%	14.1%	21.2%
3	95.4%	96.2%	3.2%	4.2%
4	89.5%	91.4%	10.3%	7.3%
5	89.6%	91.5%	10.4%	7.3%

4.2.2 Different Thresholds of Coupling. In our experiments, we choose five values of thresholds to commence our evaluations according to the result in Figure 5. Figure 5 shows that most of the coupling is between 1 to 5, therefore, we select 1, 2, 3, 4, and 5 as our final thresholds to examine the ability of *HomDroid* on covert malware detection.

Through the results in Figure 9 and Table 3, we can see that *HomDroid* with 1NN is able to maintain the best effectiveness when the threshold of coupling is 3. In particular, the FNR and FPR of *HomDroid* are only 3.2% and 4.2%, which means that about 96.8% of covert malware and 95.8% of benign apps can be correctly classified. Such result is encouraging and demonstrates that *HomDroid* is able to accurately detect Android covert malware.

In short, *HomDroid* can achieve the best effectiveness when we select 3 as our threshold of coupling to generate the most suspicious subgraph and use 1NN to detect covert malware.

4.3 Comparison with Prior Work

In this phase, we perform comparative experiments of *HomDroid* with four state-of-the-art Android malware detection approaches: *PerDroid*[2] [55], *Drebin* [20], *MaMaDroid* [42], and *IntDroid* [63]

4.3.1 With PerDroid. *PerDroid* [55] detects Android malware by analyzing risky permissions requested by an app. It scans the manifest file to collect the list of all permissions, and then applies 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, thus can be represented as a type of apps' behavior.

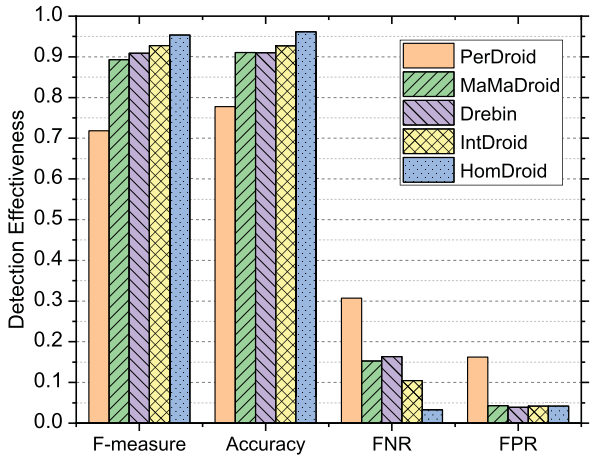

Figure 10: F-measure, Accuracy, FNR, and FPR of *PerDroid*, *MaMaDroid*, *IntDroid*, and *HomDroid* on detecting Android covert malware

The authors [55] have published their risky permissions in their open website [1]. Therefore, we directly adopt the list of their published top 88 risky permissions as our features. More specifically, we leverage *Androguard* [28] to implement the feature extraction and scikit-learn [15] to design six classifiers (*i.e.,* 1NN, 3NN, RF, DT, SVM, and LR) for completing our malware detection. The experimental results on our collected dataset indicate that *PerDroid* is able to maintain the best effectiveness when we choose 3NN as our classifier. Therefore, we select the result of 3NN as the effectiveness of *PerDroid* and show the comparative results with *HomDroid* in Figure 10.

Results in Figure 10 indicate that *HomDroid* can detect more covert malware and distinguish more benign apps than *PerDroid*. The FNR and FPR of *PerDroid* are 30.7% and 16.4% on detecting covert malware while are 3.2% and 4.2% for *HomDroid*. In Table 4, we also present the FNR of *PerDroid* on general malware detection and the result is directly adopted from their paper [55]. It is obvious that the effectiveness of *PerDroid* drops a lot when detecting covert malware. The FNR increases from 5.4% to 30.7%, such result

[2]For more convenient discussion, we call the system in [55] as *PerDroid* since it is a **per**mission-based method.

Table 4: FNR of *PerDroid*, *Drebin*, *MaMaDroid*, *IntDroid*, and *HomDroid* on detecting general malware and covert malware (The results of general malware detection are all directly adopted from their published paper [20, 42, 55, 63])

Methods	General Malware	Covert Malware
PerDroid	0.054	0.307
Drebin	0.06	0.163
MaMaDroid	0.035	0.152
IntDroid	0.009	0.104
HomDroid	—	**0.032**

indicates that *PerDroid* is not suitable to detect covert malware. This is mainly due to the fact that *PerDroid* only pays attention to requested permissions and ignores the program details of app code. However, in order to complete the concealment of malicious behaviors, covert malware samples may use the same permissions as normal code to invoke other sensitive API calls. Moreover, malware can even perform malicious activities without any permissions [35]. On the contrary, *HomDroid* distills the program semantics of app code into a call graph to detect malware, the consideration of program semantics makes *HomDroid* more effective than *PerDroid*.

4.3.2 With Drebin. In order to mitigate the inaccuracies in *PerDroid*, *Drebin* [20] has been designed to extract features not only from manifest but also from app code to complete effective Android malware detection. More specifically, *Drebin* considers extracting eight different types of feature sets from an app, four of which are extracted from the manifest, and the others are obtained from app code. Feature sets in the manifest consist 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. After extracting all the feature sets from an app, *Drebin* embeds them into a joint vector space to train an SVM to detect malware.

Through the experimental results shown in Figure 10, we can see that *Drebin* can achieve better effectiveness than *PerDroid* since *Drebin* considers features from both manifest and app code. For example, *Drebin* is able to detect 83.7% of covert malware while *PerDroid* can only distinguish 69.3% of these malware samples. However, although *Drebin* is better than *PerDroid*, it still behinds *HomDroid*. It is reasonable because *Drebin* only searches for the presence of certain strings (*i.e.,* features) rather than considering the program semantics (*e.g.,* invocations between functions). In addition, results in Table 4 show that the FNR of *Drebin* increases from 6% to 16.3% while is only 3.2% for *HomDroid* when detecting covert malware. Such case indicates that *HomDroid* is superior to *Drebin* on discovering Android covert malware. This happens because the malicious code of covert malware only accounts for a small part of the entire app, making the features obtained from the benign part and the entire part similar.

4.3.3 With MaMaDroid. To complete more comprehensive comparison, we compare *HomDroid* with another graph-based Android malware detection method, namely *MaMaDroid* [42]. Similar to *HomDroid*, *MaMaDroid* first extracts the call graph of an app based on static analysis, then all the sequences are obtained from the call graph and are abstracted into the corresponding packages to model the app's invocation behaviors. Specifically, it establishes a Markov

chain model to represent the transition probabilities between functions. Markov chains stand for multiple pairs of call relationships performed by an app and are used to construct feature vectors to detect malware.

The authors [42] have published their partial source code of *MaMaDroid* [2]. We leverage the open-source code to complete the abstraction, modeling, and feature extraction of *MaMaDroid*. The classification phase is not provided in their code, so we implement the phase by using scikit-learn [15]. Specifically, we accomplish the RF classifier to commence our comparative experiments since RF can achieve the best effectiveness as reported in their paper [42].

Comparative results of *HomDroid* and *MaMaDroid* in terms of F-measure, Accuracy, FNR, and FPR are presented in Figure 10. As we can see from Figure 10, *MaMaDroid* outperforms *PerDroid* since *MaMaDroid* maintains the program semantics by distilling them into a call graph. However, compared to *HomDroid*, *MaMaDroid* detects less covert malware. Specifically, *MaMaDroid* can only detect 84.8% of covert malware while *HomDroid* is capable of achieving a TPR of 96.8%. In addition, similar to *Drebin*, the effectiveness of *MaMaDroid* also decreases when detecting covert malware, such results are reasonable because of the two following reasons: 1) The abstraction phase of *MaMaDroid* may incur certain inaccuracies. For instance, *android.telephony.TelephonyManager.getDeviceId()* and *android.telephony.SmsManager.sendTextMessage()* are both abstracted in *android.telephony* package while their usages and corresponding security-levels are completely different; and 2) *MaMaDroid* extracts features from the entire app, which may cause false negatives since the malicious part occupies only a small part of the entire app.

4.3.4 With IntDroid. Our final comparative system is *IntDroid* [63], which combines social-network-analysis-based technique with traditional graph-based method to achieve effective Android malware detection. Specifically, *IntDroid* first applies centrality analysis to obtain the central nodes in a function call graph, then the average intimacy between sensitive API calls and these central nodes are computed as the semantic features. To complete our comparative evaluations, we adopt the best parameters in their paper [63] to commence our experiments. In other words, we select nodes with top 3% all centralities as central nodes and apply 1NN to train a model for malware detection. The comparative results are shown in Figure 10 and Table 4.

Through the results in Figure 10 and Table 4, we see that *IntDroid* performs better than *PerDroid*. This happens because *IntDroid* also distills the program semantics into a call graph and then extracts semantic features from the graph. However, when using *IntDroid* to detect covert malware, the detection effectiveness drops a lot. For example, *IntDroid* achieves 99.1% TPR when detecting general malware while the TPR can decrease to 89.6% when encountering covert malware. It is reasonable because the analysis object of *IntDroid* is the entire app, however, the malicious part of covert malware only accounts for a small part of the entire app. It may cause inaccuracies since the malicious features may be hidden under the benign features.

In conclusion, in distinguishing benign apps, *HomDroid* is similar to other comparison methods because their FPRs are similar.

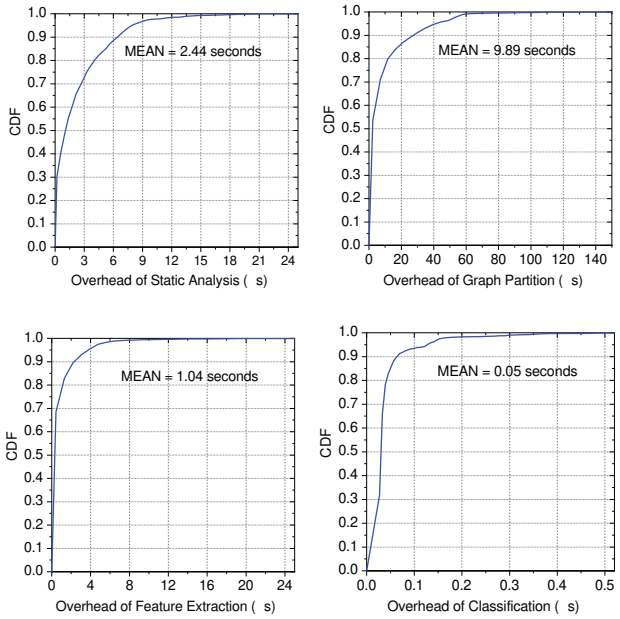

Figure 11: The *Cumulative Distribution Function* (CDF) of runtime overheads of *HomDroid* on different phases (seconds)

However, *HomDroid*'s TPR is at least 7% higher than other comparison tools, which shows that *HomDroid* can find 7% more covert malware than other tools.

4.4 Runtime Overhead

In this phase, we pay attention to evaluate the runtime overhead of *HomDroid*. As aforementioned, *HomDroid* composes of four steps to analyze an app, which are *Static Analysis*, *Graph Partition*, *Feature Extraction*, and *Classification*. We introduce the corresponding overhead in Figure 11 according to the different steps involved in *HomDroid*. Our dataset consists of 4,840 benign apps and 3,385 covert malware, the average nodes and edges of our 8,198 apps are 5,615 and 12,131, respectively.

(1) Static Analysis: *HomDroid* is a semantics-based Android covert malware detection method, it distills the program semantics of an app into a function call graph by static analysis. The runtime overhead of function call graph extraction is given in Figure 11, it needs to take about 2.44 seconds to complete the static analysis on average. Moreover, more than 90% of apps in our dataset can be obtained the call graphs within 8 seconds.

(2) Graph Partition: After obtaining the call graph of an app, we then perform community detection and homophily analysis to dig out the most suspicious subgraph. This step is the most time-consuming phase among all the steps in *HomDroid*. As shown in Figure 11, the average runtime overhead to partition a call graph is 9.89 seconds, and more than 85% of call graphs are able to be discovered the corresponding most suspicious subgraph within 20 seconds.

(3) Feature Extraction: Given the most suspicious subgraph, we extract two types of feature sets from it including the presence of

certain sensitive API calls and the ratio of the number of sensitive triads to the total number of triads within the subgraph. This step requires less time than the former two steps (*i.e.,* static analysis and graph partition). Specifically, on average, it takes about 1.04 seconds for *HomDroid* to achieve the feature extraction from a suspicious subgraph.

(4) Classification: The final step of *HomDroid* is to perform classification by a classifier. The classifier is first trained by using feature vectors extracted from the feature extraction step. Figure 11 shows the runtime overhead of *HomDroid* when we select 1NN as our classifier. In practice, classification consumes the least runtime overhead, it only requires about 0.05 seconds to distinguish a feature vector as either benign or malicious.

Table 5: The comparative runtime overheads of *PerDroid*, *Drebin*, *MaMaDroid*, *IntDroid*, and *HomDroid* on analyzing our datset

Methods	Average Runtime (s)
PerDroid	5.67
Drebin	29.8
MaMaDroid	60.4
IntDroid	40.3
HomDroid	13.4

We also compare the runtime overhead of *HomDroid* with *PerDroid*, *Drebin*, *MaMaDroid*, and *IntDroid* as shown in Table 5. It is obvious that *PerDroid* is the fastest system to detect malware among these compared four systems since *PerDroid* only extracts the requested permissions from the manifest file of an app. However, its detection effectiveness is the lowest due to the lack of consideration of semantics from app code. As for *Drebin*, it collects eight different types of feature sets from both manifest and app code, and these feature sets consist of several complex features (*e.g.,* network address), thus making it time-consuming to complete the whole analysis of an app. As regards *MaMaDroid*, due to the complex static analysis and over many features (*i.e.,* 115,600 features), the runtime overhead of *MaMaDroid* is more expensive than *Drebin* and *HomDroid*. As for *IntDroid*, it combines social network analysis with traditional graph analysis. Although the runtime overhead of social network analysis is small, the procedure of traditional graph analysis is more time-consuming. In other words, *HomDroid* achieves a more efficient malware detection than *Drebin*, *MaMaDroid*, and *IntDroid*.

5 DISCUSSIONS and LIMITATIONS

5.1 Discussions

(1) The use of homophily analysis. In our work, the generation of covert malware should meet that the normal parts and the malicious parts are highly correlated. In this generation phase, the use of homophily analysis is to check whether the normal parts and the malicious parts are highly correlated or not. Besides, in *graph partition* phase of *HomDroid*, the purpose of homophily analysis is to filter out the normal parts from sensitive subgraphs to generate an accurate suspicious subgraph. Although we choose to leverage homophily analysis in both phases, the correlation analysis in the two phases are independent. The use of same analysis technique will not incur bias in the final results.

(2) Why can HomDroid detect covert malware? Existing Android malware detection methods analyze the entire app to extract features, however, the malicious part of covert malware only occupies a small part of the entire app, thus the malicious features may be hidden under benign features. Moreover, some repackaged malware detection techniques first divide the entire app into several parts and then discover the most suspicious part. The premise of these methods to commence app partition is that the connections between the normal part and the malicious part are weak. However, covert malware samples do not fit the situation. *HomDroid* performs homophily analysis to discover the most suspicious parts, which has the ability to discover the suspicious parts from the entire graph although the connections between the normal part and the malicious part are strong.

(3) The application of HomDroid. The design of *HomDroid* is to detect Android covert malware, therefore, we can combine *HomDroid* with other state-of-the-art general malware detection techniques (*e.g., IntDroid*) to complete more comprehensive malware discovery. For example, *IntDroid* can be used as the first line of defense to filter most of malware (*i.e.,* general malware), then *HomDroid* can be applied as the second line of defense to discover more malware (*i.e.,* covert malware).

(4) Using different triads to detect covert malware. Since sensitive API calls are always being invoked by other functions to perform malicious activities, therefore, we only select six types of triads to commence our feature extraction. In reality, we also apply a feature ranking method (*i.e.,* T-test) to research the ability of these six triads on covert malware detection. The results show that *021U* triad ranks first. In our future work, we will construct different combinations of triads to find the most suitable combination to achieve better detection results.

5.2 Limitations

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

(1) Call graph extraction. In this paper, our static analysis phase is implemented by leveraging *Androguard* [28]. In reality, function call graph extracted by *Androguard* [28] is a context- and flow-insensitive call graph. We ignore these information for achieving high efficiency to detect covert malware. To mitigate the inaccuracies caused by our constructed call graph, we plan to use advanced program analysis to generate a suitable call graph to achieve the balance between the efficiency and effectiveness on detecting covert malware.

(2) Obfuscations. Since *HomDroid* is a graph-based method, it can resist some typical obfuscations (*e.g.,* rename obfuscation), however, apps can use reflection [49] to call sensitive API calls, in this 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 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, since *HomDroid* is to perform static analysis to extract the call graph, it is vulnerable to dynamic loading and encryption (*e.g., APK Protect* [9]). As for dynamic loading, it is the technique through which a

computer program at runtime loads a library into memory. Thus all static-analysis-based methods suffer from this limitation. As for 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 such as *PackerGrind* [56] to recover the actual Dex files, then static analysis can be applied to extract call graph.

(3) Sensitive API calls. *HomDroid* mainly focuses on 426 sensitive API calls that are highly correlated with malicious operations [34]. These sensitive API calls account for a small part of the whole sensitive API calls. We plan to conduct statistical analysis to select more valuable sensitive API calls to commence our experiments.

(4) The partition of our dataset. We perform 10-fold cross-validations to generate our evaluation results, which means that the dataset is partitioned into 10 subsets, each time we pick one subset as our testing set and the rest nine subsets as training set. We repeat this 10 times and report the average. In this partitioning phase, we do not distinguish the family labels of our covert malware. In practice, the result may overfit when samples from the same family are not used in both training and testing sets. We plan to label the family first by using *Avclass* [52], and then make sure that samples from the same family are used in both training and testing sets.

6 RELATED WORK

The Android platform has become the main target of choice for attackers, posing a serious threat to users' safety and privacy. Therefore, it is of great significance to establish a healthy mobile app market. In recent years, academia and industry have done a lot of important studies for completing the purpose.

Syntax-based methods [17, 20, 39, 40, 46, 51, 55, 62] ignore the semantics of app code to achieve efficient Android malware detection. For example, Wang *et al.* [55] detects Android malware by analyzing risky permissions requested by an app. 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. Because of the lack of program semantics, such approach suffers from low effectiveness on detecting Android malware. *Drebin* [20] considers extracting features from both manifest and app code. After extracting all the features, it embeds them into a joint vector space to train an SVM to detect malware. However, it only searches for the presence of particular strings, such as some restricted API calls, rather than considers the program semantics. So it can be easily evaded by attacks on syntax features [26].

In order to maintain high effectiveness on detecting Android malware, researchers [18, 22, 25, 29, 31, 36, 38, 41, 42, 54, 57–61] conduct program analysis to extract different types of app semantics. For example, *MaMaDroid* [42] first performs static analysis to obtain the graph representation of an app, then all the sequences are obtained from the graph and are abstracted into the corresponding packages to model the app's invocation behaviors. In practice, the abstraction of *MaMaDroid* may bring some false alarms, for instance, *android.telephony.TelephonyManager.getDeviceId()* and *android.telephony.SmsManager.sendTextMessage()* are both abstracted in *android.telephony* package while their usages and corresponding security-levels are completely different. The main ideas of *DroidSIFT* [60] and *Apposcopy* [31] are similar, that is, they both first extract graph representation of an app, and then conduct graph matching to detect malware. However, since heavy-weight program analyses are both performed by *DroidSIFT* and *Apposcopy* to obtain accurate graphs, they all suffer from low scalability. In their papers [31, 60], they report that the average runtime overhead are 175.8 seconds and 275 seconds for *DroidSIFT* and *Apposcopy*, respectively.

7 CONCLUSION

In this paper, we generate the first dataset of Android covert malware. To detect these covert malware samples, we design a new technique to discover the most suspicious part of covert malware by analyzing the homophily of a call graph. We implement a prototype system, *HomDroid*, a novel and automatic system that can accurately detect Android covert malware. We conduct evaluations using 4,840 benign samples and 3,358 covert malicious samples. Experimental results show that *HomDroid* is capable of detecting Android covert malware with a False Negative Rate of 3.2% while are 30.7%, 16.3%, 15.2%, and 10.4% for four comparative systems (*i.e.*, *PerDroid* [55], *Drebin* [20], *MaMaDroid* [42], and *IntDroid* [63]).

ACKNOWLEDGEMENTS

We would thank the anonymous reviewers for their insightful comments to improve the quality of the paper. This work is supported by the Key Program of National Science Foundation of China under Grant No. U1936211 and 'the Fundamental Research Funds for the Central Universities', HUST: 2020JYCXJJ068.

REFERENCES

[1] 2014. Permission-based method. http://infosec.bjtu.edu.cn/wangwei/?page_id=85/.
[2] 2017. MaMaDroid. https://bitbucket.org/gianluca_students/mamadroid_code/.
[3] 2018. Cyber attacks on Android devices on the rise. 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. 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] 2020. 44 Must-Know Malware Statistics to Take Seriously in 2020. https://https://legaljobsite.net/malware-statistics/.
[6] 2020. Dissecting DEFENSOR: a stealthy Android banking malware. https://medium.com/axdb/%EF%B8%8F-dissecting-defensor-a-stealthy-android-banking-malware-6610b0468256.
[7] 2020. Insidious Android malware gives up all malicious features but one to gain stealth. https://www.welivesecurity.com/2020/05/22/insidious-android-malware-gives-up-all-malicious-features-but-one-gain-stealth/.
[8] 2020. Stealthy new Android malware poses as ad blocker, serves up ads instead. https://blog.malwarebytes.com/android/2019/11/stealthy-new-android-malware-poses-as-ad-blocker-serves-up-ads-instead/.
[9] 2021. APK Protect - Provide Android APK Encryption and Protection. https://sourceforge.net/projects/apkprotect/.
[10] 2021. AppCritique Mobile Application Security Testing. https://www.boozallen.com/expertise/products/appcritique.html.
[11] 2021. Because Sharing is Caring. https://virusshare.com/.
[12] 2021. A Free Android App Market. http://www.appchina.com.
[13] 2021. HomDroid. https://github.com/CGCL-codes/HomDroid.
[14] 2021. SandDroid - An automatic Android application analysis system. http://sanddroid.xjtu.edu.cn/.
[15] 2021. scikit-learn. https://scikit-learn.org/.
[16] 2021. VirusTotal - free online virus, malware and URL scanner. https://www.virustotal.com/.
[17] 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)*. 86–103.

[18] 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)*. 210–221. https://doi.org/10.1145/3274694.3274744

[19] 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)*. 468–471. https://doi.org/10.1145/2901739.2903508

[20] 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 2014 Annual Network and Distributed System Security Symposium (NDSS'14)*. 1–15.

[21] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. Pscout: Analyzing the android permission specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12)*. 217–228. https://doi.org/10.1145/2382196.2382222

[22] 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)*. 426–436. https://doi.org/10.1109/ICSE.2015.61

[23] Vladimir Batagelj and Andrej Mrvar. 2001. A subquadratic triad census algorithm for large sparse networks with small maximum degree. *Social Networks* 23, 3 (2001), 237–243. https://doi.org/10.1016/S0378-8733(01)00035-1

[24] Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment* 2008, 10 (2008), P10008.

[25] 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)*. 659–674.

[26] Xiao Chen, Chaoran Li, Derui Wang, Sheng Wen, Jun Zhang, Surya Nepal, Yang Xiang, and Kui Ren. 2019. Android HIV: A study of repackaging malware for evading machine-learning detection. *IEEE Transactions on Information Forensics and Security* 15 (2019), 987–1001. https://doi.org/10.1109/TIFS.2019.2932228

[27] Morteza Dehghani, Kate Johnson, Joe Hoover, Eyal Sagi, Justin Garten, Niki Jitendra Parmar, Stephen Vaisey, Rumen Iliev, and Jesse Graham. 2016. Purity homophily in social networks. *Journal of Experimental Psychology General* 145, 3 (2016), 1–10. https://doi.org/doi/10.1037/xge0000139

[28] Anthony Desnos et al. 2011. Androguard. https://github.com/androguard/androguard.

[29] Ming Fan, Jun Liu, Wei Wang, Haifei Li, Zhenzhou Tian, and Ting Liu. 2017. Dapasa: Detecting android piggybacked apps through sensitive subgraph analysis. *IEEE Transactions on Information Forensics and Security* 12, 8 (2017), 1772–1785. https://doi.org/10.1109/TIFS.2017.2687880

[30] Karimi Fariba, Génois Mathieu, Wagner Claudia, Singer Philipp, and Strohmaier Markus. 2018. Homophily influences ranking of minorities in social networks. *Scientific Reports* 8, 1 (2018), 1–12.

[31] 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)*. 576–587. https://doi.org/10.1145/2635868.2635869

[32] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. 2013. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security (WAIS'13)*. 45–54. https://doi.org/10.1145/2517312.2517315

[33] Benjamin Golub and Matthew O. Jackson. 2012. How homophily affects the speed of learning and best-response dynamics. *The Quarterly Journal of Economics* 127, 3 (2012), 1287–1338.

[34] Liangyi Gong, Zhenhua Li, Feng Qian, Zifan Zhang, and Yunhao Liu. 2020. Experiences of landing machine learning onto market-scale mobile malware detection. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys'20)*. 1–14. https://doi.org/10.1145/3342195.3387530

[35] Michael C. Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. 2012. Systematic detection of capability leaks in stock android smartphones. In *Proceedings of the 2012 Annual Symposium on Network and Distributed System Security (NDSS'12)*. 1–15.

[36] Wenjun Hu, Jing Tao, Xiaobo Ma, Wenyu Zhou, Shuang Zhao, and Ting Han. 2014. Migdroid: Detecting app-repackaging android malware via method invocation graph. In *Proceedings of the 23th International Conference on Computer Communication and Networks (ICCCN'14)*. 1–7. https://doi.org/10.1109/ICCCN.2014.6911805

[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)*. 318–329. https://doi.org/10.1145/2931037.2931044

[38] Ying-Dar Lin, Yuan-Cheng Lai, Chien-Hung Chen, and Hao-Chuan Tsai. 2013. Identifying android malicious repackaged applications by thread-grained system call sequences. *Computers & Security* 39, part B (2013), 340–350. https://doi.org/10.1016/j.cose.2013.08.010

[39] Xueqing Liu, Yue Leng, Wei Yang, Wenyu Wang, Chengxiang Zhai, and Tao Xie. 2018. A large-scale empirical study on android runtime-permission rationale messages. In *Proceedings of the 2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC'18)*. 137–146. https://doi.org/10.1109/VLHCC.2018.8506574

[40] Xueqing Liu, Yue Leng, Wei Yang, Chengxiang Zhai, and Tao Xie. 2018. Mining android app descriptions for permission requirements recommendation. In *Proceedings of the 2018 IEEE International Requirements Engineering Conference (RE'18)*. 147–158. https://doi.org/10.1109/RE.2018.00024

[41] 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)*. 112–123. https://doi.org/10.1145/3274694.3274731

[42] 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 2017 Annual Symposium on Network and Distributed System Security (NDSS'17)*. 1–16.

[43] Miller Mcpherson and LovinSmith Lynn. 2001. Birds of a feather: Homophily in social networks. *Annual Review of Sociology* 27, 1 (2001), 415–444. https://doi.org/10.1146/annurev.soc.27.1.415

[44] Mark EJ Newman. 2006. Finding community structure in networks using the eigenvectors of matrices. *Physical Review E* 74, 3 (2006), 036104. https://doi.org/10.1103/PhysRevE.74.036104

[45] Mark EJ Newman and Michelle Girvan. 2004. Finding and evaluating community structure in networks. *Physical Review E* 69, 2 (2004), 026113. https://doi.org/10.1103/PhysRevE.69.026113

[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 2012 ACM Conference on Computer and Communications Security (CCS'12)*. 241–252. https://doi.org/10.1145/2382196.2382224

[47] Yu Qu, Xiaohong Guan, Qinghua Zheng, Ting Liu, Lidan Wang, Yuqiao Hou, and Zijiang Yang. 2015. Exploring community structure of software call graph and its applications in class cohesion measurement. *Journal of Systems and Software* 108 (2015), 193–210. https://doi.org/10.1016/j.jss.2015.06.015

[48] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. 2007. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E* 76, 3 (2007), 036106. https://doi.org/10.1103/PhysRevE.76.036106

[49] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. 2013. Catch me if you can: Evaluating android anti-malware against transformation attacks. *IEEE Transactions on Information Forensics and Security* 9, 1 (2013), 99–108. https://doi.org/10.1109/TIFS.2013.2290431

[50] Martin Rosvall and Carl T. Bergstrom. 2008. Maps of random walks on complex networks reveal community structure. *National Academy of Sciences* 105, 4 (2008), 1118–1123. https://doi.org/10.1073/pnas.0706851105

[51] 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)*. 13–22. https://doi.org/10.1145/2295136.2295141

[52] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. 2016. Avclass: A tool for massive malware labeling. In *Proceedings of the 2016 International Symposium on Research in Attacks, Intrusions, and Defenses (RAID'16)*. 230–253.

[53] Andrew T. Fiore and Judith S. Donath. 2005. Homophily in online dating: When do you like someone like yourself?. In *Proceedings of the 2005 Extended Abstracts on Human Factors in Computing Systems (EAHFCS'05)*. 1371–1374. https://doi.org/10.1145/1056808.1056919

[54] Ke Tian, Danfeng Yao, Barbara G. Ryder, Gang Tan, and Guojun Peng. 2017. Detection of repackaged android malware with code-heterogeneity features. *IEEE Transactions on Dependable and Secure Computing* 17, 1 (2017), 64–77. https://doi.org/10.1109/TDSC.2017.2745575

[55] 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 Transactions on Information Forensics and Security* 9, 11 (2014), 1869–1882. https://doi.org/10.1109/TIFS.2014.2353996

[56] Lei Xue, Xiapu Luo, Le Yu, Shuai Wang, and Dinghao Wu. 2017. Adaptive unpacking of android apps. In *Proceedings of the 2017 IEEE/ACM International Conference on Software Engineering (ICSE'17)*. 358–369. https://doi.org/10.1109/ICSE.2017.40

[57] Wei Yang, Deguang Kong, Tao Xie, and Carl A. Gunter. 2017. Malware detection in adversarial settings: Exploiting feature evolutions and confusions in android apps. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC'17)*. 288–302. https://doi.org/10.1145/3134600.3134642

[58] 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). 384–394. https://doi.org/10.1145/3180155.3180223

[59] 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)*. 303–313. https://doi.org/10.1109/ICSE.2015.50

[60] 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 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS'14)*. 1105–1116. https://doi.org/10.1145/2660267.2660359

[61] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. 2013. Fast, scalable detection of piggybacked mobile applications. In *Proceedings of the 3rd ACM Conference on Data and application security and privacy (CODASPY'13)*. 185–196. https://doi.org/10.1145/2435349.2435377

[62] 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 2012 Annual Symposium on Network and Distributed System Security (NDSS'12)*. 1–13.

[63] 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 Transactions on Software Engineering and Methodology* 30, 3 (2021), 1–32. https://doi.org/10.1145/3442588