2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) | 978-1-6654-5278-6/23/$31.00 ©2023 IEEE | DOI: 10.1109/SANER56733.2023.00065

RGDroid: Detecting Android Malware with Graph Convolutional Networks against Structural Attacks

Yakang Li†, Yikun Hu†, Yizhuo Wang, Yituo He, Haining Lu, Dawu Gu*
Shanghai Jiao Tong University
Shanghai, China
{liyakang_n, yikunh, mr.wang-yz, yituo_he, hnlu, dwgu}@sjtu.edu.cn

***Abstract*—The rapid growth of Android malware calls for anti-malware systems to detect malware automatically. Detecting malware effectively is a non-trivial problem due to the high overlap in behaviors between malware and benign apps. Most existing automated Android malware detection methods use statistic features extracted from apps or graphs generated from method calls to identify malware. However, the methods that only use statistic features lead to false positives due to ignoring program semantics. Existing graph-based approaches suffer scalability problems due to the heavy-weight program analysis and time-consuming graph matching. In addition, graph-based approaches could be evaded by modifying dependencies among method calls. As a result, crafted malicious apps resemble the benign ones.**

In this paper, we propose a novel deep learning-based detection system, named RGDroid, which is capable of detecting malware under graph structural attacks. It combines API information extracted from Android document and learns behavior features from function call graph by graph neural network. Specifically, to defend against graph adversarial attacks, RGDroid reduces the connectivity of different functional parts to mitigate the effect of structural modifications on the final graph embedding. To comprehensively evaluate the robustness of RGDroid, we implement four influential graph adversarial attacks to simulate current capabilities and knowledge of Android malware attackers. The attack success rate (ASR) of two state-of-the-art detection systems (i.e., MaMaDroid, MalScan) is above 70.0% while the ASR of RGDroid under the four graph attacks is below 6.1%.

***Index Terms*—malware, deep learning, adversarial attack**

I. Introduction

The Android platform has been exploited by malware developers due to its huge market share and open-source features. Android malware greatly endangers the privacy and data security of users, such as stealing user information and controlling the user's mobile remotely. The exponential growth of Android malware led to a strong focus on automated malware detection [1]–[8] to protect user data and privacy. In the meantime, malware developers also use various adversarial attacks to evade detection [9]–[13].

To detect Android malware, the main task is to identify malicious behaviors and resist potential adversarial attacks on detectors, which faces two challenges. On the one hand, malicious code usually occupies a small part of the entire app and malware show similar behaviors to benign apps. Existing methods [3], [5], [7], [8] based on program statistics usually use API frequency or opcode frequency, resulting in low detection accuracy due to ignoring program semantics. To increase accuracy, recent work proposes graph-based detection methods [14]–[16], which distill the program semantics into graph representations and further perform graph matching to identify malicious behaviors. However, graph-based methods suffer a scalability problem due to the heavy-weight program analysis and time-consuming graph matching. Recent studies [1], [2], [17] use lightweight analysis to generate graphs from the program and perform graph analysis. When benign apps show similar behaviors as malware, these methods may lead to false positives. On the other hand, the detection performance of graph-based methods could be degraded seriously under graph structural attacks [1], [9], [10], which modify the calling relationships between the methods in the malware. Such modification causes features extracted from crafted malware to resemble benign apps.

In this paper, we propose a robust malware detection system based on Graph Convolutional Networks (GCN), namely RGDroid. The goal of RGDroid is to detect malware under graph structural attacks. To extract semantic features of behavior, RGDroid constructs a relation graph of Android APIs according to Android document and converts each API entity into an embedding. Each API in the function call graph (FCG) generated from apps can be mapped to an embedding as the initial feature of the FCG's node. Then GCN model automatically learns the structure and semantic information of FCG by iteratively aggregating and propagating node information. The cost for each round of node feature propagation for each FCG input is $O(|V|+|E|)$ [1] when using a well-trained GCN model. That is much more scalable than general graph isomorphism algorithms whose general bound is in exponential time [18]. Therefore, RGDroid can scale to complex FCG.

To resist adversarial attacks, RGDroid reduces the connectivity of different functional parts based on the insight that the edges between different functional parts have a comparatively large impact on graph embedding. The state-of-the-art attack establishes a connection between the malicious part and the benign part to hide the malicious features by adding redundant edges, i.e., fake calling relationships [9]. RGDroid removes

† Yakang Li and Yikun Hu contributed equally to this paper.
* Dawu Gu is the corresponding author.

[1] $|V|$ is the number of nodes and $|E|$ is the number of edges.

2640-7574/23/$31.00 ©2023 IEEE
DOI 10.1109/SANER56733.2023.00065

those redundant edges by dividing the graph into subgraphs according to functionalities. Although the graph structures are changed, functions with more similar functionality are still "closer" to each other. Therefore, RGDroid divides the graph into subgraphs with different functions by community detection algorithm. Then it extracts features from subgraphs and ensembles them to make final prediction.

We evaluate RGDroid on a dataset of 26,463 apps including 13,363 malware and 13,100 benign apps spanning from 2013 to 2019. And we produce two experiments to evaluate the effectiveness and robustness against adversarial attacks respectively. Our first experiment focuses on the effectiveness of malware detection. Compare to two state-of-the-art detectors (i.e., MalScan [1], MaMaDroid [2]), RGDroid is able to detect Android malware with up to 96.1% f1-measure accuracy while the f1-measure of MaMadroid and MalScan are 91.6% and 93.5% respectively. The result shows that RGDroid has better performance in detecting malware. Our second experiment focuses on the robustness against adversarial attacks. We present four graph adversarial attacks (i.e., HRAT [9], ReWatt [19], GRABNEL [20], Gradient-based [21]) for the detectors considering Android malware attackers' current capabilities and knowledge. The attack success rate (ASR) of RGDroid under the four graph attacks is below 6.1% while the ASR of MaMadroid and MalScan is above 70%. The experimental result indicates that RGDroid is effective in detection and is robust against adversarial attacks.

In summary, The major contributions of this work include:

- We propose an Android malware classifier that achieves comparable detection performance when compared to MalScan [1] and MaMaDroid [2], two state-of-the-art methods. We combine API information extracted from Android document and graph neural network to learn behavior features.
- We propose an ensemble approach based on community detection to defend against adversarial attacks. To the best of our knowledge, this is the first work that considers FCG-based classifier defenses against adversarial attacks.
- We conduct an evaluation demonstrating that our method is effective for malware detection and is resilient against practical adversarial malware attacks. The results of extensive experiments show that our method can achieve a 96.1% f1 measure. And the attack success rate (ASR) of our method under the four graph attacks is below 6.1% while the ASR of another two state-of-the-art FCG-based systems (i.e., MaMaDroid, MalScan) is above 70.0%.

II. Preliminary

In this section, we present the attack problem statement faced by the graph based malware detection systems and introduce the basic knowledge of Graph Convolution Networks.

A. Adversarial Attack

1) Attacker capabilities: To ensure that the modified app is consistent with the original function, we consider four types of modification actions [9], namely adding edges, rewiring, adding nodes, and deleting nodes given an FCG extracted from an Android app. The details are shown in Figure 1.

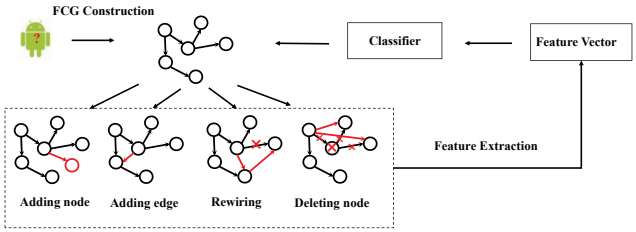

Fig. 1. Attack process

- **Constraints.** An attacker cannot modify nodes and edges in FCG arbitrarily. For example, an attacker cannot modify Android framework APIs used in app. Therefore, let $\mathbb{C}$ denote the set of modifiable nodes.
- **Adding node.** An attacker adds a new node v_{new} on the graph and select a node from $\mathbb{C}$ to invoke it.
- **Adding edge.** An attacker selects two nodes v_{beg}, v_{end} from $\mathbb{C}$ and builds an invocation relation from v_{beg} to v_{end}.
- **Rewiring.** An attacker first finds an edge which connects the caller $v_{beg} \in \mathbb{C}$ and the callee v_{end}, and then removes the edge from the graph. To maintain the connectivity of nodes in the removed edge, the attacker finds another intermediate node $v_{mid} \in \mathbb{C}$ and creates two new edges from v_{beg} to v_{mid} and from v_{mid} to v_{end}.
- **Deleting node.** An attacker selects a node $v_{tar} \in \mathbb{C}$ and removes it. To keep the functionality, the attacker first collects node sets v_{caller} and v_{callee} representing functions that call v_{tar} and called by v_{tar}, respectively. Then the attacker builds call relations from all nodes in v_{caller} to each node in v_{callee} after removing v_{tar}.

2) Attack process: An attacker misleads a detector by manipulating the FCG of a malicious app. In order to evaluate the robustness of our method, we consider both black-box and white-box attacks. In white-box attack scenario, the adversary has access to the dataset, feature space, and model parameters of target systems. In black-box attack scenario, the adversary can only interact with the classifier by querying it with an input graph and observing the model output. According to the feedback from the classifier, the attacker iteratively modifies the graph using four operations. Then the call relationships in malicious app can be modified through the sequence of modifications generated by the attack process. In this paper, we only consider the success of the attack in the feature space rather than problem space.

B. Graph Convolution Networks

Graph Neural Networks have been shown to be effective in graph representation learning. These models usually learn node representations by iteratively aggregating, transforming, and propagating node information. In this work, we adopt graph

convolutional networks (GCN) [22]. A graph convolutional layer in the GCN framework can be represented as

$$H^l = \delta(D^{-\frac{1}{2}}AD^{\frac{1}{2}}H^{l-1}W^l) \quad (1)$$

where $H^l \in \mathbb{R}^{N\times d_l}$ is the output of the l-th layer, W^l is the learnable parameters of l-th layer, d_l is the node feature dimension of l-th layer. A GCN model usually contains l graph convolutional layers. To perform graph classification, we apply a readout operation $pool(\cdot)$ to obtain a graph level embedding u_G.

$$\phi(G) = pool(H^l) \quad (2)$$

Then a multilayer perceptron (MLP) and softmax layer are then sequentially applied on the graph embedding to predict the label of the graph.

$$y = f(u_G) = softmax(MLP(u_G, \theta)) \quad (3)$$

III. METHODOLOGY

In this section, we first present an overview of RGDroid, and then describe the details of RGDroid.

A. Approach Overview

We formulate malware detection as a binary classification problem at graph representation. Figure 2 presents an overview of our approach, which consists of the following three main stages.

- **Graph-based Program Representation.** For each app, RGDroid distills the program into function call graph (FCG) and combines Android document into FCG. The enhanced FCG can be used to determine whether a program contains malicious behaviors.
- **Graph Partition.** After generating the graph representation, RGDroid then divides it into certain subgraphs by community detection to reduce the connectivity of different functional modules.
- **Learning and Detection.** With subgraphs generated by community detection, RGDroid adopts a graph convolutional network to learn graph embedding and then uses a single full-connected layer with sigmoid function as the classifier to decide whether each app contains malicious behaviors.

B. Graph-based Program Representation

1) API Embedding Generation: API information is one of the most important features since the applications usually invoke framework APIs to implement specific function. For example, malware usually invokes sensitive APIs that operate on sensitive data to perform malicious activities. In the program, the API is represented as signature information (i.e. package name, class name return type, method name, and parameter list). However, the signature needs to be transformed into a feature vector for subsequent algorithms usage. An intuitive method is to use a one-hot encoded vector as function attributes, which is very sparse and does not contain function semantic information. It can only provide little information for app behavior analysis.

For a better representation of the API information, RGDroid builds an API relation graph by collecting Android API documents and extracting entities such as APIs and permissions and relations between those entities. For example, the API document refers to another API with similar functionality in the functional description section of one API. Therefore, there is a relationship between these two API entities. After building the relation graph, RGDroid converts all the entities in the relation graph into an embedding representation using the algorithm TransE [23]. The generated API embedding can characterize the API similarity and the relation with other APIs. Finally, RGDroid obtains a API mapping from API signatures to API embeddings.

2) FCG Construction: To distill the program semantics as graph representation, RGDroid constructs FCG from Dalvik code by static analysis. For better semantic representation, RGDroid initializes the node attributes to the corresponding API embedding according to the API mapping. The enhanced FCG is represented as a directed, unweighted graph G = (V, E, X).

- $V = \{v_i | 1 \le i \le N\}$ denotes the set of functions, where N is the number of nodes.
- $E \subseteq V \times V$ denotes the set of function calls, where edge $(v_i, v_j \in E)$ indicates that there exists a function call from the caller function v_i to the callee function v_j.
- $X \subseteq \mathbb{R}^{N\times D}$ denotes the set of node attributes, where D represents the attribute dimension. Each D-dimensional vector is the API embedding corresponding to the node.

C. Graph Partition

After generating the enhanced function call graph, RGDroid divides the graph into subgraphs by performing community detection to resist adversarial attacks. Although the Graph Neural Network (GNN) can embed the graph into a feature vector by propagating and aggregating the information of neighboring nodes, GNN suffers graph structural attacks. As described in Section II-A, the calling relationship among method calls can be modified to cause the final embedding to be disturbed. The modifications are guided by the model gradient to maximally perturb the propagation process. The greater the functional difference between two nodes, the greater the value of edge gradient connecting them.

To mitigate the effect of structural modifications on the final graph embedding, RGDroid reduces the connectivity of different functional parts. An Android app is made up of certain specific modules and each module completes different functionality. The nodes in one module should be highly cohesive, and nodes in the different modules should be loosely coupled. Furthermore, a previous study [24] has demonstrated that a software call graph can be treated as a network with community structures. Therefore, RGDroid performs community detection to divide a FCG into a set of subgraphs in order to reduce interference between different functional parts. For example, the multi-level algorithms [25] start with every node belongs to a separate community and then iteratively move nodes between communities in an attempt to improve

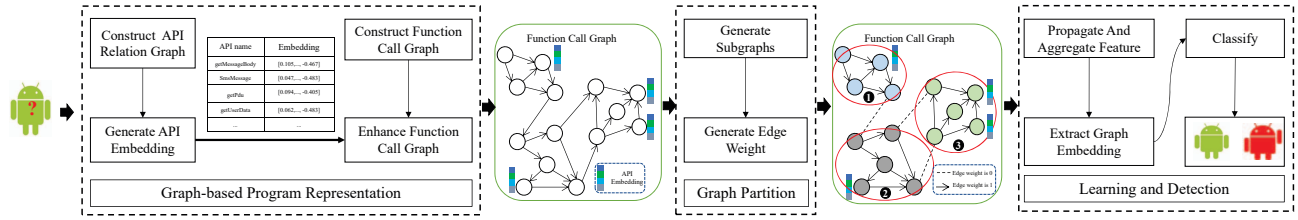

Fig. 2. System overview of RGDroid

the modularity score which is a measure of the quality of a partition of a graph into communities. The algorithm stops when it is no longer possible to increase the modularity score by merging communities into single vertices. The eigenvector algorithms [26] calculate the eigenvector of the modularity matrix for the largest positive eigenvalue and then separating vertices into two community based on the sign of the corresponding element in the eigenvector. The number of communities divided depends on the functionality distribution of the target under analysis. For example, a real malicious app (ccvimapgames, ccbikinihunt) can be partitioned into 19 functional partitions including device information acquisition part, network connection part and game function part. How to choose specific community detection algorithms is discussed in Section IV-A3

To conveniently represent the subgraphs obtained by community detection, RGDroid generates edge weights to represent the importance of calling relationships. Edges in the same subgraph are given a weight of 1 while edge weights connecting different subgraphs are 0. Through the above operations, RGDroid removes the edges between subgraphs to reduce information propagation in different functional parts. Further, to reduce the damage to functional integrity of removing edges, RGDroid preserves calls from arbitrary methods to the APIs.

D. Learning and Detection

After obtaining subgraphs of a FCG, RGDroid employs Graph Convolutional Network (GCN) to learn graph embedding from subgraphs and classifies the input sample. RGDroid uses the label of the entire graph instead of subgraphs as the supervision signal for training. The label of the whole graph comes from the specific app which is labeled by VirusTotal as 0 or 1 according to whether the app is confirmed as malicious.

1) Feature Learning: GCN can operate directly on graphs and leverage their structural information and node features to generate graph embeddings. Given a FCG, the GCN takes as input: an adjacency matrix $A \in \{0,1\}^{|V|\times|V|}$ representing the graph structure, a feature matrix $X \in \mathbb{R}^{N\times D}$ where each row is a vector representation of a node, a weight vector $\mathcal{E} \in \{0,1\}^{|E|}$ representing the edge weights described in Section III-C. Then the GCN model generates node embedding that incorporates topological structure and node features iteratively. A GCN model usually consists of L graph convolutional layers. At each layer, the features are aggregated to form the next layer's features using a propagation rule. As the FCG is direct graph, each hidden layer can be expressed as below:

$$h_j^{l+1} = \sigma(\sum_{j\in N_i} \frac{e_{ji}}{c_{ji}} h_j^{(l)} W^{(l)} + b^{(l)}) \tag{4}$$

where N_i is the set of neighbors of node i, σ is an activation function (e.g., ReLU), c_{ji} is the product of the square root of node degrees, $e_{ji} \in \mathcal{E}$ is the weight on the edge from node j to node i, $h_j^{(l)}$ is the feature of the node i of layer l, $W^{(l)}$ is the learnable parameters matrix of layer l. RGDroid obtains node feature matrix H^{l+1} after propagation. Further, RGDroid does not use the normalization term c_{ji} to minimize node degree impact as much as possible. Note that the initial feature matrix $H^0 = X$. After updating the node feature, RGDroid uses MaxPooling layer as ReadOut layer to calculate the max of every column in the node feature matrix to obtain the graph embedding.

$$u_l = max(H_l) \tag{5}$$

Then RGDroid concatenates the features of each layer as the final graph embedding u_G which is behavior level feature vector. The cost for each round of propagation is $O(|V|+|E|)$. $|V|$ is the number of nodes and $|E|$ is the number of edges. Therefore, feature extraction based GCN can scale to complex FCG.

2) Classification: Given extracted graph embedding, RGdroid adopts a multilayer perceptron (MLP) and sigmoid layer to predict the label probability of the graph $\hat{y}$ and uses Binary Cross Entropy loss as loss function.

$$\hat{y} = sigmoid(MLP(u_G)) \tag{6}$$

$$\mathcal{L} = ylog\hat{y} + (1-y)log(1-\hat{y}) \tag{7}$$

where $\mathcal{L}$ denotes the loss, y denotes the ground-truth label of G. In order to reduce information loss caused by dividing subgraphs and achieve better defense, RGDroid ensembles the model's predictions for each set of subgraphs generated by different community detection algorithms to make final predictions. The subgraphs obtained by different community detection algorithms contain various information since different algorithms divide subgraphs from different perspectives. The perturbation made by the attacker must be valid for multiple sets of subgraphs to evade successfully. Specifically, RGDroid obtains different edge weights by performing different community detection algorithms and uses the same GCN model

Algorithm 1: RGDroid: Learning and Detection

Stage I: Initialization

- Initialize node attribute X by API mapping
- Generate edge weight $C = \{\mathcal{E}_1, \mathcal{E}_2, ..., \mathcal{E}_m\}$ by m community detection algorithms
- Set learning rate λ
- Initialize the weights W of the parameters of model M.

Stage II: Training

- **Input:** Training data set $D = \{G_i, y_i\}_{i=1}^{n}$
- **Output:** Model M
- In each epoch:
  - Set loss $\mathcal{L} \leftarrow 0$
  - Sample a batch of D
  - **foreach** $\mathcal{E}$ **in** C **do**
    - Set the edge weight as $\mathcal{E}$
    - Calculate node feature using (4)
    - Get graph embedding according to (5)
    - Calculate loss $\mathcal{L}^{'}$ according to (6) and (7)
    - $\mathcal{L} \leftarrow \mathcal{L} + \mathcal{L}^{'}$

    end
  - Update the parameters W with learning rate λ

$$W \leftarrow W - \lambda \nabla \mathcal{L}$$

TABLE I
SUMMARY OF DATASETS USED IN OUR EXPERIMENTS

Dataset	Benign	Malware	Total
2013	1,852	1,985	3,837
2014	1,802	1,965	3,767
2015	1,768	1,830	3,616
2016	2,002	1,926	3,928
2017	1,876	1,818	3,694
2018	2,047	1,906	3,953
2019	1,735	1,933	3,668
Total	**13,100**	**13,363**	**26,463**

to extract the graph embedding. But it adopts different MLPs to classify the graph with different edge weights. Finally, it takes the maximum value of different classification results to get the final prediction. During training, the whole model is iteratively updated by minimizing the loss function 7. The detailed implementation of RGDroid is given in Algorithm 1.

IV. EVALUATION

In this section, we conduct experiments to evaluate the accuracy and robustness of RGDroid and answer the following research questions:

- RQ1: How effective is RGDroid compared to other Android malware detection systems?
- RQ2: How robust is RGDroid under graph structural attack compared to other methods?
- RQ3: What is the runtime overhead of RGDroid on detecting Android malware?

A. Evaluation Setup

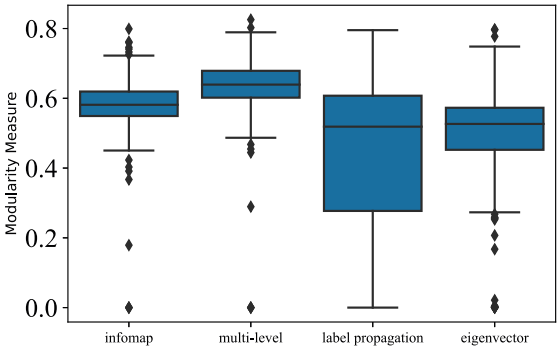

Fig. 3. The values of modularity of four different community detection algorithms

1) Data Collection: Dataset used to evaluate our method includes 26,463 Android apps. We crawled these apps from AndroZoo [27] which contains tens of millions of APK files, each of which has been detected by several different antivirus products in VirusTotal. Apps that are marked by more than 5 antivirus products are considered malicious and those that are not marked are considered benign. The dataset contains 13,363 malware and 13,100 benign apps and the time period of our datasets ranges from 2013 to December 2019. Table I lists the details of our datasets.

2) Dataset Partition: We split the data into three parts: train set, valid set, and test set. We sample 2,002 benign examples and 2,002 malware examples as the testing set stratified by year and the rest as the training set. We use five-fold cross-validations for all detectors. For each fold, we use the model that performs best on the valid set to predict the test set and take the average results as the final performance.

3) RGDroid Configuration: We describe the specific configuration used by RGDroid.

Partition Configuration. In Section III-C, RGDroid divides the function call graph through community detection algorithms. There are four widely used community detection algorithms (i.e., infomap [28], label propagation [29], multi level [25], and leading eigenvector [26]). To confirm their clustering performance, we sample 1000 apps, extract their FCGs and conduct community detection on them. Then we record modularity values which measure the strength of division of a graph into communities. Graphs with high modularity have dense connections between the nodes within communities but sparse connections between nodes in different communities. The result shows the average modularity value of communities generated by informap, label propagation, multi level, and leading eigenvector are 0.58, 0.45, 0.64 and 0.51, respectively. Due to the poor performance of label propagation, we use the other three algorithms. We also use Normalized Mutual Information (NMI) to measure the similarity of the community structures obtained by three community detection

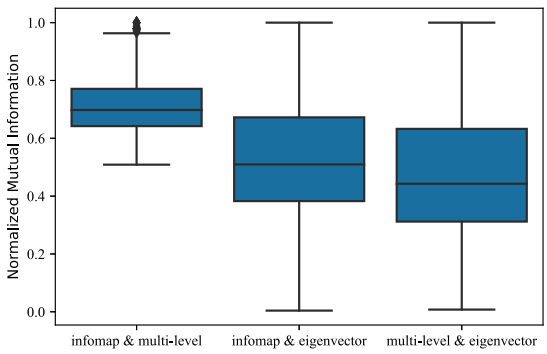

Fig. 4. The Normalized Mutual Information (NMI) of the communities generated by different community detection algorithms

algorithms. The larger the NMI, the more relevant the community structure is perfect correlation. As shown in Figure 4, the NMI between the community detection pair is 0.72, 0.55 and 0.48, respectively. It indicates that the subgraphs obtained by different community detection algorithms are relatively different. Therefore, RGDroid selects informap, multi level and leading eigenvector as community detection algorithms.

Training Configuration. We set the number of graph convolutional layers of model as 3 and hidden dim is 64. The GCN model is trained via Adam optimizer with the learning rate set to 0.001. Our model was trained on a server machine with a Nvidia GTX 3080Ti GPU and a 2.20GHZ CPU with 10 cores and 128G memory. We have released the source code of our approach at https://github.com/saneranonymous/RGDroid with the sha256 of apps which we used in our experiments.

B. RQ1: Detection Effectiveness

In this section, we evaluated the effectiveness of RGDroid in scenarios where no adversarial examples exist compared with the following methods.

- **MalScan [1]**. We implement MalScan according to the source code. Note that, RGDroid uses the API embedding to enhance the detection capabilities. For a fair comparison, we use API embedding to enhance MalScan. Specifically, we divide API embeddings into different API clusters according to their similarity and replace each sensitive API call used in MalScan's implementation with corresponding API cluster. we select the best among experimental results conducted by different classification models (i.e., 1-Nearest Neighbor (1NN), 3-Nearest Neighbor (3NN), Support Vector Machine (SVM), and Random Forest (RF)).
- **MaMaDroid [2]**. We implement MaMaDroid according to the description and source code in its paper. And We also adopt different classifiers (i.e., 1NN, 3NN, SVM, RF) and select the best result. We find that 1NN is able to maintain better effectiveness on detecting malware and select it as the best model.
- **GCN-based classifier**. We implement the following three versions of the GCN-based classifier. (i) Baseline GCN: It takes the original FCG as input and uses the node degree as the initial attribute. (ii) AGCN: It takes the FCG as input and uses generated node embedding as the initial node attribute. (iii) AGCN (Single): It conducts a single community detection algorithm to split the FCG into subgraphs as inputs. For all the above classifiers, we set the number of convolutional layers as 3, hidden dim is 64. And we concatenate the features of each layer and then use a full-connected layer and sigmoid layer to get a final prediction.

1) Evaluation Measures: We adopt widely used metrics to measure the effectiveness of our method. We report F-measure and accuracy for presenting the overall detection effectiveness of detectors. Moreover, we also report precision, recall, FNR and FPR to see how detectors perform on classifying both malicious and benign samples.

2) Result: As shown in Table II, we see that RGDroid can maintain a high f-measure and accuracy above 96%. The f-measure of RGDroid is 96.1% while MaMaDroid and Malscan can achieve 91.6% and 93.5% f-measure respectively. Although the recall of Malscan is 97.4% higher than RGDroid, the precision of Malscan is only 88.2%. The reason is that the Malscan only considers simple semantic information (i.e., centrality) which may lead to false positives when benign apps show similar behaviors as malware. As for MaMadroid, it only uses the package or family information of the method while RGDroid makes full use of methods and call information.

For GCN-based classifiers, the Baseline GCN achieves 90.4% F-measure and 91.3% recall which shows that it has a certain detection ability as it can learn the topology information from the FCG. Compared to Baseline GCN, GCN with API embedding (AGCN) greatly improves detection performance and the F-measure improves from 90.4% to 96.3%, since the function attribute is crucial for malware detection. The overall performance of RGDroid is slightly lower than AGCN. Because it divides the graph into multiple subgraphs and removes the edges between the subgraphs, resulting in some information loss. But the performance of RGDroid is slightly higher than the models with a single input. This indicates that combining the subgraphs generated by different community detection is beneficial to reduce information loss.

We also evaluate how well RGDroid on detecting malware by training and testing using samples that are developed in the same year. Figure 5 presents the detection results achieved by MalScan, MaMaDroid, and MalScan enhanced by API Embedding and RGDroid on each dataset, respectively. We see that for each dataset, RGDroid can maintain a high F-measure above 94%. Compared to other detectors, RGDroid has better performance.

Answer to RQ1: RGDroid consistently achieves high accuracy rates and F1 scores, with a consistent performance of over 95%. API embedding extracted from Android document significantly contributed to the achieved effectiveness of RGDroid.

TABLE II
EFFECTIVENESS COMPARISON OF DIFFERENT DETECTORS

Methods	F-measure	Accuracy	Precision	Recall	FPR	FNR
MaMaDroid	91.6%	88.2%	89.8%	93.9%	17.6%	6.0%
MalScan	93.5%	92.2%	88.2%	**97.4%**	12.9%	**2.5%**
Baseline GCN	90.4%	90.1%	89.4%	91.3%	11.1%	8.6%
AGCN	**96.3%**	**96.3%**	96.8%	95.7%	3.2%	4.2%
AGCN (Single)	95.9%	96.0%	**97.2%**	94.6%	**2.7%**	5.3%
RGDroid	96.1%	96.2%	96.8%	95.4%	3.1%	4.5%

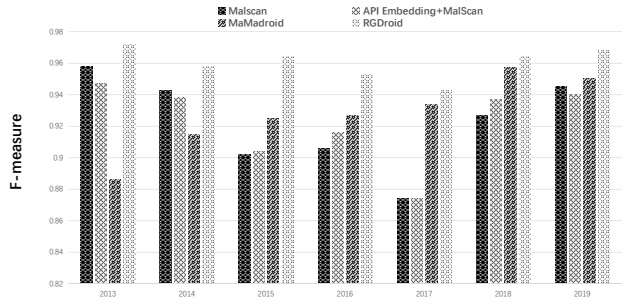

Fig. 5. F-measure of MalScan, MalScan enhanced by API Embedding, MaMadroid, RGDroid with datasets from the same year

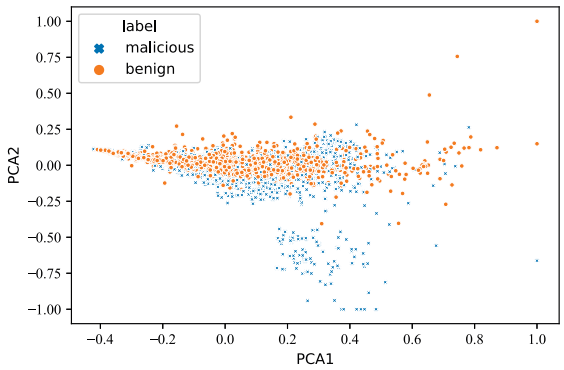

Fig. 6. Positions of benign vs malicious apps in the feature space of the first two components of the PCA.

3) Principal Component Analysis: We apply PCA to select the two most important PCA components of graph embedding. We plot the positions of the two components for benign and malicious samples. Figure 6 shows that benign and malware have relatively different distributions in feature space. The difference highlight that RGDroid can learn discriminative features to represent the behavior of malicious and benign samples.

4) Case Study: We analyze 74 benign apps mistakenly detected as malware by RGDroid. These apps usually use dangerous permissions and sensitive APIs to perform specific functions. However, some seemingly malicious behavior actually matches the behavior described by the apps. For example, the main function of an app is social media and video chats. It is able to access fine location and record audio, which is malicious behavior in other apps. Ignoring the feature declaimed in the app itself leads RGDroid to flag benign apps as malware mistakenly. We also analyze malware apps not detected by RGDroid. We find that 9% of the false negatives cannot be parsed correctly. The FCGs of these samples contain less than 25 nodes, which means RGDroid cannot properly build the function call graph due to dynamic loading and encryption. We sample 10 apps from 79 false negatives for further analysis. We find 8 apps are adware which does not perform clearly malicious activities. And the other two apps did not perform malicious behavior and may have been mislabeled as malware.

C. RQ2: Robustness against Adversarial Attack

In this section, we evaluate the robustness of RGDroid under the following graph adversarial attacks.

- **HRAT [9]** performs four types of graph modification operations and uses reinforcement learning (RL) to optimize the attack process. It is designed for MalScan and MaMaDroid. It does not consider the GNN-based model. Therefore, according to the attack framework described in the paper and the source code, we implement the attack against the GCN-based classifier. Specifically, in order to select optimal edges or nodes to conduct the modifications on the graph, we use the graph edge gradient obtained from the GCN model to guide selection. And the state in RL framework is the graph embedding extracted from the GCN model.
- **ReWatt [19]** is an black-box attack for the graph classification task. ReWatt uses a graph rewiring operation to perform the attack and utilizes deep reinforcement learning to learn the strategy to attack effectively attack. The attacker uses GCN to learn node and edge embeddings, which are used as input to Policy Networks to make decisions about the next action. We change it to the white-box attack and the attacker can directly use the target model to obtain node and edge embeddings. Moreover, we modified the rewiring operation to suit the malware scenario.
- **Gradient-based [21]** means a gradient-based attack which greedily adds or deletes edges based on the magnitude computed input gradient. In this attack, we only add edges since deleting edges may break app functionality.
- **GRABNEL [20]** is a novel Bayesian optimisation-based black-box attack method for graph classification models.

TABLE III
ASRs[a] OF FOUR GRAPH ATTACK TOWARDS SIX DIFFERENT DETECTORS

Algorithm	HRAT	ReWatt	GRABNEL	Gradient-based
MaMadroid	70.7%	-	-	-
Malscan	77.7%	-	-	-
Baseline GCN	54.2%	30.0%	15.1%	15.0%
AGCN	50.6%	35%	17.3%	14.7%
AGCN (Single)	17.4%	5.7%	3.0%	4.4%
RGDroid	6.1%	2.3%	1.0%	1.2%

[a]Lower ASR means better defense against adversarial attacks.

We implemented it completely according to the source code.

For comparison robustness with other FCG-based methods, we also implemented HRAT attack on MaMadroid and Malscan. Since the other three attacks are proposed for GNN-based models, we do not consider these attacks on MaMadroid and Malscan. For Malscan, we get the entire source code of HRAT attack and use the same configuration as HRAT attack paper. But for MaMadroid, the code is not provided. We implemente HRAT attack on MaMadroid as described in its paper.

1) Evaluation Measures: To evaluate the effectiveness of the attacks, we use the attack success rates as our evaluation metric, which are defined as follows.

$$ASR = N_s/N$$

where N is the number of malware correctly predicted by the classifier, N_s is the number of malware that can successfully deceive the classifier after the attacker modifies its FCG. Since modifying misclassified malware makes no sense, we only use the samples that RGDroid predicted correctly in Experiment IV-B. Considering the cost of the attacker's attack, we set the maximum number of modifications as 500 for all the above attacks, which is the same as the HRAT [9] setting. Here, we mainly consider the defense effectiveness of the feature space. For attacks in the real world, not all modified FCGs can be repackaged due to the anti-repackage protection [30], and not all repackaged apps can run successfully. Therefore, the attack success rate shown in our experiments is higher than the actual attack success rate.

2) Result: From Table III, the ASR of RGdroid is lower than other methods, which indicates that RGdroid is more resilient against the graph structural attack compared to the other methods.

We can see that the performance of MaMadroid [2] and Malscan [1] drop significantly under HRAT [9] attack. For MaMadroid and Malscan, the ASR under HRAT attack is 70.7% and 77.7% respectively. For Malscan, the reason is that the centrality analysis which Malscan use to characterize the apps' malicious behaviors contains simple semantic information and can be easily changed. For example, when an attacker adds one node to the graph, the number of nodes increases, and degree centrality decreases. When an attacker adds one edge or removes one edge between one node and another node, the degree centrality will change. For MaMadroid, the function call probability can be changed by attackers. When inserting one edge from state i to state j, the function call probability between i and j increases and vice versa for deleting one edge. When adding or removing nodes i from graph, corresponding state calls probability can also be changed.

RGDroid can enhance security compared to the original GCN classifier and make attackers harder to evade detection. The ASR has been greatly reduced, especially in the scenarios of HRAT attack and ReWatt attack. The ASR decreased from 50.6% to 6.1% under HRAT attack and decreased from 35.0% to 2.3% under ReWatt attack. For Gradient-based attack, its ASR decreased from 14.7% to 1.2% compared to the ASR of AGCN. GRABNEL attack only achieves 1.0% ASR because it is a black-box attack and has not enough information about the model. And our defense can make the ASR of GRABNEL decrease from 17.3% to 1.0% because we remove most added useless edges. Comparing the ASR of Baseline GCN and GCN with API Embedding (AGCN) under the Rewiring attack, the ASR of AGCN is 35% which is higher than Baseline GCN. The reason is that the node attribute of AGCN contains more information than the node attribute of GCN. Rewiring which connects a node to its original target node through an unrelated intermediate node makes node features of AGCN contain more noise according to the aggregation scheme. When using multiple graphs as input, the ASR is reduced by about half for the all four attacks. The results demonstrate that graphs generated by different communities are more difficult to attack. Only the edge inserted by the attacker is not detected and eliminated by the all three communities, can the attacker successful evade detection.

Among these attacks, the HRAT attack is the most effective strategy, since they use four types of graph modifications to attack through reinforcement learning framework to optimize the structural attack process. Compared with the other three attacks, HRAT can achieve higher ASR on all classifiers. But, for the strongest attack HRAT, RGDroid still achieves good defensive performance.

Answer to RQ2: RGDroid is more resilient to adversarial attacks compared to MaMadroid and MalScan.

3) Attack Analysis: We analyze how an attacker affects our feature extraction process. We consider the four modification operations mentioned in the section II-A.

- For adding node $v_{new} \in \mathbb{C}$ and its caller v_{beg}, we can see that adding a user defined function can not impact the feature of node v_{beg} according to (4). Because the initial feature of v_{new} is a vector of all zeros and will not change in subsequent updates. Moreover, RGDroid ignores the normalization term c_{ji}. Although adding node changes the degree of v_{new}, the final feature can not be changed.
- When an attacker deletes one node $v_{tar} \in \mathbb{C}$, the callers and callees of v_{tar} will lost. But to preserve the functionality, those callers invoke the callees of v_{tar} by integrating the code of v_{tar}. Therefore, the callees information can be propagated to the caller directly and the caller still aggregates valid information.

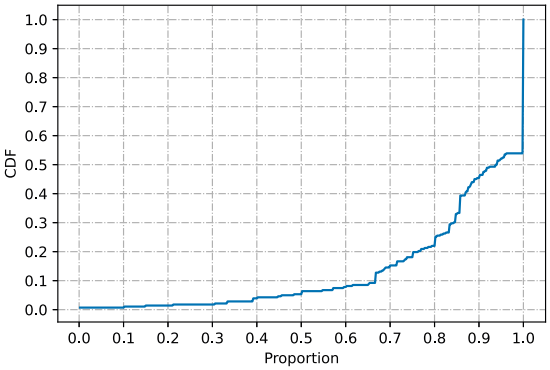

Fig. 7. The Cumulative Distribution Function (CDF) of the proportion of edges connecting different communities

- As for adding edge between v_{beg} and v_{end}, it can be the main operation that affects the aggregation of node features. We analyze the impact of community detection on adversarial attacks. From an attacker's perspective, an effective modification tends to modify the edge or node that can make the feature perturb to the greatest extent. An attacker usually uses gradients to select the most influential edges or nodes for modification. We first generated 200 adversarial samples using the gradient-based graph attack [21]. As discovered by analysis, most of the added edges connect different communities. We counted the proportion of edges connecting different communities to the total number of added edges. The cumulative distribution function (CDF) of proportion is given in Figure 7. Nearly 80% of the added edges are between communities. These changes in the graph structure can affect the node aggregation thus the representation of the entire graph. RGDroid can mitigate the effect on the final graph embedding by removing these edges.
- As for rewiring, it can be split into two steps. An attacker first removes an edge between v_{beg} and v_{tar} and add a new edge from intermediate node v_{mid} to v_{tar}, and from v_{beg} to v_{mid}. Similar to adding edges, rewiring operation uses the gradient to find the edge that maximizes the loss. Therefore, RGDroid is also effective against rewiring.

4) Case Study: To better illustrate the capability of RGDroid, we analyzed an interesting case using a real-world sample. Specifically, Figure 8 shows the malicious part of the FCG extracted from a malware with package name *ccvimapgames.ccbikinihunt* which steals device information (IMEI, IMSI, etc). The green nodes and the edges between them represent the path of the malware to obtain device information. The *AdsConnect.startConnect()* call *DeviceInfo.getInstance()* and the *DeviceInfo.getInstance()* call *TelephonyManager.getDeviceId() and TelephonyManager.getSubscriberId()* to get device information. The blue nodes and the edges between them show *AdsConnect.startConnect()* call *AdsConnect.parseHttpConfig()* to

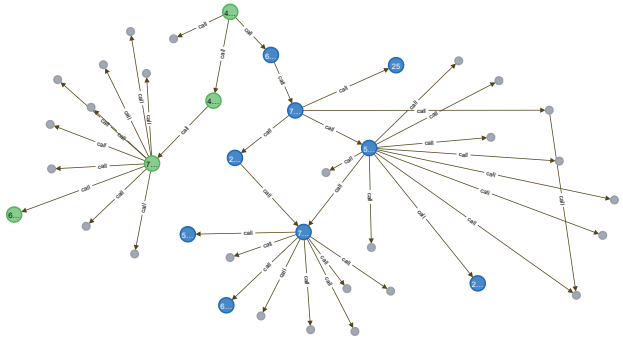

Fig. 8. The malicious part of a real malware (ccvimapgames.ccbikinihunt)

parse and configure http parameters and call *DefaultHttpClient.execute* to send device information to the address assigned by malware developer. RGDroid recognizes this app as malware according to the functionality of the API and the calling relationship between methods. To deceive RGDroid, an attacker adds a fake call from *AdsConnect.parseHttpConfig()* to *analytics.b.a()*. Then the attacker adds dozens of calls from *analytics.b.a()* to benign part *vimapgames.bikinihunt*. The malicious features can be diluted by connecting to the benign part. There is only one edge between *AdsConnect.parseHttpConfig()* and *analytics.b.a()*. However, these two methods are closely connected to the nodes in their respective communities. Therefore, RGDroid can remove the fake call between these two methods by community detection, which eliminates the effect of benign part on the malicious features.

D. RQ3: Runtime Overhead

In this phase, we evaluate the runtime overhead of RGDroid. As aforementioned, RGDroid is mainly composed of three parts to complete the detection which are Graph-based Program Representation, Graph Partition, Learning and Detection. We introduce the corresponding overhead in RGDroid.

Graph-based Program Representation: RGDroid first distills the program semantics of an app into a function call graph by static analysis. It needs to take about 19.7 seconds to complete the static analysis on average. This step is the most time-consuming phase of all the steps in RGDroid.

Graph Partition: After generating the call graph, we then perform three community detection algorithms to split the entire graph into multiple subgraphs. The average runtime overhead is 1.1s.

Learning and Detection We first learn the model parameters using the training data. The training phase takes less than 1.5 hours. After training, we are able to obtain the trained model to extract features and classify them. Given the subgraphs, feature extraction and classification are performed within the same periods. This phase consumes the least runtime overhead, it only requires about 0.0079s seconds to distinguish an input as either benign or malicious in GPU-enable and 0.01s in CPU. We also compare the runtime overhead of RGDroid with MaMadroid, and MalScan. MaMadroid takes

1.36 seconds on average to complete the feature extraction and classification and Malscan takes 0.131s on average. Compared to these methods, RGDroid which is based on GCN does not add much runtime overhead. In other words, RGDroid gains accuracy and robustness with less cost.

Answer to RQ3: RGDroid achieves a more efficient malware detection than MaMaDroid and MalScan.

V. Related Work

Automated Android Malware Detection. Many proposed approaches on Android malware detection rely on syntax features [3], [8], [11], [13], [31]–[35]. Arp *et al.* [3] extract broad features, such as permissions, filtered intents and API calls, and embed them in a vector to classify malware. However, they only use frequency information and ignore the program semantics. In addition, the method can be evaded by attacks [36]. Aafer *et al.* [5] conduct a thorough analysis to extract certain API calls used by malware and perform classification based on API frequency. The detection performance may be affected due to its use of specific calls.

To complete more effective Android malware detection, there are many graph-based approaches proposed [1], [2], [4], [14]–[17], [37]–[40]. Zou *et al.* [17] combines the high accuracy of traditional graph-based methods with the high scalability of social-network-analysis to detect malware. Wu *et al.* [37] design a new technique to discover the most suspicious part of covert malware by analyzing the homophily of a call graph. Cai *et al.* [38] propose the concept of enhanced function call graphs and develop a GCN-based algorithm to obtain vector representations of E-FCGs. Wu *et al.* [41] proposed a GCN-based algorithm and weighted mechanism to detect malware and find malicious nodes implied in the Android application function call graph. Gao *et al.* [4] map apps and Android APIs into a large heterogeneous graph and solve the node classification task based on the Graph Convolutional Network (GCN). However, these methods do not consider the robustness against the graph adversarial attack. Zhang *et al.* [39] propose APIGraph to slow down the classifier aging. The API embedding obtained in APIGraph is only used for clustering, and similar APIs are used as a class to enhance the sustainability of classifiers under evolved malware samples. However, these classifiers do not fully utilize the semantic information contained in the embedding. RGDroid is able to use the API embedding as node features to enhance the GCN-based classifier in order to take full advantage of API embeddings.

Adversarial Example Defense. Chen *et al.* [42] consider different importance of the features associated with their contributions to the classification problem as well as their manipulation costs, and present a novel feature selection method to make the classifier harder to be evaded. Li *et al.* [12] employ a similarity constraint to squeeze the room for adversarial examples and propose a new VAE (variational autoencoder) to detect malware. Li *et al.* [11] propose a new mixture of attacks by combining multiple attack methods and studying the usefulness of ensemble for both the defender and the attacker in the context of adversarial malware detection. But these methods only consider the detectors which rest on a set of syntax-based features (i.e., permissions, filtered intents, API calls, and new instances) extracted from the Android apps. Each app is usually represented by a binary feature vector. Therefore, these defense mechanisms can not be applied to the graph-based methods which extract features from the graph. Different from the existing works, in this paper, We consider defenses against graph structural attacks in Android malware detection.

VI. Limitations

FCG Extraction. In this paper, our static analysis is implemented by leveraging FlowDroid [43]. To reduce the runtime overhead caused by our constructed call graph, we can conduct a low-cost program analysis by leveraging Androguard which is a context- and flow-insensitive analysis. Our FCG is constructed based on the Dalvik code. Thus our approach would miss the malicious behaviors implemented in native code. However, other analysis frameworks [44] can help us address this limitation by constructing the FCG of the native code. We plan to use advanced program analysis to generate a suitable call graph to achieve the balance between the efficiency and effectiveness on detecting malware.

Obfuscations. Similar to any static analysis approach, our approach is vulnerable to dynamic loading and encryption. An app can loads a library into memory at runtime. If a malicious payload is hidden in such a library, our methods can not detect it effectively. As for encryption, packers can protect apps by using encryption techniques to hide the actual Dex code. We can use some unpacker tools such as PackerGrind [45] to recover the actual Dex files. Then the call graph can be extracted from actual Dex files.

VII. Conclusion

In this paper, we propose a novel Android malware detection system, RGDroid, which can accurately detect Android malware and defend against adversarial attacks. We combine API information and structural information to detect malware effectively. To defend against adversarial attacks, we reduce the connectivity of different functional parts to minimize adversarial perturbations. The promising experiments demonstrate our model's advantage in accuracy and robustness.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable feedback to improve our manuscript. This work is partially supported by the National Key Research and Development Program of China (No. 2021YFB3101402) and Shanghai Pujiang Program (No. 22PJ1405700). We especially thank Ant Group for the support of this research within the SJTU-Ant Security Research Centre.

REFERENCES

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

[2] L. Onwuzurike, E. Mariconti, P. Andriotis, E. D. Cristofaro, G. Ross, and G. Stringhini, "Mamadroid: Detecting android malware by building markov chains of behavioral models (extended version)," *ACM Trans. Priv. Secur.*, vol. 22, no. 2, apr 2019. [Online]. Available: https://doi.org/10.1145/3313391

[3] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of android malware in your pocket," in *NDSS*, 2014.

[4] H. Gao, S. Cheng, and W. Zhang, "Gdroid: Android malware detection and classification with graph convolutional network," *Comput. Secur.*, vol. 106, p. 102264, 2021.

[5] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android," in *SecureComm*, 2013.

[6] S. Hou, A. Saas, L. Chen, and Y. Ye, "Deep4maldroid: A deep learning framework for android malware detection based on linux kernel system call graphs," *2016 IEEE/WIC/ACM International Conference on Web Intelligence Workshops (WIW)*, pp. 104–111, 2016.

[7] A. Narayanan, Y. Liu, L. Chen, and J. Liu, "Adaptive and scalable android malware detection through online learning," *2016 International Joint Conference on Neural Networks (IJCNN)*, pp. 2484–2491, 2016.

[8] T. Kim, B. Kang, M. Rho, S. Sezer, and E. G. Im, "A multimodal deep learning method for android malware detection using various features," *IEEE Transactions on Information Forensics and Security*, vol. 14, pp. 773–788, 2019.

[9] K. Zhao, H. Zhou, Y. Zhu, X. Zhan, K. Zhou, J. Li, L. Yu, W. Yuan, and X. Luo, "Structural attack against graph based android malware detection," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 3218–3235. [Online]. Available: https://doi.org/10.1145/3460120.3485387

[10] X. Chen, C. Li, D. Wang, S. Wen, J. Zhang, S. Nepal, Y. Xiang, and K. Ren, "Android hiv: A study of repackaging malware for evading machine-learning detection," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 987–1001, 2020.

[11] D. Li and Q. Li, "Adversarial deep ensemble: Evasion attacks and defenses for malware detection," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 3886–3900, 2020.

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

[13] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. Mcdaniel, "Adversarial examples for malware detection," in *ESORICS*, 2017.

[14] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual api dependency graphs," in *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, 2014, pp. 1105–1116.

[15] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of android malware through static analysis," in *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, 2014, pp. 576–587.

[16] M. Fan, J. Liu, X. Luo, K. Chen, T. Chen, Z. Tian, X. Zhang, Q. Zheng, and T. Liu, "Frequent subgraph based familial classification of android malware," in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2016, pp. 24–35.

[17] D. Zou, Y. Wu, S. Yang, A. Chauhan, W. Yang, J. Zhong, S. Dou, and H. Jin, "Intdroid: Android malware detection based on api intimacy analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, pp. 39:1–39:32, 2021.

[18] L. Babai and E. M. Luks, "Canonical labeling of graphs," *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, 1983.

[19] Y. Ma, S. Wang, T. Derr, L. Wu, and J. Tang, "Graph adversarial attack via rewiring," *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, 2021.

[20] X. Wan, H. Kenlay, B. Ru, A. Blaas, M. A. Osborne, and X. Dong, "Adversarial attacks on graph classification via bayesian optimisation," *ArXiv*, vol. abs/2111.02842, 2021.

[21] D. Zügner, A. Akbarnejad, and S. Günnemann, "Adversarial attacks on neural networks for graph data," *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018.

[22] T. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *ArXiv*, vol. abs/1609.02907, 2017.

[23] A. Bordes, N. Usunier, A. García-Durán, J. Weston, and O. Yakhnenko, "Translating embeddings for modeling multi-relational data," in *NIPS*, 2013.

[24] Y. Qu, X. Guan, Q. Zheng, T. Liu, L. Wang, Y. Hou, and Z. J. Yang, "Exploring community structure of software call graph and its applications in class cohesion measurement," *J. Syst. Softw.*, vol. 108, pp. 193–210, 2015.

[25] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, p. 10008, 2008.

[26] M. E. J. Newman, "Finding community structure in networks using the eigenvectors of matrices." *Physical review. E, Statistical, nonlinear, and soft matter physics*, vol. 74 3 Pt 2, p. 036104, 2006.

[27] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 468–471. [Online]. Available: http://doi.acm.org/10.1145/2901739.2903508

[28] M. Rosvall and C. T. Bergstrom, "Maps of random walks on complex networks reveal community structure," *Proceedings of the National Academy of Sciences*, vol. 105, pp. 1118 – 1123, 2008.

[29] U. N. Raghavan, R. Albert, and S. R. T. Kumara, "Near linear time algorithm to detect community structures in large-scale networks." *Physical review. E, Statistical, nonlinear, and soft matter physics*, vol. 76 3 Pt 2, p. 036106, 2007.

[30] X. Zhan, L. Fan, S. Chen, F. Wu, T. Liu, X. Luo, and Y. Liu, "Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications," *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 1695–1707, 2021.

[31] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Using probabilistic generative models for ranking risks of android apps," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 241–252.

[32] W. Wang, X. Wang, D. Feng, J. Liu, Z. Han, and X. Zhang, "Exploring permission-induced risk in android applications for malicious application detection," *IEEE Transactions on Information Forensics and Security*, vol. 9, no. 11, pp. 1869–1882, 2014.

[33] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *ArXiv*, vol. abs/1801.01681, 2018.

[34] J. M. S. Miguel, M. Kline, R. A. Hallman, S. M. Slayback, A. Rogers, and S. S. F. Chang, "Aggregated machine learning on indicators of compromise in android devices," *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[35] K. Xu, Y. Li, R. H. Deng, K. Chen, and J. Xu, "Droidevolver: Self-evolving android malware detection system," *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 47–62, 2019.

[36] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel, "Adversarial examples for malware detection," in *European symposium on research in computer security*. Springer, 2017, pp. 62–79.

[37] Y. Wu, D. Zou, W. Yang, X. Li, and H. Jin, "Homdroid: detecting android covert malware by social-network homophily analysis," *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021.

[38] M. Cai, Y. Jiang, C. Gao, H. Li, and W. Yuan, "Learning features from enhanced function call graphs for android malware detection," *Neurocomputing*, vol. 423, pp. 301–307, 2021.

[39] X. Zhang, Y. Zhang, M. Zhong, D. Ding, Y. Cao, Y. Zhang, M. Zhang, and M. Yang, *Enhancing State-of-the-Art Classifiers with API Semantics to Detect Evolved Android Malware*. New York, NY, USA: Association for Computing Machinery, 2020, p. 757–770. [Online]. Available: https://doi.org/10.1145/3372297.3417291

[40] S. Hou, Y. Fan, Y. Zhang, Y. Ye, J. Lei, W. Wan, J. Wang, Q. Xiong, and F. Shao, "αcyber: Enhancing robustness of android malware detection system against adversarial attacks on heterogeneous graph based model," *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, 2019.

[41] Z. Ma, H. Ge, Z. Wang, Y. Liu, and X. Liu, "Droidetec: Android malware detection and malicious code localization through deep learning," *ArXiv*, vol. abs/2002.03594, 2020.

[42] L. Chen, S. Hou, and Y. Ye, "Securedroid: Enhancing security of machine learning-based detection against adversarial android malware attacks," *Proceedings of the 33rd Annual Computer Security Applications Conference*, 2017.

[43] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. Mcdaniel, "Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[44] J. Samhi, J. Gao, N. Daoudi, P. Graux, H. Hoyez, X. Sun, K. Allix, T. F. Bissyand'e, and J. Klein, "Jucify: A step towards android code unification for enhanced static analysis," *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pp. 1232–1244, 2022.

[45] L. Xue, H. Zhou, X. Luo, L. Yu, D. Wu, Y. Zhou, and X. Ma, "Packergrind: An adaptive unpacking system for android apps," *IEEE Transactions on Software Engineering*, vol. 48, pp. 551–570, 2022.