Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

TC 11 Briefing Papers

GDroid: Android malware detection and classification with graph convolutional network

Han Gao, Shaoyin Cheng, Weiming Zhang**

CAS Key Laboratory of Electromagnetic Space Information, School of Information Science and Technology, University of Science and Technology of China, China

ARTICLE INFO

Article history:
Received 29 April 2020
Revised 25 February 2021
Accepted 25 March 2021
Available online 5 April 2021

Keywords:
Android malware
Malware detection
Malware familial classification
API Embedding
Graph neural network

ABSTRACT

The dramatic increase in the number of malware poses a serious challenge to the Android platform and makes it difficult for malware analysis. In this paper, we propose a novel approach for Android malware detection and familial classification based on the Graph Convolutional Network (GCN). The general idea is to map apps and Android APIs into a large heterogeneous graph, converting the original problem into a node classification task. We build the "App-API" and "API-API" edges based on the invocation relationship and the API usage patterns, respectively. The heterogeneous graph is then fed into the GCN model, iteratively generating node embeddings that incorporate topological structure and node features. Eventually, the unlabeled apps are classified by their final embeddings. To our knowledge, this paper is the first study to explore the application of graph neural network in the field of malware classification. We develop a prototype system named GDroid. Experiments show that GDroid can effectively detect 98.99% of Android malware with a low false positive rate of less than 1%, outperforming the existing approaches. It also achieves an average accuracy of almost 97% in the malware familial classification task with surpassing the baselines. Additionally, we cooperate with QI-ANXIN Technology Research Institute to evaluate its real-world impact, and GDroid also maintains satisfactory performance in real-world scenarios.

1. Introduction

As the most used mobile operating system, Android has always been the vital target of hackers. A recent security report shows that an average of 12,000 new mobile malware was captured per day (360, 2018). The emergence of massive malware poses a considerable challenge for malware mitigation.

Previous research indicates that the primary source of new malicious apps is the variants of knowns (Fan et al., 2018). To accelerate malware analysis, researchers divide malware into various families to assist analysis. The malicious apps belonging to the same family exhibit similar behaviors, even are variants of the same malware. For example, family `GinMaster` silently transmits confidential information to the remote server, and `DroidKungFu` allows the hacker to control the device remotely. Correctly classifying unseen malicious apps into their families is helpful for malware mitigation.

This paper presents a novel approach for Android malware detection and familial classification based on the graph neural network. Concretely, we first map apps and Android APIs into a large heterogeneous graph. Then we respectively utilize two

* Corresponding authors.
E-mail addresses: gh2018@mail.ustc.edu.cn (H. Gao), sycheng@ustc.edu.cn (S. Cheng), zhangwm@ustc.edu.cn (W. Zhang).
https://doi.org/10.1016/j.cose.2021.102264
0167-4048/

relationships, (1) the invocation relationship between apps and APIs and (2) the API usage patterns, to build "App-API" and "API-API" edges. The heterogeneous graph is subsequently fed into GCN (Kipf and Welling, 2016), a graph neural network model, to generate informative embeddings (i.e., high-dimensional numerical vectors) for nodes. Eventually, the unlabeled apps are classified by their final embeddings.

The existing work most similar to ours is HinDroid (Hou et al., 2017). They utilized the Heterogeneous Information Network (HIN) (Sun and Han, 2012) to model apps, APIs with their relationships. They modeled three relationships: (1) the invocation relationship between apps and APIs, (2) the relative positional relationship among APIs (i.e., whether APIs coexist in the same code block), and (3) the package and method name of APIs (i.e., whether APIs have the same packages or method names). Then they calculated the path-based similarity by meta-paths (Sun and Han, 2012) over apps and performed classification by the multi-kernel Support Vector Machines. From our perspective, both package name and method name are part of API so that the relationships can be simplified. Besides, their approach relies on the path-based similarity, while recent research indicates that it is unable to fully mine latent structure information of graph (Shi et al., 2019). In contrast to HinDroid, we only model two relationships, making the heterogeneous graph more simple. Moreover, our approach is based on the graph neural network, which can leverage topological structure and node features to generate informative embedding for each node.

We develop a prototype system named GDroid and conduct extensive experiments to evaluate its performance. For the malware detection task, the experimental dataset consists of two parts: The benign apps are collected from Google Play Store (GP) (Google, 2017) and the malicious apps are from Android Malware Dataset (AMD) (Wei et al., 2017). For the malware familial classification task, three malware datasets, (1) Android Malware Genome Project (AMGP) (Zhou and Jiang, 2012), (2) Drebin (DB) (Gascon et al., 2014) and (3) AMD (Wei et al., 2017), are used to construct a series of experimental datasets with the different number of families. We compare GDroid with the existing approaches in various metrics, including Accuracy, Precision, Recall, F-measure, False Positive Rate (FPR), and False Negative Rate (FNR)[1]. The comparison results demonstrate the superiority of GDroid for the two tasks.

The main contributions can be summarized as follows:

- We present a novel approach for malware detection and familial classification based on the GCN model. To our knowledge, this is the first study to explore the application of graph neural network in the field of malware classification.
- We propose an embedding-based approach to mine the API usage patterns. It allows our model to utilize the relevance among Android APIs.
- We develop a prototype system named GDroid. Experimental results show that it can effectively detect 98.99% of malware with low FPR (<1%) and FNR. In addition, it achieves an average accuracy of almost 97% in the malware familial classification task. The comparison results indicate that GDroid outperforms the existing approaches for the two tasks. GDroid also maintains satisfactory performance in real-world scenarios.

[1] FPR and FNR are only used in the malware detection task.

2. Related work

Our study is based on the recent progress of the following fields: malware detection and classification, word embedding and graph neural network.

2.1. Graph-based android malware detection

Low-level raw features such as bytecodes (Xu et al., 2018; Yuan et al., 2020; Zhang et al., 2019), opcodes (Canfora et al., 2016; Kim et al., 2019; McLaughlin et al., 2017; Pektas and Acarman, 2019), strings (Kim et al., 2019; Wang et al., 2018), permissions (Gascon et al., 2014; Kim et al., 2019; Li et al., 2018; Vinayakumar et al., 2018; Yerima and Sezer, 2019) and APIs (Allen et al., 2018; Gascon et al., 2014; Jerbi et al., 2020; Nix and Zhang, 2017; Zhang et al., 2018; Zhou et al., 2019) are shallow. They are susceptible to code obfuscation techniques. In contrast, it is recognized that high-level structured features have better robustness because the cost of modifying such features is relatively high. Thus some researchers tend to use the graph-based features, such as Control Flow Graph (CFG) (Narayanan et al., 2016; Xu et al., 2017), API Dependency Graph (ADG) (Zhang et al., 2014) and Function Call Graph (FCG) (Gascon et al., 2013; Hassen and Chan, 2017; Narayanan et al., 2016) to characterize the code block, function, or the entire program. They are internal to the programs (i.e., they can be constructed for each program), so we call them "intra-app" features. There are also "inter-app" features, implying that they exist among programs. HinDroid (Hou et al., 2017) used HIN to model apps, the related APIs and their relationships. The disadvantage of HinDroid is discussed in Section 1.

2.2. Malware familial classification

Recent studies for malware familial classification utilize multi-level features which are extracted by dynamic or static analysis. The dynamic analysis focuses on the runtime information of the program. Martín et al. (2018) modeled the runtime app behaviors with Markov chains. Cai et al. (2019) recorded the runtime API invocations and inter-component communication of intents. Alzaylaee et al. (2020) proposed a deep learning system to detect malicious Android apps through stateful input generation. In theory, dynamic analysis is more robust at the expense of more resource consumption. However, dynamic analysis is unable to cover all behaviors of the program because of the limitation of simulation time and trigger ways. The event-driven mechanism of Android also brings difficulties to simulation. In contrast, static analysis can achieve comprehensive coverage of the program in most cases without too much resource consumption. Zhang et al. (2019) used n-gram of multiple raw features to model apps. Fan et al. (2018) proposed a weighted security-relevant API call graph matching algorithm to characterize apps. Zhou et al. (2017) tried to

find the maximum isomorphic subgraph of each security-relevant API within each malware family to extract the familial features. Garcia et al. (2018) used sensitive APIs and the invocations in libraries to describe apps and performed malware detection and familial classification based on the machine learning algorithms. Mirzaei et al. (2019) proposed a characterization system for Android malware families based on ensembles of sensitive API calls extracted from aggregated call graphs of different families.

However, existing studies for malware familial classification are limited to the "intra-app" features. The "inter-app" information has not received enough attention. In this paper, we utilize the "inter-app" information to build a large heterogeneous graph for app classification.

2.3. *Word2vec*

Wording embedding is an unsupervised learning technique in the field of Natural Language Processing (NLP). Word2Vec (Mikolov et al., 2013a) is a widely used approach that can generate context-aware word embeddings. If two words have similar contexts, the corresponding embeddings will be near each other in the embedding space, and vice versa. For example, the word "Paris" is near the word "Tokyo", while far from the word "Apple". Word2Vec contains two models, Continuous Bag-of-Words (CBOW) and Skip-gram. The CBOW model predicts the current word from a window of surrounding context words, while the Skip-gram model uses the current word to predict the surrounding words. Empirically, the Skip-gram model performs better than CBOW for the infrequent words (Mikolov et al., 2013a). Since the security-relevant APIs are not invoked frequently, and the number of API invocations may vary significantly in the different methods, we opt for the Skip-gram model to perform API embedding.

2.4. *Graph neural network*

The Graph Convolutional Network proposed by (Kipf and Welling, 2016) is a semi-supervised classification model for graph-structured data. The basic idea is to update the node representations by propagating information among nodes. GCN shows strong ability of representation and performs well in the tasks of various fields, such as the NLP (Gao et al., 2019; Liu et al., 2018; Yao et al., 2018), recommender system (Wang et al., 2019; Wu et al., 2019; Ying et al., 2018), computer vision (Chen et al., 2018; 2019) and biomedicine (Mao et al., 2019; Zhang and Kabuka, 2018). To our knowledge, this paper is the first study to explore the application of graph neural network in malware classification.

3. Methodology

Fig. 1 illustrates the overall workflow of GDroid. Given a set of apps, both labeled and unlabeled, and there are the following steps: (1) Extracting the API co-occurrence feature from apps. (2) Conducting API embedding based on the API co-occurrence feature. (3) Mining the patterns of API usage by the distance metric among embeddings. (4) Mapping apps and APIs into a heterogeneous graph. The "App-to-API" edges are established by the invocation relationships, and the "API-to-API" edges are built by the API usage patterns. (5) Feeding the heterogeneous graph into the GCN model, iteratively generating node embeddings that aggregate neighboring information with node features. (6) Classifying unlabeled apps by their final embeddings.

3.1. *Feature extraction*

Empirically, programs achieve their functionalities by calling system APIs so that the invocations of Android APIs are directly related to apps' behaviors. Existing studies usually regard API as the binary feature (e.g., constructing feature vector by checking whether the app invokes the specific API while each API corresponds to a dimension). However, this practice treats APIs as independent of each other, ignoring the potential relevance among them. Here we take a more comprehensive approach. We build a heterogeneous graph to hold apps and APIs, using the invocation relationship between apps and APIs and utilizing the patterns of API usage to model the relevance among APIs.

The invocation relationship can be extracted through common static analysis, while the acquisition of API usage patterns is complicated. To this end, we propose an embedding-based approach based on the API co-occurrence feature. Specifically, we deem the APIs invoked by the same method as a co-occurrence. The API co-occurrence feature exists in API sequences, and each internal method corresponds to an API sequence. Fig. 2 shows an example. It is an internal method related to check the network connection. According to the smali code, three different APIs are invoked so that they consist of the co-occurrence feature of this method.

3.2. *Mining of API usage patterns*

To mine the API usage patterns, we first perform API embedding based on the API co-occurrence feature and then summarize APIs that have similar usages according to the distance metric among embeddings. Next, we first introduce the original Skip-gram model and then detail our embedding-based approach.

3.2.1. *The Skip-Gram model*

The Skip-gram model (Mikolov et al., 2013b) can generate context-aware word embeddings. It uses a fixed-size sliding window moving on the texts to generate training samples. Training sample exists in pair (x, y), where x is the input, and y is the label. The model uses a sliding window moving on the texts to generate training data. Fig. 3 illustrates this process. The word *sat* in the middle is treated as the input, and the other words are treated as the targets. As a result, four training samples are generated: (sat, the), (sat, cat), (sat, on) and (sat, a). The training objective is to adjust word embeddings so that they can be utilized to predict the surrounding words. Formally, given a word sequence $w_1, w_2, w_3, \ldots, w_T$, the model maximize the average log probability as

$$J(w) = \frac{1}{T}\sum_{t=1}^{T}\sum_{-d\le j\le d}\log P(w_{t+j}|w_t) \quad (1)$$

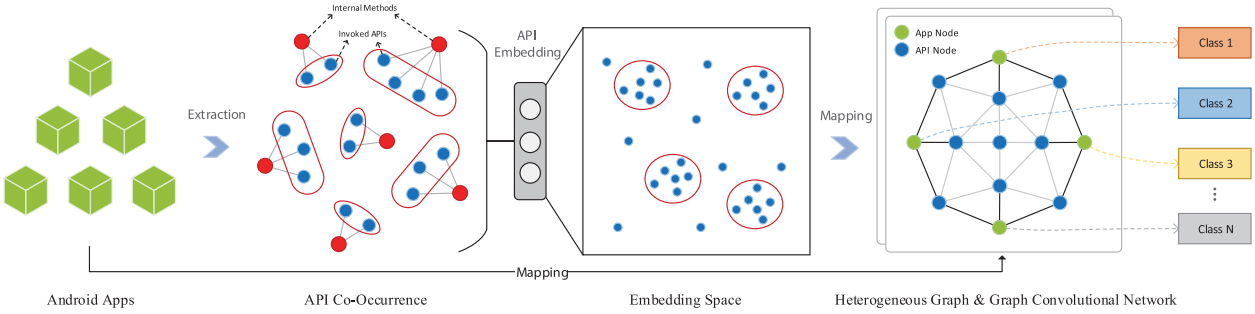

Fig. 1 – The overall workflow of GDroid. We first extract the API co-occurrence feature from apps. Then we perform API embedding, representing APIs as numerical vectors. The patterns of API usage are mined by the distance metric among embeddings. Next, we map apps and APIs into a heterogeneous graph and build "App-API" and "API-API" edges. Finally, we train the GCN model and classify unlabeled apps.

```
.method public static isConnected(Landroid/content/Context;)Z
......
   .line 74
......
   invoke-virtual {p0, v2}, Landroid/content/Context;
->getSystemService(Ljava/lang/String;)Ljava/lang/Object;                    (1)
......
   .line 77
......
   invoke-virtual {v0, v3}, Landroid/net/ConnectivityManager;
->getNetworkInfo(I)Landroid/net/NetworkInfo;                               (2)
......
   .line 78
......
   invoke-virtual {v1}, Landroid/net/NetworkInfo;->isConnected()Z           (3)
......
   .line 80
......
   invoke-virtual {v0, v4}, Landroid/net/ConnectivityManager;
->getNetworkInfo(I)Landroid/net/NetworkInfo;                               (2)

   .line 81
   invoke-virtual {v1}, Landroid/net/NetworkInfo;->isConnected()Z           (3)
......
.end method
```

Fig. 2 – An example of the API co-occurrence feature. We omit some statements except the API invocations due to space limits. According to the smali code, three distinct APIs are invoked to achieve the functionality of checking network connection. They consist of the co-occurrence feature of this method.

The sliding window size is $2d + 1$. $P(w_{t+j}|w_t)$ is defined as

$$P(w_O|w_I) = \frac{\exp(e_{w_O}^T e_{w_I})}{\sum_{i=1}^{W} \exp(e_i^T e_{w_I})} \tag{2}$$

where w_I and w_O are the input (w_t) and output (w_{t+j}) respectively, e_{w_I} and e_{w_O} are the corresponding embeddings, W is the size of vocabulary. However, this formulation is expensive to optimize because $\sum_{i=1}^{W} \exp(v_i^T v_{w_I})$ sums over all the words in vocabulary. In practice, negative sampling (Mikolov et al., 2013b) and hierarchical softmax (Mikolov et al., 2013a) are usually used to simplify the objective and accelerate the training process.

3.2.2. *API Embedding*

The API embedding model is adapted from the Skip-gram model. It is expected to encode Android APIs, preserving the context information to make the APIs with similar usages near each other in the high-dimensional space.

The original Skip-gram model uses a fixed-size window moving on the texts. However, this practice does not fit our task because most internal methods do not invoke too many APIs, so the API sequences are not long enough. In our experiments, we opt for an alternative way: Each API sequence is deemed as an API window, where we go through each API, taking the current API as input and targeting the other APIs. For example, given an API sequence $[api_1, api_2, \ldots, api_N]$. In the beginning, the first API api_1 is treated as the input, and

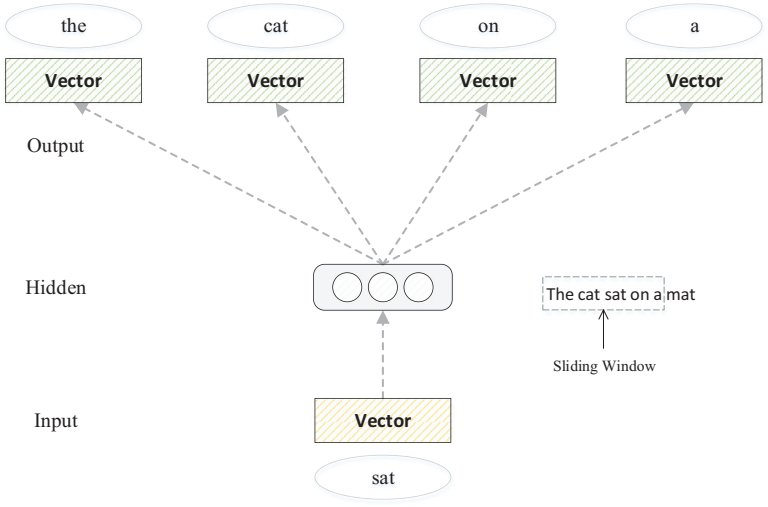

Fig. 3 – The Skip-gram model. There are five words in the window, which are *the*, *cat*, *sat*, *on* and *a*. The model uses the word *sat* in the middle to predict the surrounding four words.

the other APIs ($api_2, api_3, \ldots, api_N$) are treated as the targets. As a result, $N-1$ training samples are generated, which are (api_1, api_2), (api_1, api_3), $\ldots$, (api_1, api_N). Next, we treat the second API api_2 as input and targeting the other APIs (including api_1), and so on, for each API in the window. An API sequence that contains N APIs can generate $N \times (N-1)$ training samples.

The detailed procedure is presented in Algorithm 1. The training objective is similar to the Skip-gram model, which maximizes the probability of co-occurrence for the neighboring APIs.

3.2.3. *Distance-based mining*

The embedding process makes the APIs with similar usages cluster together in the high-dimensional space. Therefore, the API usage patterns can be obtained by the distance metric. Concretely, we present a set of APIs that need to be mined for usage patterns. Generally speaking, they are critical for modeling apps' behaviors, so we call them key APIs. For each of them, we calculate the cosine distance between it and other APIs (not only key APIs). Then we summarize the top-n nearest APIs. The summarized APIs have similar usages with the corresponding key API and can represent the patterns of API usage. Fig. 4 illustrates this process.

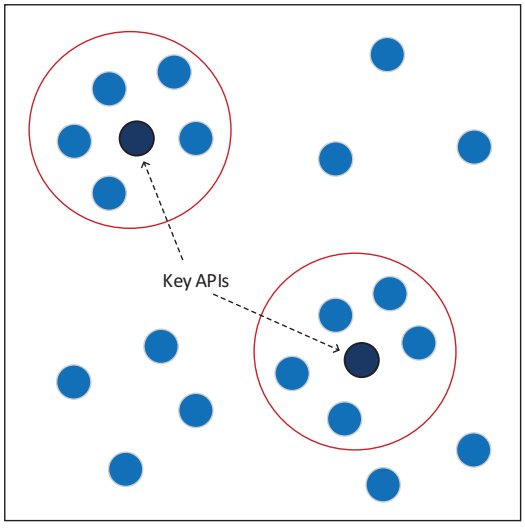

Fig. 4 – The key APIs with their top-5 nearest APIs in the embedding space.

Algorithm 1 Training Samples Generation For API Embedding.

Input: A set of apps $\mathcal{A}$ **Output:** A set of training samples for embedding

```
train_samples ← {}
for ∀app ∈ A do // for all apps
    internal_methods ← get_internal_methods(app)
    for ∀m ∈ internal_methods do // for all internal methods
        seq_API ← π(m) // extract the API invocation sequencein m
        pairs ← product(seq_API, seq_API) // generate trainingsamples
        train_samples ← train_samples ∪ pairs
return train_samples
```

The setting of key APIs is task-related. We use the sensitive APIs summarized by Au et al. (2012) as key APIs for the malware detection task. They are discovered based on the sensitive permission requirements and directly related to the malicious behaviors. As for the malware familial classification task, things have changed. According to our investigations, the divide of malware families is based not only on malicious behaviors but also on normal functionalities. For instance, the families `FakePlayer` is classified by the normal functionality of multimedia. These functionalities are achieved by other functional APIs, so the non-sensitive APIs also need to be

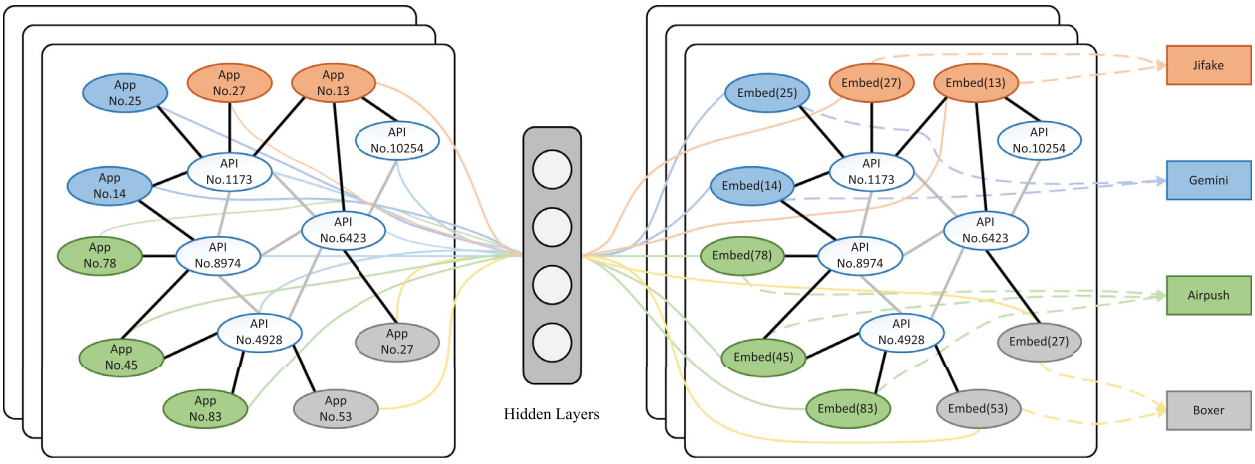

Fig. 5 – The schematic depiction of the GCN model in this paper. In the heterogeneous graph, the white nodes in the middle represent APIs, while the surrounding orange, blue, green and gray nodes represent apps. Each color of app nodes represents a malware family. The black straight lines between app nodes and API nodes are the "App-to-API" edges, while the straight gray lines among API nodes are the "API-to-API" edges. The heterogeneous graph is subsequently fed into GCN. After training, the right half is the graph with node embeddings. The unlabeled apps are classified into malware families (i.e., "Jifake", "Gemini", "Airpush", "Boxer") by their final embeddings. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

considered. Exactly, we mine the usage patterns for all APIs to comprehensively model the family behaviors. In the pre-experiments, we found it better than only considering sensitive APIs.

3.3. *Heterogeneous graph construction*

We construct a large heterogeneous graph to hold the apps, APIs with their relationships. As shown in the left half of Fig. 5, apps and APIs are mapped into the graph. We build two types of edges to model the relationships: "App-to-API" and "API-to-API" edges. Formally, the heterogeneous graph G is defined as

$$G = (V, E) \tag{3}$$

$$V = V_{app} \cup V_{api} \tag{4}$$

$$E = (V_{app} \times V_{api}) \cup (V_{api} \times V_{api}) \tag{5}$$

where V_{app} and V_{api} are sets of apps and APIs, respectively. G is a simple graph, and there is no more than one edge between every two nodes. The "App-to-API" edges are built by the invocation relationship. If the app invokes an API, an "App-to-API" edge will be established. The "API-API" edges are established by the patterns of API usage. For each key API, we build edges between it and each of its top-n nearest APIs.

3.4. *GCN-Based classification*

Our graph neural network model is adapted from GCN Kipf and Welling (2016), a semi-supervised learning model for graph-structured data. GCN embeds nodes with different features while taking the topological information into account. Each of the unlabeled nodes is represented by neighboring labeled nodes and itself, incorporating the topological structure and the node features. Note that GCN can capture information only about immediate neighbors with one layer. Stacking multiple layers can integrate information from larger neighbors. Fig. 5 shows the schematic depiction of the heterogeneous and the GCN model for this paper.

Given a graph $G = (V, E)$, where $V(|V| = n)$ and E are sets of nodes and edges, respectively. Let $X \in \mathbb{R}^{n \times d}$ be the feature matrix of V, where d is the dimension of the feature vectors, each row $x_v \in \mathbb{R}^d$ is the feature for node v. Let $A \in \mathbb{R}^{n \times n}$ be the adjacency matrix. If there is an edge between node v_i and node v_j, and then set A_{ij} to 1, otherwise $A_{ij} = 0$. According to the theory of GCN, each node should be self-loop. Hence the diagonal elements of A are set to 1. The adjacency matrix of graph added with self-connections is written as

$$\hat{A} = A + I_n \tag{6}$$

Every layer of GCN can be written as

$$H^{(l+1)} = f(H^{(l)}, \hat{A}) \tag{7}$$

where l ($l = 1, \ldots, L$) denotes the layer number and f is an activation function. The input layer $H^{(0)} = X$ and the output after the last layer $H^{(L)} = Z$. Then the GCN layer-wise propagation is

$$H^{(l+1)} = f(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}) \tag{8}$$

where $\hat{D}$ is the degree matrix($\hat{D}_{ii} = \sum_j \hat{A}_{ij}$), and $\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}$ is the normalized adjacency matrix.

In this paper, we build a two-layers GCN model. The second layer is not connected to an activation function. It is connected

to the softmax classifier to perform node classification.

$$Z = \text{softmax}(\tilde{A}\text{ReLU}(\tilde{A}XW^{(0)})W^{(1)}) \quad (9)$$

where $\tilde{A}$ is the normalized adjacency matrix. Recall that X is the feature matrix. In the following experiments, we set $X = I$, which means initial node features are one-hot vectors. In pre-experiments, we tried to use API embeddings as the feature of API nodes, and for each app, we used the mean of embeddings of its invoked APIs as its feature. However, this practice did not improve the results, so we chose the one-hot feature finally.

As for the loss function, we use the categorical cross-entropy error over all labeled apps

$$\mathcal{L} = -\sum_{a\in\mathcal{Y}_A}\sum_{f=1}^{F} Y_{af}\ln Z_{af} \quad (10)$$

where $\mathcal{Y}_A$ is the set of labeled apps and F is the dimension of the output vectors, which is equal to the number of classes. Y is the label indicator matrix. The weight matrix W is trained by Adam optimization method (Kingma and Ba, 2015).

4. Evaluation

In this section, we first introduce the dataset and experimental setup, and then we conduct a series of experiments to answer the following research questions:

- **RQ 1.** How does GDroid perform in the malware detection task? (Section 4.4.1)
- **RQ 2.** Does GDroid outperform the existing malware detection approaches? (Section 4.4.2)
- **RQ 3.** How does GDroid perform in the malware familial classification task? (Section 4.5.1)
- **RQ 4.** Does GDroid outperform the baselines of malware familial classification task? (Section 4.5.2)
- **RQ 5.** Do the API usage patterns contribute to the model performance compared with the straightforward practices? (Section 4.4.1)

4.1. *Dataset*

Three malware datasets: (1) AMGP (Zhou and Jiang, 2012), (2) DB (Gascon et al., 2014), (3) AMD (Wei et al., 2017) and a benign dataset GP collected from Google Play Store (Google, 2017) were used to construct experimental datasets. AMGP and DB have been widely used in the past five years. However, a recent study (Irolla and Dey, 2018) point that 49.35% of malware samples in DB have at least one repackaged app, and the only differences between the original malware sample and the repackaged versions are the resource files. The duplication issue is detrimental to build a robust model. Therefore, we also used the AMD dataset released in 2017, which could reflect the recent trend of Android malware to a certain extent. We randomly selected a part of apps from these datasets to compose the experimental datasets.

For the malware detection task, 1200 malicious apps from AMD and 2100 benign apps from GP were randomly selected to construct an experimental dataset (GP-AMD).

Table 1 – Summary statistics of the experimental datasets.

Dataset	# Apps	# APIs	# Classes
GP-AMD	3300	98,015	2
AMGP1	911	10,738	10
AMGP2	1070	11,005	20
DB1	1723	12,550	10
DB2	2364	14,436	20
DB3	2593	14,625	30
DB4	2738	15,367	40
AMD1	2000	20,232	10
AMD2	3654	27,758	20
AMD3	4095	28,591	30
AMD4	4298	29,247	40

The reasons for choosing malicious apps from AMD are as follows:

- The AMD dataset was collected from Google Play Store and released in 2017, consistent with our benign dataset. The alignment of the collection time and source is important.
- The AMD dataset contains the most malware families and samples. In other words, the malware diversity of AMD is greater, which can minimize the effect of malware samples belonging to the same family on the malware detection model.

The imbalance in the number of malicious and benign apps is derived from reality. We tried our best to simulate the situation when performing malware detection in the real world.

For the malware familial classification task, we noticed that the number of malware samples in different families was very unbalanced. Some families have hundreds of samples (e.g., `BaseBridge`), but some families have fewer than 10 samples (e.g., `Ackposts`). To mitigate this problem, we first sorted all malware families according to the number of samples and then selected families that contain sufficient samples to construct experimental datasets. The details of experimental datasets are listed in Table 1. For DB and AMD, we constructed four experimental datasets that contain 10, 20, 30, 40 families, respectively. For AMGP, we only constructed two datasets that contain 10 and 20 families due to the limited number of malware samples.

4.2. *Experimental setup*

4.2.1. *Environment*

We used an Ubuntu 16.04 machine with Intel Core i7-8700k, GeForce GTX 1080Ti and 32GB RAM to deploy GDroid. GPU is used to accelerate the training process of the neural network model. We implemented the proposed approaches using Python with several packages: Androguard (And, 2021), TensorFlow (Google, 2021), Scikit-learn (skl, 2021), and Matplotlib (Mat, 2021).

4.2.2. *Dataset splitting*

The number of samples in the training set accounted for 70% of the whole experimental dataset, while the validation set and the test set accounted for 15%, respectively.

Table 2 – Descriptions of the metrics.

Term	Abbr	Definition
True Positive	TP	# of apps correctly classified as malicious
True Negative	TN	# of apps correctly classified as benign
False Positive	FP	# of apps mistakenly classified as malicious
False Negative	FN	# of apps mistakenly classified as benign
Precision	P	TP/(TP+FP)
Recall	R	TP/(TP+FN)
F-measure	F	2TP/(2TP+FP+FN)
Accuracy	ACC	(TP+TN)/(TP+TN+FP+FN)×100%
False Positive Rate	FPR	FP/(TN+FP)×100%
False Negative Rate	FNR	FN/(TP+FN)×100%

4.2.3. *Hyper parameters*

The embedding size of API embeddings is 128. The stochastic gradient descent optimizer was used to train the API embeddings, and the learning rate is 1.0. The GCN model has two layers. The number of units of the hidden layer is 500. The Adam optimizer (Kingma and Ba, 2015) is used to train the GCN model, and the learning rate is 0.01.

4.2.4. *Metrics*

Table 2 lists the metrics used in this paper. For the malware detection task, we used Precision, Recall, F-measure, False Positive Rate, and False Negative Rate to evaluate GDroid. Ideal anti-virus software should have high precision, low FPR and FNR. We used Precision, Recall, F-measure to evaluate the model performance for the malware familial task, just like most existing studies.

4.3. *Mining of API usage patterns*

We first mined the API usage patterns. It is an essential step for subsequent experiments. Specifically, we extracted the API co-occurrence feature from apps via static analysis. Each internal method corresponds to an API invocation sequence. Then we performed API embedding based on the API co-occurrence feature. The number of training steps is set to a maximum of ten million until the loss function stabilizes. The embedding process is fast because of the shallow structure of the embedding network. It is a simple, fully connected neural network containing one hidden layer, so the back-propagation does not consume much time. With the acceleration of GPU, the embedding process took about several hours in our experiments.

Empirically, the APIs in the same class or package exhibit functional relevance. They are usually invoked together. An example is shown in Fig. 2. Three APIs are invoked to achieve the functionality of checking network connection while two of them belong to the same package (`Landroid/net`). Therefore, their embeddings should be near to each other in the embedding space. We exploited this observation to monitor the embedding process. Concretely, we randomly selected 10,000 APIs. While embedding, for each of them, we calculated the top-5 nearest APIs and counted the number of APIs that have the same class or package name as it. The results are shown in Fig. 6. With the iteration of embedding, the total number of APIs that met the requirements increased and finally tended to be stable. It indicates that the API embedding approach effectively captures the functional relevance among APIs.

After embedding, we summarized the usage patterns according to the cosine distance metric among embeddings. We list several security-relevant APIs with their top-5 nearest APIs in Table 3. As expected, the APIs nearest to the security-relevant API are usually used with it. For example, the APIs nearest to `Ljava/io/FileWriter→flush` are related to file operations. The APIs nearest to `Lorg/apache/http/client/HttpClient→execu-te` are all about network request. And the APIs nearest to `Ljava/security/MessageDigest→update` are related to message digest algorithm.

4.4. *Android malware detection*

Malware detection is essentially a binary classification problem, so there are two classes of app nodes in the heterogeneous graph, benign and malicious.

In our experiments, we used the sensitive APIs as key APIs. For each of them, we summarized n nearest APIs in the embedding space. We respectively set n to 5, 10, 20, extracting multi-level API usage patterns to build graphs in the experiments. There are two types of API nodes in the heterogeneous graph: sensitive APIs and some non-sensitive APIs with similar usages as sensitive APIs. We used the invocation relationship between apps and these APIs and the APIs' usage patterns to build the graph. The upper part of Table 4 lists the summary statistics of heterogeneous graphs under different n. In practice, we included all apps and APIs in the graph for convenience of implementation because the isolated nodes did not affect the experimental results. The density in Table 4 refers to the graph density. A simple undirected graph with p nodes and q edges has the graph density $\frac{q}{\frac{1}{2}p(p-1)}$. The graph density of the completed graph is 1.

In order to verify the contribution of the API usage patterns, we designed four Straightforward Practices (SP) for comparison:

- **SP 1.** Only using the invocation relationship between apps and sensitive APIs to construct graph.
- **SP 2.** Only using the invocation relationship between apps and all APIs to construct the graph.

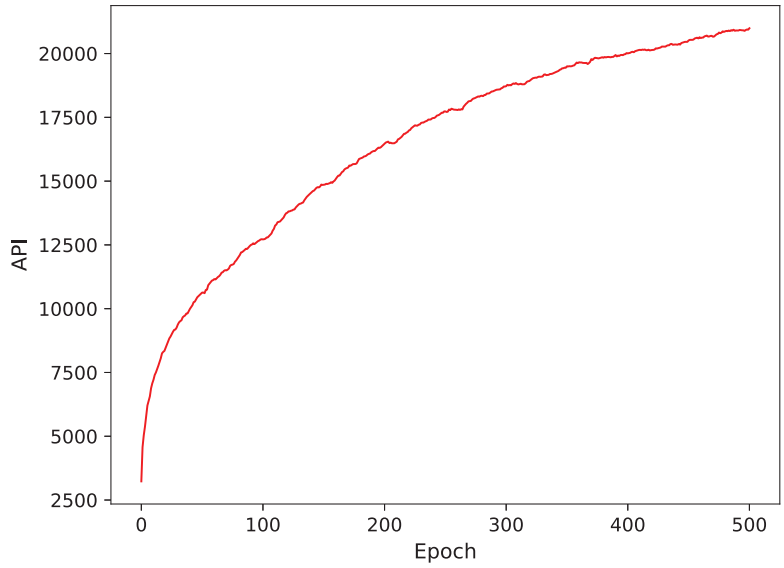

Fig. 6 – With the iteration of embedding, the number of APIs with the same class or package name as their corresponding APIs is increasing.

Table 3 – arest APIs in the embedding space.

Ljava/io/File→mkdir	Ljava/io/FileWriter→flush
Ljava/lang/Exception→toString	Ljava/io/FileWriter→append
Ljava/io/FileOutputStream→write	Ljava/io/FileWriter→init
Ljava/io/File→init	Ljava/io/FileWriter→close
Lorg/apache/http/client/HttpClient→execute	Landroid/util/Log→getStackTraceString
Ljava/text/NumberFormat→format	Ljava/io/FileWriter→init
Ljava/security/MessageDigest→update	**Landroid/location/Criteria→init**
Ljava/security/MessageDigest→digest	Landroid/location/Criteria→setAccuracy
Ljava/math/BigInteger→init	Landroid/location/LocationManager→getBestProvider
Ljava/security/MessageDigest→getInstance	Landroid/location/Criteria→setCostAllowed
Ljava/security/MessageDigest→reset	Landroid/location/Location→init
Ljava/math/BigInteger→toString	Landroid/location/Criteria→setBearingRequired
Ljava/lang/reflect/Method→getModifiers	**Lorg/apache/http/client/HttpClient→execute**
Ljava/lang/reflect/Method→getParameterTypes	Lorg/apache/http/StatusLine→getStatusCode
Ljava/lang/reflect/Modifier→isStatic	Lorg/apache/http/HttpEntity→getContent
Ljava/lang/reflect/Constructor→getParameterTypes	Lorg/apache/http/HttpResponse→getEntity
Ljava/lang/Class→getDeclaredMethods	Lorg/apache/http/HttpResponse→getStatusLine
Ljava/lang/Class→getDeclaredConstructors	Lorg/apache/http/message/BasicNameValuePair→init

Table 4 – Summary statistics of the heterogeneous graphs for the malware detection experiments.

Graph	# Nodes	# Edges	Density
GDroid ($n = 5$)	101,315	1,623,980	3.16E-04
GDroid ($n = 10$)	101,315	1,641,100	3.20E-04
GDroid ($n = 20$)	101,315	1,675,506	3.26E-04
SP 1	101,315	160,6,871	3.13E-04
SP 2	101,315	11,159,909	2.17E-03
SP 3	101,315	1,704,671	3.32E-04
SP 4	101,315	14,444,939	2.81E-03

- **SP 3.** Using the invocation relationship between apps and sensitive APIs and the co-occurrence relationship among sensitive APIs to construct the graph.
- **SP 4.** Using the invocation relationship between apps and all APIs and the co-occurrence relationship among all APIs to construct the graph.

Co-occurrence refers to whether APIs are invoked by the same method (i.e., co-occurrence in a method). The lower part of Table 4 lists the summary statistics of heterogeneous graphs constructed by the straightforward practices. As shown in Table 4, the usage pattern-based practices, SP 1 and SP 2 have relatively few edges. The SP 2 and SP 4 have much more edges, which implies the training time is extended.

4.4.1. *Results*

The experimental results are shown in Table 5. When setting n to 10, GDroid achieved the best performance without too much

Table 5 – The Performance of GDroid in Android Malware Detection.

Scheme	ACC	P	R	F	FPR	FNR	Time
GDroid ($n = 5$)	98.59	0.978	0.983	0.981	1.27	1.67	3.29
GDroid ($n = 10$)	**98.99**	**0.989**	**0.983**	**0.986**	**0.63**	**1.67**	**3.40**
GDroid ($n = 20$)	98.38	0.973	0.983	0.978	1.59	1.67	3.42
SP 1	97.37	0.966	0.961	0.964	1.90	3.89	3.45
SP 2	97.58	0.972	0.961	0.966	1.59	3.89	18.36
SP 3	97.37	0.961	0.967	0.964	2.22	3.33	3.56
SP 4	98.59	0.978	0.983	0.981	1.27	1.67	25.75

ACC, FPR and FNR are presented in the form of percentages.

Table 6 – Comparison of GDroid with other malware detection approaches.

Method	ACC	P	R	F	FPR	FNR
McLaughlin et al. (2017)	95.73	0.941	0.941	0.941	3.34	5.41
Onwuzurike et al. (2019)	93.74	0.952	0.872	0.910	2.54	12.64
Zhang et al. (2018)	96.67	0.967	0.967	0.967	1.90	3.26
GDroid	**98.99**	**0.989**	**0.983**	**0.986**	**0.63**	**1.67**

time consumption. It detected 98.99% of malware with a low FPR (<1%) and FNR.

As for SP 1 and SP 3, the training time consumption of them is similar to GDroid, while the performance is not as good as GDroid. For the SP 2 and SP 4, the GCN model required much more time for training without performance improvement. It indicates that the API usage patterns contribute to the model performance.

Answer to RQ 1 and RQ 5: GDroid can effectively detect 98.99% of malware with low FPR (<1%) and FNR. The API usage patterns contribute to the model performance, which help the model reach higher performance with less time for training.

4.4.2. Comparison

We used three representative approaches as baselines for comparison. They achieved good performance on their datasets at the time. McLaughlin et al. (2017) extracted the opcode sequences and utilized the Convolutional Neural Network (CNN) to automatically find appropriate features and perform classification. Onwuzurike et al. (2019) modeled the API sequences as Markov chains, using the probabilities of state transitioning to construct the feature vectors for apps. They also utilized machine learning algorithms for classification. Zhang et al. (2018) extracted the API invocations and used the n-gram of package names as the features to construct the feature vectors. Then they utilized the learning-based algorithms to perform binary classification. We emphasize that (Onwuzurike et al., 2019) and (McLaughlin et al., 2017) are published in the reputable venues and have influence in malware classification. Zhang et al. (2018) is our previous research, so credibility is not a concern. In addition, open source is an important consideration because it guarantees the consistency of the method and avoids potential biased. The authors of Onwuzurike et al. (2019) and McLaughlin et al. (2017) have open sourced their code so that we can easily reproduce them on our dataset (i.e., GP-AMD).

Table 7 – The Performance of GDroid in Android Malware Familial Classification.

Dataset	ACC	Precision	Recall	F-measure
AMGP1	97.10	0.976	0.970	0.973
AMGP2	96.88	0.974	0.931	0.939
DB1	98.84	0.990	0.983	0.986
DB2	99.15	0.995	0.984	0.989
DB3	96.92	0.969	0.975	0.970
DB4	95.85	0.957	0.932	0.935
AMD1	99.00	0.990	0.990	0.990
AMD2	95.45	0.958	0.956	0.956
AMD3	95.28	0.947	0.951	0.945
AMD4	95.34	0.933	0.919	0.921
Avg.	**96.98**	**0.969**	**0.959**	**0.960**

Table 6 lists the comparison results. As expected, GDroid surpassed the baselines in all metrics. It demonstrates the effectiveness of our approaches.

Answer to RQ 2: GDroid outperforms the existing approaches in the malware detection task. It achieves the highest accuracy and the lowest FPR and FNR.

4.5. *Android malware familial classification*

Malware familial classification is a multi-class classification problem so that there are multiple classes of app nodes.

According to our investigations, the divide of malware families is based not only on malicious behaviors but also on normal functionalities. Therefore, we consider all APIs to model the family behaviors. According to the pre-experiments, we set n to 5 to achieve the best performance of GDroid.

4.5.1. Results

Table 7 lists the experimental results. GDroid achieved an average accuracy of almost 97% on various datasets. As the number of families increasing, the accuracy remained above 95%. It indicates that our approach effectively captures the differences in API usage patterns among different malware families.

Answer to RQ 3: GDroid can classify malicious apps into their families with an average accuracy of almost 97%. As the number of families involved in experiments increases, GDroid maintains high performance.

4.5.2. Comparison

We select three machine learning-based approaches as baselines for comparison. Each of them represents a typical practice for malware classification. The first approach is based on the frequency of API invocation. It ignores the relevance among APIs. The second approach is based on the n-gram of opcodes. It uses the frequency of the trigram feature to characterize apps. The third approach draws inspiration from computer vision. It first transforms the `dex` files into gray-scale images (one byte corresponds to one pixel). Then it resizes the images to the same size (224×224), converting the malware classification into an image classification task. The idea of performing malware classification based on the gray-scale image classification is popular in recent years (Liu et al., 2020; Vasan et al., 2020; Yuan et al., 2020). Machine learning algorithms are

Table 8 – Comparison of GDroid with other malware familial classification approaches on average performance.

Method	ACC	Precision	Recall	F-measure
GBDT+API	91.28	0.926	0.891	0.904
GBDT+Opcode	90.70	0.919	0.884	0.897
CNN+Gray Image	82.39	0.832	0.785	0.802
GDroid	**96.98**	**0.969**	**0.959**	**0.960**

For each method, the results presented in the above table are the average results on ten familial classification datasets.

Table 9 – The performance of GDroid for familial classification on real-world data.

# Families	ACC	Precision	Recall	F-measure
10	97.40	0.972	0.975	0.973
20	95.55	0.948	0.961	0.952
30	95.13	0.912	0.916	0.913
40	92.07	0.887	0.856	0.865
Avg.	**95.04**	**0.930**	**0.927**	**0.926**

applied for classification. For the first and second approaches, we applied Principal Component Analysis (PCA) to reduce the dimensionality of feature vectors, and then we respectively used Random Forests (RF), Support Vector Machines (SVM), and Gradient Boosting Decision Tree (GBDT) algorithms to perform classification. For the second approach, we used a widely used CNN model named AlexNet (Krizhevsky et al., 2017) for gray-image classification.

The comparison experiments are based on all the familial classification datasets (AMGP1-2, DB1-4, and AMD1-4). We compare the average performance of GDroid with the baselines. The results are listed in Table 8. Since RF and SVM performed worse than GBDT, we omitted the corresponding results. As expected, GDroid achieved the best performance in all metrics.

Answer to RQ 4: GDroid outperforms the baselines in malware familial classification.

5. Deployment

We have demonstrated that GDroid can perform Android malware classification with high performance in the experimental environment. In this section, we discuss the deployment of GDroid in real-world scenarios.

The Graph Convolutional Network used by GDroid is inherently transductive, which means that all nodes should be included in the training stage (but the test samples are unlabeled). It is originally designed to be learned with the presence of both training and test samples and does not naturally generalize to unseen nodes. As a result, each inference needs to retrain the model. When deploying GDroid, the transductive feature makes GDroid be applied best to the scenarios where simultaneously inferring apps as many as possible, rather than inferring few apps at a time and performing multiple inferences. Additionally, we emphasize that the old parameters of Android APIs can be reused so that the time for retraining is much less than that of training from scratch.

In our research, we cooperate with QI-ANXIN Technology Research Institute (QI-ANXIN, 2021), deploying GDroid in real-world scenarios to evaluate its impact. Overall, there are about 10,000 malware samples captured in November 2020 and labeled by ensemble methods (including manual analysis and automated dynamic analysis based on sandbox), which can reflect the latest Android malware status.

The results are shown in Table 9. We emphasize that a slight degradation of performance in real-world scenarios is acceptable compared to the experimental environment because there is a gap between the experimental environment and the real-world scenarios (especially the malware samples). Comparing the results in Table 9 (ACC: 95.04%) and Table 7 (ACC: 96.98%), we can conclude that GDroid maintains satisfactory performance in real-world scenarios. Our approaches can help to complete the goal of malware classification in the real world.

6. Discussion

We use the API usage patterns to model the relevance of APIs, thereby facilitating the subsequent classification. In this section, we discuss the reasonability of applying the API usage patterns in malware detection and classification.

We would like to take the (Zhang et al., 2020) as evidence, a study on enhancing malware classifiers against model aging. The key observation is that malware samples, during evolution, often keep the same semantics but switch to a different implementation (mainly alternate use of APIs with similar functionalities) so that the evolve malware can avoid being detected by existing classifiers. The authors mine the APIs with similar usages from the official documents of Android API and then use the functionally similar API to enhance malware classifiers against model aging. In our research, we also mine the API usage patterns, summarizing the APIs with similar usages to facilitate malware detection and classification. They mine the API usage from the documents, while we adopt a data-driven strategy that mining the usage patterns from the real-world app samples. In our opinion, our practice is closer to reality, and the results should be more reflective of the real-world scenarios in principle.

7. Conclusion

This paper proposes a novel GCN-based approach for Android malware detection and familial classification. We map apps and Android APIs into a large heterogeneous graph, converting the app classification into a node classification task. To model the relevance among APIs, we present an embedding-based approach to mine API usage patterns.

We develop a prototype system named GDroid and conduct extensive experiments to evaluate its performance. Experimental results show that GDroid outperforms the existing approaches in the malware detection task and surpasses the baselines in the malware familial classification task. We also verify the contribution of API usage patterns to the improvement of model performance. Our work yields insights

into the utilization of API usage patterns for malware classification and shows promising results for studying malware classification via the graph neural network.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Han Gao: Conceptualization, Methodology, Software, Validation, Formal analysis, Writing - original draft, Writing - review & editing. **Shaoyin Cheng:** Resources, Funding acquisition, Supervision. **Weiming Zhang:** Resources, Funding acquisition, Supervision.

Acknowledgment

This work was supported in part by the Natural Science Foundation of China under Grant U20B2047, 62072421 and 62002334, Exploration Fund Project of University of Science and Technology of China under Grant YD3480002001, Fundamental Research Funds for the Central Universities under Grant WK2100000011, National Key Research and Development Program of China under Grant 2020YFA0309702, Public Service Platform Project for Industrial Technology Foundation under Grant 2019-00893-2-2, and by Anhui Initiative in Quantum Information Technologies under Grant AHY150400. The authors would like to thank the security experts of QI-ANXIN Technology Research Institute, especially Dr. Lingyun Ying for the help of real-world evaluation.

REFERENCES

Allen J, Landen M, Chaba S, Ji Y, Chung SPH, Lee W. Improving Accuracy of Android Malware Detection with Lightweight Contextual Awareness. In: Proceedings of the 34th Annual Computer Security Applications Conference. ACM; 2018. p. 210–21.

Alzaylaee MK, Yerima SY, Sezer S. Dl-droid: deep learning based android malware detection using real devices. Computers & Security 2020;89:101663 http://www.sciencedirect.com/science/article/pii/S0167404819300161. doi:10.1016/j.cose.2019.101663.

Androguard 2021. Androguard. https://github.com/androguard/androguard.

Au KWY, Zhou YF, Huang Z, Lie D. Pscout: Analyzing the android permission specification. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security. Association for Computing Machinery; 2012. p. 217–28. doi:10.1145/2382196.2382222.

Cai H, Meng N, Ryder B, Yao D. Droidcat: effective android malware detection and categorization via app-level profiling. IEEE Trans. Inf. Forensics Secur. 2019;14(6):1455–70.

Canfora G, Mercaldo F, Visaggio CA. An hmm and structural entropy based detector for android malware: an empirical study. Computers & Security 2016;61:1–18 http://www.sciencedirect.com/science/article/pii/S0167404816300499. doi:10.1016/j.cose.2016.04.009.

Chen D, Xu D, Li H, Sebe N, Wang X. Group consistent similarity learning via deep crf for person re-identification. The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018.

Chen Z, Wei X, Wang P, Guo Y. Multi-label image recognition with graph convolutional networks. CoRR 2019;abs/1904.03582.

Fan M, Liu J, Luo X, Chen K, Tian Z, Zheng Q, Liu T. Android malware familial classification and representative sample selection via frequent subgraph analysis. IEEE Trans. Inf. Forensics Secur. 2018;13(8):1890–905.

Gao H, Chen Y, Ji S. Learning graph pooling and hybrid convolutional operations for text representations. CoRR 2019;abs/1901.06965.

Garcia J, Hammad M, Malek S. Lightweight, obfuscation-Resilient detection and family identification of android malware. ACM Trans. Software Eng. Method. 2018;26(3):1–29.

Gascon H, Hübner M, Rieck K, Spreitzenbarth M, Arp D. In: 21st Annual Network and Distributed System Security Symposium, NDSS. Drebin: Effective and Explainable Detection of Android Malware in Your Pocket; 2014.

Gascon H, Yamaguchi F, Arp D, Rieck K. Structural detection of android malware using embedded call graphs. In: Proceedings of the ACM Workshop on Artificial Intelligence and Security. ACM; 2013. p. 45–54.

Google, 2017. Google Play Store. https://play.google.com/store.

Google, 2021. TensorFlow. https://www.tensorflow.org.

Hassen M, Chan PK. Scalable Function Call Graph-based Malware Classification. In: Proceeding of the ACM Conference on Data and Applications Security and Privacy (CODASPY); 2017. p. 239–48.

Hou S, Ye Y, Song Y, Abdulhayoglu M. HinDroid: An Intelligent Android Malware Detection System Based on Structured Heterogeneous Information Network. In: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM; 2017. p. 1507–15.

Irolla P, Dey A. The duplication issue within the drebin dataset. Journal of Computer Virology and Hacking Techniques 2018;14(3):1–5.

Jerbi M, Dagdia ZC, Bechikh S, Said LB. On the use of artificial malicious patterns for android malware detection. Computers & Security 2020;92:101743. doi:10.1016/j.cose.2020.101743. http://www.sciencedirect.com/science/article/pii/S0167404818309994.

Kim T, Kang B, Rho M, Sezer S, Im EG. A multimodal deep learning method for android malware detection using various features. IEEE Trans. Inf. Forensics Secur. 2019;14(3):773–88.

Kingma DP, Ba J. Adam: A method for stochastic optimization. 3rd International Conference on Learning Representations ICLR, 2015.

Kipf TN, Welling M. Semi-Supervised Classification with Graph Convolutional Networks. International Conference on Learning Representations (ICLR), 2016.

Krizhevsky A, Sutskever I, Hinton GE. Imagenet classification with deep convolutional neural networks. Commun. ACM 2017;60(6):84–90.

Li J, Sun L, Yan Q, Li Z, Srisa-An W, Ye H. Significant permission identification for machine-Learning-Based android malware detection. IEEE Trans. Ind. Inf. 2018;14(7):3216–25.

Liu B, Zhang T, Niu D, Lin J, Lai K, Xu Y. Matching long text documents via graph convolutional networks. CoRR 2018;abs/1802.07459.

Liu X, Lin Y, Li H, Zhang J. A novel method for malware detection on ml-based visualization technique. Computers & Security 2020;89:101682. doi:10.1016/j.cose.2019.101682. http://www.sciencedirect.com/science/article/pii/S0167404818314627.

Mao C, Yao L, Luo Y. Imagegcn: multi-relational image graph

convolutional networks for disease identification with chest x-rays. CoRR 2019;abs/1904.00325.
Matplotlib, 2021. https://matplotlib.org.
Martín A, Rodríguez-Fernández V, Camacho D. CANDYMAN: Classifying android malware families by modelling dynamic traces with markov chains. Eng Appl Artif Intell 2018;74:121–33.
McLaughlin N, Doupé A, Joon Ahn G, Martinez del Rincon J, Kang B, Yerima S, Miller P, Sezer S, Safaei Y, Trickel E, Zhao Z. Deep Android Malware Detection. In: Proceeding of the ACM Conference on Data and Applications Security and Privacy CODASPY. Association for Computing Machinery (ACM); 2017. p. 301–8.
Mikolov T, Chen K, Corrado G, Dean J. Efficient estimation of word representations in vector space. CoRR 2013;abs/1301.3.
Mikolov T, Sutskever I, Chen K, Corrado G, Dean J. Distributed Representations of Words and Phrases and Their Compositionality. In: Burges CJC, Bottou L, Welling M, Ghahramani Z, Weinberger KQ, editors. In: Advances in Neural Information Processing Systems 26. Curran Associates, Inc.; 2013. p. 3111–19.
Mirzaei O, Suarez-Tangil G, de Fuentes JM, Tapiador J, Stringhini G. Andrensemble: Leveraging api ensembles to characterize android malware families. In: Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security. Association for Computing Machinery; 2019. p. 307–14. doi:10.1145/3321705.3329854.
Narayanan A, Meng G, Yang L, Liu J, Chen L. Contextual weisfeiler-lehman graph kernel for malware detection. In: 2016 International Joint Conference on Neural Networks (IJCNN); 2016. p. 4701–8.
Nix R, Zhang J. Classification of Android apps and malware using deep neural networks, Vol. 2017-May; 2017. p. 1871–8.
Onwuzurike L, Mariconti E, Andriotis P, Cristofaro ED, Ross G, Stringhini G. Mamadroid: detecting android malware by building markov chains of behavioral models (extended version). ACM Transactions on Privacy and Security 2019;22(2) 14:1–14:34.
Pektas A, Acarman T. Learning to detect android malware via opcode sequences. Neurocomputing 2019.
QI-ANXIN, 2021. QI-ANXIN Technology Research Institute. https://research.qianxin.com/.
Shi C, Hu B, Zhao WX, Yu PS. Heterogeneous information network embedding for recommendation. IEEE Trans. Knowl. Data Eng. 2019;31(2):357–70.
scikit-learn: Machine Learning in Python, 2021. https://scikit-learn.org.
Sun Y, Han J. Mining heterogeneous information networks: principles and methodologies. Synthesis Lectures on Data Mining and Knowledge Discovery 2012;3(2):1–159.
Vasan D, Alazab M, Wassan S, Safaei B, Zheng Q. Image-based malware classification using ensemble of cnn architectures (imcec). Computers & Security 2020;92:101748. doi:10.1016/j.cose.2020.101748. http://www.sciencedirect.com/science/article/pii/S016740482030033X.
Vinayakumar R, Soman KP, Poornachandran P, Sachin Kumar S. Detecting android malware using long short-term memory (LSTM). Journal of Intelligent and Fuzzy Systems 2018;34(3):1277–88.
Wang H, Zhao M, Xie X, Li W, Guo M. Knowledge graph convolutional networks for recommender systems. CoRR 2019;abs/1904.12575.
Wang S, Yan Q, Chen Z, Yang B, Zhao C, Conti M. Detecting android malware leveraging text semantics of network flows. IEEE Trans. Inf. Forensics Secur. 2018;13(5):1096–109.
Wei F, Li Y, Roy S, Ou X, Zhou W. Deep Ground Truth Analysis of Current Android Malware. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment DIMVA. Springer; 2017. p. 1–20.
Wu L, Sun P, Fu Y, Hong R, Wang X, Wang M. A neural influence diffusion model for social recommendation. CoRR 2019;abs/1904.10322.
Xu K, Li Y, Deng RH, Chen K. DeepRefiner: Multi-layer Android Malware Detection System Applying Deep Neural Networks. In: Proceedings - 3rd IEEE European Symposium on Security and Privacy; 2018. p. 473–87.
Xu X, Liu C, Feng Q, Yin H, Song L, Song D. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. ACM; 2017. p. 363–76.
Yao L, Mao C, Luo Y. Graph convolutional networks for text classification. CoRR 2018;abs/1809.05679.
Yerima SY, Sezer S. Droidfusion: A Novel multilevel classifier fusion approach for android malware detection. IEEE Trans. Cybern. 2019;49(2):453–66.
Ying R, He R, Chen K, Eksombatchai P, Hamilton WL, Leskovec J. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In: Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM; 2018. p. 974–83.
Yuan B, Wang J, Liu D, Guo W, Wu P, Bao X. Byte-level malware classification based on markov images and deep learning. Computers & Security 2020;92:101740. doi:10.1016/j.cose.2020.101740. http://www.sciencedirect.com/science/article/pii/S0167404820300262.
Zhang D, Kabuka MR. Protein Family Classification with Multi-Layer Graph Convolutional Networks. In: Proceedings - IEEE International Conference on Bioinformatics and Biomedicine, BIBM; 2018. p. 2390–3.
Zhang L, Thing VL, Cheng Y. A scalable and extensible framework for android malware detection and family attribution. Computers & Security 2019;80:120–33. doi:10.1016/j.cose.2018.10.001. http://www.sciencedirect.com/science/article/pii/S016740481830419X.
Zhang M, Duan Y, Yin H, Zhao Z. Semantics-aware android malware classification using weighted contextual api dependency graphs. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM; 2014. p. 1105–16.
Zhang P, Cheng S, Lou S, Jiang F. A novel android malware detection approach using operand sequences. In: Third International Conference on Security of Smart Cities, Industrial Control System and Communications (SSIC); 2018. p. 1–5.
Zhang X, Zhang Y, Zhong M, Ding D, Cao Y, Zhang Y, Zhang M, Yang M. Enhancing state-of-the-art classifiers with api semantics to detect evolved android malware. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. Association for Computing Machinery; 2020. p. 757–70. doi:10.1145/3372297.3417291.
Zhou H, Zhang W, Wei F, Chen Y. Analysis of Android Malware Family Characteristic Based on Isomorphism of Sensitive API Call Graph. In: Proceedings - IEEE 2nd International Conference on Data Science in Cyberspace (DSC); 2017. p. 319–27.
Zhou Q, Feng F, Shen Z, Zhou R, Hsieh MY, Li KC. A novel approach for mobile malware classification and detection in android systems. Multimed. Tools Appl. 2019;78(3):3529–52.
Zhou Y, Jiang X. Dissecting Android malware: Characterization and evolution. In: Proceedings - IEEE Symposium on Security and Privacy; 2012. p. 95–109.
360, 2018. Android Malware Special Report. http://zt.360.cn/1101061855.php?dtid=1101061451&did=610100815.

Han Gao received the B.S. degree from the Wuhan University of Technology, Wuhan, China, in 2018. He is currently pursuing the M.S. degree with the University of Science and Technology of China. His research interests include malware analysis and software engineering.

Shaoyin Cheng received the B.S. and Ph.D. degrees from the School of Information Science and Technology, University of Science and Technology of China in 2003 and 2009. He is currently a Lecturer with the School of Information Science and Technology, University of Science and Technology of China. His research interests include information security, computer network and program analysis.

Weiming Zhang received the M.S. and Ph.D. degrees from the Zhengzhou Information S cience and Technology Institute, Zhengzhou, China, in 2002 and 2005, respectively. He is currently a Professor with the School of Information Science and Technology, University of Science and Technology of China. His research interests include multimedia s ecurity, information hiding, and privacy protection.