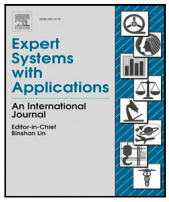
Expert Systems With Applications 235 (2024) 121125

Contents lists available at ScienceDirect

Expert Systems With Applications

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



SeGDroid: An Android malware detection method based on sensitive function call graph learning

Zhen Liu [a], Ruoyu Wang [b,*], Nathalie Japkowicz [c], Heitor Murilo Gomes [d], Bitao Peng [a], Wenbin Zhang [e]

[a] School of Information Science and Technology/School of Cyber Security, Guangdong University of Foreign Studies, Guangzhou, 510006, PR China
[b] Information and Network Engineering Research Center, South China University of Technology, Guangzhou, 510641, PR China
[c] Department of Computer Science, American University, Washington DC, 20016, USA
[d] School of Engineering and Computer Science, Victoria University of Wellington, Wellington, 6140, New Zealand
[e] Department of Computer Science, Michigan Technological University, Houghton, 49931, USA

ARTICLE INFO

Keywords:
Android malware detection
Function call graph
Graph neural network
Semantic knowledge
Model explanation

ABSTRACT

Malware is still a challenging security problem in the Android ecosystem, as malware is often obfuscated to evade detection. In such case, semantic behavior feature extraction is crucial for training a robust malware detection model. In this paper, we propose a novel Android malware detection method (named SeGDroid) that focuses on learning the semantic knowledge from sensitive function call graphs (FCGs). Specifically, we devise a graph pruning method to build a sensitive FCG on the base of an original FCG. The method preserves the sensitive API (security-related API) call context and removes the irrelevant nodes of FCGs. We propose a node representation method based on word2vec and social-network-based centrality to extract attributes for graph nodes. Our representation aims at extracting the semantic knowledge of the function calls and the structure of graphs. Using this representation, we induce graph embeddings of the sensitive FCGs associated with node attributes using a graph convolutional neural network algorithm. To provide a model explanation, we further propose a method that calculates node importance. This creates a mechanism for understanding malicious behavior. The experimental results show that SeGDroid achieves an F-score of 98% in the case of malware detection on the CICMal2020 dataset and an F-score of 96% in the case of malware family classification on the MalRadar dataset. In addition, the provided model explanation is able to trace the malicious behavior of the Android malware.

1. Introduction

Due to the open-source nature of the Android operating system, Android apps have become the main target of cybercriminals (Lo et al., 2022). Malicious behaviors of cybercriminals include: browser hijacking, malicious collection of user information (such as credit card and contact data), malicious bundling and launching of unwanted advertisements (Android Statistics, 2022, Xu et al., 2021). The emergence of a massive number of malware attacks poses a considerable challenge to malware mitigation (Gao et al., 2021).

Many methods have been proposed to cope with malware detection, including signature-based (Grace et al., 2012; Zheng et al., 2013) and machine learning based (Razgallah et al., 2021) methods. Recently, machine learning, specifically deep learning, has been a promising solution for malware detection because it has the potential to keep up with the speed at malware evolving (Guerra-Manzanares et al., 2021; Ou & Xu, 2022). Feature extraction is crucial to improve malware detection performance in machine learning approaches.

Commonly, two types of techniques are used to extract features from Android apps, i.e. static analysis (Martín et al., 2019; Ou & Xu, 2022; Vasan et al., 2020) and dynamic analysis (Ananya et al., 2020; Lin et al., 2022). Static analysis-based techniques extract features from the assembled files of app installation packages. For example, AndroPyTool (Martín et al., 2019) extracts permissions, intents, services and providers from AndroidManifest.xml and API calls from Smali codes. Dynamic analysis-based techniques mainly focus on the system or network traffic data while running apps. These techniques require

The code (and data) in this article has been certified as Reproducible by Code Ocean: (https://codeocean.com/). More information on the Reproducibility Badge Initiative is available at https://www.elsevier.com/physical-sciences-and-engineering/computer-science/journals.

* Corresponding author.
E-mail addresses: liuzhen@gdufs.edu.cn (Z. Liu), rywang@scut.edu.cn (R. Wang), japkowic@american.edu (N. Japkowicz), heitor.gomes@vuw.ac.nz (H.M. Gomes), pengbitao@gdufs.edu.cnu (B. Peng), wenbinzh@mtu.edu (W. Zhang).

https://doi.org/10.1016/j.eswa.2023.121125
Received 14 February 2023; Received in revised form 3 August 2023; Accepted 4 August 2023
Available online 11 August 2023

executing the apps, which causes overhead and inconvenience (Cai et al., 2021). Most approaches thus rely on static analysis for feature extraction (Shar et al., 2020).

In the static analysis, recent works pay more attention on semantic feature extraction from graphs (Lo et al., 2022; Onwuzurike et al., 2019; Ou & Xu, 2022). This is because that semantic features are more robust than syntax features when confronting malware evasion (Wu, Li, et al., 2019). An function call graph (FCG) consists of a set of program functions and their interprocedural calls (Lo et al., 2022), which can capture the caller-callee relationship between methods inside an APK. The structure of the graph can be further embedded as the input of machine learning algorithms. In particular, recent work has used the Graph Neural Networks(GNNs) to embed FCGs (Xu et al., 2021). The GNN is able to leverage the topological structure and node features to generate informative embedding for each node (Gao et al., 2021).

The nodes in the FCG may be the system APIs or the self-defined functions. A variety of methods have been proposed for node representation (obtaining node feature vectors). Xu et al. (2021) applied the SIF (Arora et al., 2017) to vectorize the opcode sequences on a node. Cai et al. (2021) took the functions as words and leveraged word2vec to embed the functions into vectors; however, attackers can easily obfuscate the self-defined functions. Vinayaka and Jaidhar (2021) leveraged the API package list for representing the external node (system API) and the opcode list for the internal node (self-defined function). This allowed them to handle the obfuscation problem of self-defined functions. In addition, they also proposed a node balance method for handling the node imbalance between benign and malware. However, their node attributes are based solely on the occurrence of the APIs or the opcodes. They do not carry semantic knowledge in the API and opcode sequence. The node balance method is implemented by randomly removing the samples from the training set based on occurrence and not on semantic knowledge, and it is likely that some informative samples are removed. To address the different shortcomings mentioned in this discussion, this paper proposes a new Android malware detection method named SeGDroid that takes semantic knowledge information. The main contributions of this paper include the following five items:

(1) We propose a new Android malware detection method based on the sensitive FCG and GNN. Our method first builds the sensitive FCGs and extracts the semantic attributes for function nodes. It then implements a GNN for embedding FCGs into feature vectors. These vectors can be further used to train a malware detection model using a machine learning algorithm.

(2) Regarding sensitive FCG building, we propose a graph pruning method that preserves the context of sensitive API calls (i.e. security-related API calls that require specific permissions) while leaving out the others. This reduces the number of nodes included in huge FCGs while simultaneously handling the node imbalance problem without removing any app samples. In addition, it causes the model to pay more attention to the malicious behavior that correlates with sensitive APIs.

(3) From the aspect of feature representation for graph nodes, we propose to train two models based on the word2vec (Mikolov et al., 2013a) algorithm: an API2vec model and an opcode2vec model for embedding external nodes (system APIs) and internal nodes (opcode sequence), respectively. This is a good way to keep the semantics of APIs and the opcode sequence. System APIs and opcodes are not easily obfuscated by attackers, so the node representation based on them is robust to obfuscation techniques. In addition, social-network-based centrality (Wasserman & Faust, 1994) is further used to weight the feature vectors. The feature vectors thus carry the graph structure knowledge. This mechanism is able to accelerate the convergence of graph learning.

(4) To explain the FCG based malware detection model, we propose a method to visualize the importance of different nodes and find that the majority of nodes with high importance correspond to the sensitive APIs. This provides useful information to the user.

(5) We perform multiple experiments to evaluate the performance of our method, including binary classification, category classification and malware family classification. Experimental results show that our method achieves an F-score of 98% for detecting malware and an F-score of 96% for malware family classification on average.

The remainder of the paper is organized as follows. Section 2 overviews related works. Section 3 analyzes the characteristics of FCGs and proposes our malware detection method. Section 4 introduces our experimental datasets, analyzes the experimental results and visualizes the node importance in FCGs. Section 5 concludes this paper.

2. Related work

2.1. Related work on feature extraction in Android malware detection

Android malware detection methods have evolved from signature-based methods to machine learning-based methods. Machine learning approaches critically depend on feature extraction. The feature extraction methods for malware detection can be categorized into static analysis and dynamic analysis methods. Static analysis methods (Liu et al., 2021; Onwuzurike et al., 2019) extract features from app installation files, including AndroidManifest.xml and decompiled Smali codes. Various features, such as permissions, API calls, intent filters, opcode, FCG, etc., have been widely researched (Kabakus, 2022; Qiu et al., 2023; Tang et al., 2022; Zhang et al., 2021). Static analysis-based methods implement malware detection before running apps. The dynamic analysis-based methods (Ananya et al., 2020; Guerra-Manzanares et al., 2022; Lin et al., 2022; Wang et al., 2020) extract features from the data (such as system calls and network traffic) while running apps. It acquires executing the apps for detecting malware. Static analysis and dynamic analysis have also been combined for Android malware detection (Alzaylaee et al., 2020; Li et al., 2018; Martín et al., 2019).

In static analysis approaches, the extracted features are categorized into: (1) occurrence-based features; (2) image-based features; (3) text-based features; and (4) graph-based features.

Regarding *occurrence-based features*, the API calls, permissions, intents, activities, and others extracted from the static APK files are combined together as features (Badhani & Muttoo, 2019; Scalas et al., 2019). The feature value is 1 if the feature exists in an app, and the feature vector with binary value is used to represent an app (Badhani & Muttoo, 2019; Kong et al., 2022). Also, the frequency of the feature occurring in an app could be recorded for each feature (Martín et al., 2019). Badhani and Muttoo (2019) propose a malware detection method named CENDroid, which uses the API and Permissions as features. The feature value is binary according to whether the feature exists in the Smali codes and AndroidManifest.xml file. The clustering and ensemble of classifiers are applied for malware detection. Experimental results on their datasets show that it obtains 98% accuracy in the best case. OmniDroid (Martín et al., 2019) builds benchmark datasets taking API, Permission, Intents, Service, Activity, Receiver, System commands, FlowDroid, Package name, and Strings as features. The feature value is the frequency of the feature that occurs in an app.

Regarding *image-based features*, the DEX files are transformed into images, and the images are fed into the CNN(Convolutional Neural Networks) for training an Android malware detection model (D'Angelo et al., 2020; Naït-Abdesselam et al., 2020; Vasan et al., 2020). Vasan et al. (2020) propose an image-based malware detection method using a fine-tuned CNN. It first converts the raw malware binaries into color images. The pretrained CNN on ImageNet datasets is fine-tuned on the malware datasets. The results on the IoT-android mobile dataset show that it obtains 97% accuracy. Naït-Abdesselam et al. (2020) also propose an image-based malware detection method. It transforms each APK into an RGB image. The permissions and components from Android AndroidManifest.xml obtain the green channel of the image. The API calls and unique opcode sequences of the DEX file obtain the red channel of the image. The blue channel of the image is obtained

by the strains, suspected permission, app components and API calls. The experiments on the AndroidZoo dataset show that it obtains 99% accuracy.

Concerning the *text-based features*, the APIs and opcodes are handled as texts (Sun et al., 2019; Zhang et al., 2021), and then the feature learning method in NLP (Natural Language Processing) is adopted, such as word2vec (Rong, 2014). Sun et al. (2019) utilize APIs, permissions and metadata for characterizing apps. The metadata include the category and description of apps. The word2vec is applied to vectorize the metadata. Zhang et al. (2021) propose a TC-Droid method based on the text sequence of permissions, service, intent and receiver. They apply a text CNN to learn the features from the original text. The results on the Genome dataset show that it obtains approximately 96% accuracy.

On the aspect of *graph-based features*, graphs provide an abstraction representation for modeling the behaviors of Android applications. A variety of graphs have been researched for Android malware detection, including the graph with control flow and data flow (Alhanahnah et al., 2020), the heterogeneous information network among apps and APIs (Hou et al., 2017), and FCG (Lei et al., 2019; Onwuzurike et al., 2019; Wu, Li, et al., 2019). MaMadroid (Onwuzurike et al., 2019) builds API paths for each app obtained from an FCG and then abstracts APIs to their corresponding package families. It then transfers all abstracted paths to a feature vector for an app using the Markov model. Their experimental results show that the model achieves an F-measure of 98% in the best case. Malscan (Wu, Li, et al., 2019) combines all sensitive APIs as the feature set. The feature vector values denote the occurrence frequency of the corresponding APIs. To learn the structure of FCGs, it leverages the centrality metrics defined in the social network to weight the feature vectors. For the results on the Androidzoo dataset, Malscan achieves 98% accuracy.

2.2. Related work on GNN based Android malware detection

Recently, researchers have paid more attention to utilizing the GNN to extract the feature vectors for representing FCGs. Cai et al. (2021) propose enhanced FCGs (E-FCGs) to characterize app behaviors. They build the corpus of functions by putting together all the function call records and adopting the CBOW(Continuous Bag of Words) algorithm for embedding functions. They further acquire the enhanced FCGs with node attributes obtained by function embedding. Then, the GCN algorithm is used to learn a feature vector for each FCG. The learned features are used as the input of the linear regression, decision tree, SVM(support vector machine), KNN(K nearest neighbor), random forest, MLP(Multilayer Perceptron) and CNN algorithms. The experimental results on the Androzoo and Google app store datasets show that it obtains 99% accuracy in the best case. However, the function-based node attributes rely on the function names that are easily obfuscated by renaming.

Xu et al. (2021) apply the opcode sequences of each function for node representation. They utilize the SIF network (Arora et al., 2017) to learn the feature vectors of the opcode sequence and the structure2vec algorithm for graph embedding. The obtained vectors are used as the input of MLP. The results on the Drebin, AMD, AndroZoo and PraGuard datasets show that it obtains 99% accuracy. However, the DEX file does not contain the implementation of external functions (Vinayaka & Jaidhar, 2021), on which this method cannot obtain the opcode sequence about the function implementation. Vinayaka and Jaidhar (2021) apply APIs to represent the external nodes and opcode sequences to represent the internal nodes. Then, they utilize multiple GNNs for graph embedding and acquire the feature vector for each app. Specifically, they proposed a node balance method for handling the node imbalance problem in FCGs. The results on the CIC and Androzoo datasets show that it obtains 92% accuracy with the GraphSAGE algorithm. On the aspect of node representation, this method only considers the existence of API calls and opcodes but omits the semantic knowledge of the API calls and opcode sequences. The node balance method may remove informative samples for malware detection. In addition, there is no work involving the model explanation in the field of FCG-based Android malware detection.

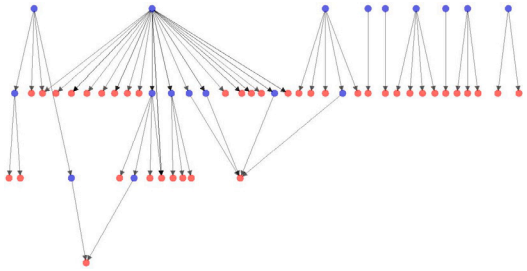


Fig. 1. The example of an FCG.

To cope with the challenges mentioned above, we propose a new Android malware detection method called SeGDroid. Our method utilizes word2vec (Mikolov et al., 2013a) and centrality (Wasserman & Faust, 1994) for the node semantic representation. Specifically, with the domain knowledge of Android malware detection, we propose a graph pruning method for simplifying the FCG. It preserves the sensitive API correlated nodes and removes the irrelevant nodes to decrease the node imbalance ratio between the malware and benign app, reduce the resource consumption of training, and encourage the effect of nodes with sensitive APIs for malware detection. In addition, to explain the FCG-based model, we devise a method to visualize the importance of graph nodes for Android malware detection.

3. Methodology

3.1. Function call graph analysis

This section mainly analyzes the characteristics of FCGs. The FCG is defined as:

Definition 1 Function call graph: FCG is a directed graph. $FCG = \langle \mathcal{V}, \mathcal{E} \rangle$, where $\mathcal{V}$ is the set of functions (i.e., callers and callees); $\mathcal{E}$ is the set of directed edges between callers and callees.

One example of an FCG is shown in Fig. 1, in which a function is denoted by a node. If a function A is invoked in another function B, there is an edge from B to A. The external functions are represented in red, and the internal functions are in blue. The function node without indegree is the entry point of a function running path (Lei et al., 2019), such as onReceive() and onCreate(). Previous works (Ou & Xu, 2022; Wu, Li, et al., 2019) have utilized sensitive API calls for malware detection. The context of sensitive API calls is also important for malicious reorganization. For example, if the functions of acquiring user information are related to GUI events, this may be triggered by the users; otherwise, this may be triggered by attackers (Meng et al., 2018). Therefore, we also consider the context of the invoked sensitive API, i.e., these nodes in the path from the entry point to the sensitive API and the path from the sensitive API to the leaf node.

Next, we analyze the sensitive API node distribution in the malware FCGs and benign FCGs. The CDF (Cumulative Distribution Function) of the sensitive API ratio is shown in Fig. 2. It shows that the sensitive API ratio in the malware is larger than that in the benign class. This indicates that malwares invoke sensitive APIs with a high probability.

3.2. The framework of SeGDroid

The framework of SeGDroid is shown in Fig. 3. It mainly includes three parts.

(1) **Function call graph building and pruning:** We first unzip, decompile and build the FCGs from the APKs using Androidguard (Androguard, 2022). Some graphs may have a large number of nodes,

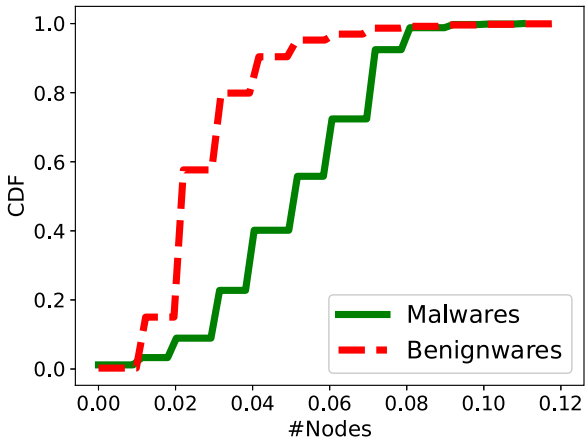


Fig. 2. The CDFs of the sensitive API ratios in malwares and benign apps.

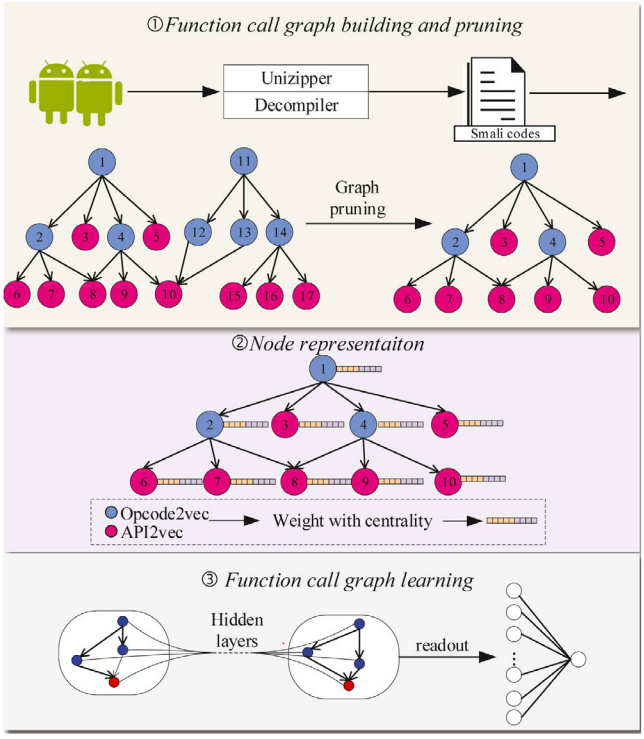


Fig. 3. The framework of SeGDroid.

which would increase the complexity of graph learning. To simplify the graphs while preserving the semantic knowledge of malicious behavior, we propose a graph pruning method that retains the nodes in the function running path with sensitive APIs. The sensitive APIs used for building sensitive FCGs are on the basis of the mappings of APIs and permissions reported by PScout (Au et al., 2012). There are 21,986 sensitive APIs (Wu, Li, et al., 2019).

(2) **Node representation:** We transfer the API and opcode sequence into a vector for each node to obtain the node attributes. We use a centrality measure to weight the node vectors to improve the representation ability of node vectors. Centrality measures (Wasserman & Faust, 1994) are widely used in social network studies to denote the importance of nodes to some extent. To the best of our knowledge, this is the first work that utilizes the centrality measure to weight the node feature vectors used for GNN.

(3) **Function call graph learning:** On the FCGs associated with node feature vectors, we utilize GraphSAGE (Hamilton et al., 2017) to perform graph embedding. GraphSAGE embeds each graph node by iteratively learning the knowledge from the corresponding neighbor nodes. All graph node vectors are combined into a vector by the readout. We then train a malware detection model on the obtained feature vectors using machine learning algorithms.

3.3. Function call graph pruning

A graph with a large number of nodes would increase the time consumption of graph learning. Graph pruning is a way to decrease the number of nodes in a graph. In malware detection, it is important to preserve malicious behavior-related nodes. The sensitive APIs are correlated with malicious behavior (Wu, Li, et al., 2019). In addition, the context of sensitive APIs is also helpful for malware detection. Therefore, this paper proposes a sensitive API-based graph pruning method. It aims at preserving the sensitive APIs and their context. The pruned graph is called sensitive FCG. The sensitive node and sensitive FCG are respectively defined as below.

Definition 2 Sensitive node: Among the nodes in an FCG, the function of a node that matches a sensitive API is defined as a sensitive node, i.e., $\{S_{sv} \mid S_{sv} \in S_{api}, S_{sv} \in \mathcal{V}\}$. S_{sv} denotes the set of sensitive nodes and S_{api} denotes the sensitive API set.

Definition 3 Sensitive function call graph: the sensitive function call graph is a subgraph of an FCG. $SG = \{\mathcal{V}, \mathcal{E} \mid \mathcal{V} \in \mathcal{N}(S_{sv})\}$; $\mathcal{N}(S_{sv})$ is the set of all neighbors that can reach the sensitive nodes in S_{sv}.

Graph pruning is shown in Algorithm 1. It aims to preserve the sensitive API-related nodes and leave out the remaining ones. Fig. 4 is an example of graph pruning. The details of Algorithm 1 are illustrated as follows.

(1) Lines 1 to 5 search the sensitive nodes. It obtains a set S_{sv} including all sensitive nodes in the graph. This corresponds to the step 1 in Fig. 4, acquiring $S_{sv} = \{v_8\}$. The node v_8 is highlighted in yellow.

(2) Lines 6 to 8 search the neighbors of the nodes in S_{sv} set in the upward direction of the graph. This means that it searches all the ancestor nodes of the sensitive nodes until the entry point. This step obtains the ancestor node set of sensitive nodes that is denoted by S_{sav}. This corresponds to the step 2 in Fig. 4, acquiring $S_{sav} = \{v_2, v_4, v_1\}$.

(3) Lines 9 to 11 further search the descendant nodes of the nodes in S_{sav}. For each node in S_{sav}, it searches all the descendant nodes until the leaf node in the downward direction of the graph. This step acquires the descendant node set that is denoted by S_{sdv}. This corresponds to the step 3 in Fig. 4, acquiring $S_{sdv} = \{v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}\}$.

(4) Line 12 removes the nodes and correlated edges that are not in S_{sdv}. This corresponds to the step 4 in Fig. 4. According to the obtained node set (the union of S_{sdv} and S_{sav}), $\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}\}$, we preserve those nodes and remove all other nodes of the graph in Fig. 4. As a result, we acquire the simplified graph used for graph embedding.

The graph pruning method has the following four priorities. (1) It reduces the training resource consumption of graph embedding by decreasing the number of nodes. (2) It handles the node imbalance problem among apps because it significantly reduces the number of nodes for a big graph that has a large number of nodes. (3) It encourages the effect of the sensitive nodes in graph embedding. We implement graph readout on all nodes' vectors and acquire the final vector for a graph. The effect of the sensitive nodes in the readout is encouraged by removing the nodes that are not correlated with sensitive nodes. (4) It preserves the context of invoking sensitive APIs. This is because it retains all function running paths passing through the sensitive nodes.

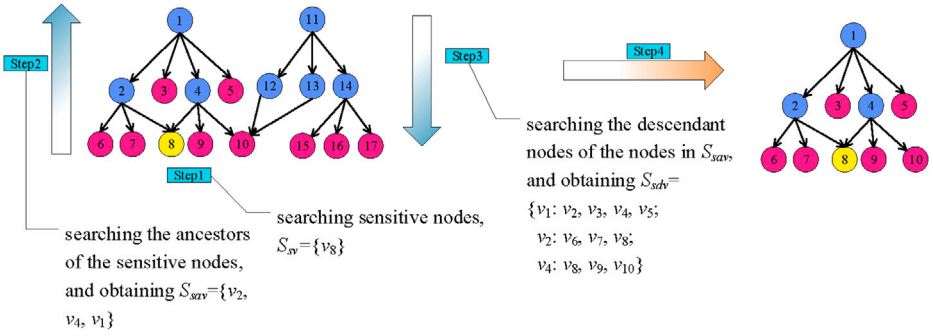


Fig. 4. The example of FCG pruning process.

Algorithm 1 Function call graph pruning algorithm

Input: an FCG $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, sensitive API set S_{api}
Output: a sensitive FCG $\mathcal{SG} = (\mathcal{V}, \mathcal{E})$

```
for v_i in V do
    if v_i ∈ S_api then
        S_sv ← S_sv ∪ v_i
    end if
end for
for sv_i in S_sv do
    addAncestor(sv_i, S_sav, G)
end for
for sav_i in S_sav do
    addDescendant(sav_i, S_sdv, G)
end for
SG=obtainPrunedGraph(G, S_sdv,S_sav)
```

3.4. Node representation

3.4.1. The node embedding based on word2vec

This section aims at learning a feature vector for each node in an FCG while considering the semantic information of functions in each node. To represent the function calls, we handle the internal and external functions differently. This is because the internal functions are usually those defined by the programmers, whose function names are easily obfuscated. Instead of using function names, we utilize the opcode sequence to represent the internal function. External functions that are usually the APIs of existing program libraries may also be changed when updating the libraries. The package names are more stable than the function names. Therefore, the API package names are used to represent the external nodes.

To learn the semantics of APIs and opcode sequences, we respectively train an embedding model based on word2vec algorithm, obtaining API2vec and opcode2vec models. To build the API2vec model, the API packages are collected to form the corpus. Word2vec is applied to acquire the feature vector for each word in the package. The average of the vectors of all words in a package is the feature vector for an external node. The package names have the semantic knowledge of APIs. According to the high cohesion in software design, the APIs in the same package may have similar usage purpose and their feature vectors should be close.

To build the opcode2vec model, the opcode corpus is first built by the opcode sequences in the training set. An opcode is handled as a word. The opcode2vec model is trained on the opcode corpus. For a node with an opcode sequence, the average over the vector of each opcode is acquired as the vector for an opcode sequence. The surrounding opcodes of an opcode in the sequence represent its usage context. The opcodes that appear in the similar usage context should have close vectors (Khan et al., 2022).

Word2vec is an unsupervised learning NLP technique that generates context-aware embeddings for words (Gao et al., 2021; Mikolov et al., 2013a). Word2vec contains two models: Skip-gram and CBOW (Rong, 2014). Skip-gram performs better for the infrequent words (Gao et al., 2021) empirically. We adopt the skip-gram for node embedding, because sensitive APIs are not invoked frequently.

Next, taking API2vec as an example, we further illustrate the process of node embedding. Since the self-defined functions are easily confused by changing their names, we only handle the system APIs. On the system API corpus collected from the apps in the training set, we train the API2vec model shown in Fig. 5.

The skip-gram utilizes a fixed-size sliding window that moves on the texts with multiple words to generate the training samples. The objective of training is to update the word embedding, so as to predict the surrounding context words. Given the words in an API package, $a_1, a_2, \ldots, a_K$ and the window size of 2m+1, the model maximizes the average log probability as

$$J(a) = 1/K \sum_{t=1}^{K} \sum_{-m \le j \le m} log P(a_{t+j}|a_t) \tag{1}$$

$P(a_{t+j}|a_t)$ is defined as

$$P(a_{t+j}|a_t) = \frac{exp(\mathbf{V}_{a_t}^{\top} \mathbf{V}_{a_{t+j}})}{\sum_{i=1}^{L} \mathbf{V}_{a_t}^{\top} \mathbf{V}_{a_i}} \tag{2}$$

Where $\mathbf{V}_{a_t}$ and $\mathbf{V}_{a_{t+j}}$ are the corresponding embeddings of the a_t and a_{t+j}, respectively, and L is the size of the vocabulary. However, this formulation is expensive to optimize because the number of parameters to be updated is much high when the vocabulary is very large. In practice, negative sampling (Mikolov et al., 2013b) and hierarchical softmax (Mikolov et al., 2013) are used to decrease the resource consumption and to improve the embedding quality.

In an FCG, each node is represented by the concentration of the feature vectors obtained by API2vec and opcode2vec. For an external node $v_i \in \mathcal{V}$, $\mathbf{V}_a^i$ is a vector obtained by API2vec, and $\mathbf{V}_o^i$ is a vector with zero value. For an internal node $v_j \in \mathcal{V}$, $\mathbf{V}_o^j$ is a vector obtained by opcode2vec, and $\mathbf{V}_a^j$ is a vector with zero value. Therefore, the vector $\mathbf{V}^i$ of a node v_i is formulated as

$$\mathbf{V}^i = [\ \mathbf{V}_a^i, \mathbf{V}_o^i] \tag{3}$$

Where $\mathbf{V}^i \in \mathbb{R}^{(|\mathbf{V}_a^i|+|\mathbf{V}_o^i|)}$, $\mathbf{V}_a^i = mean(\mathbf{V}_{a_1}^i, \mathbf{V}_{a_2}^i, \ldots, \mathbf{V}_{a_k}^i)$ for the external node; $\mathbf{V}_o = mean(\mathbf{V}_{o_1}^i, \mathbf{V}_{o_2}^i, \ldots, \mathbf{V}_{o_q}^i)$ for the internal node; a_i denotes the ith word in an API package; o_i denotes the ith opcode in an opcode sequence.

3.4.2. Feature vector weighting based on the centrality metric

Considering the importance of different functions, we introduce the centrality concept into FCG-based malware detection. To the best of our knowledge, this is the first paper to utilize the centrality in

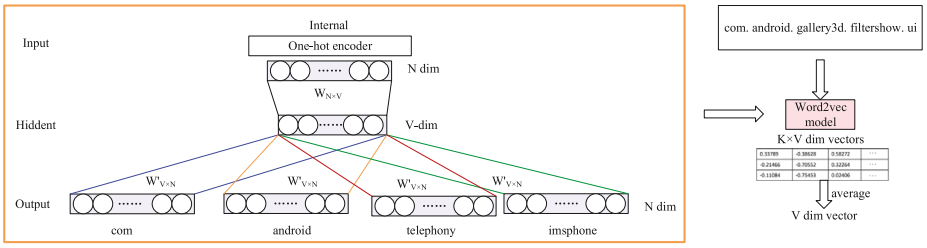


Fig. 5. The architecture of API2vec model.

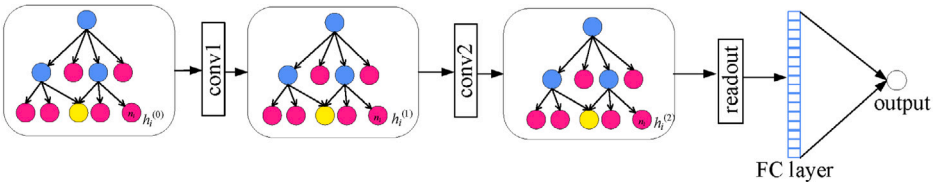


Fig. 6. The architecture of graph convolutional network model.

graph learning-based Android malware detection. The centrality concepts were first devised in social network analysis and used to measure the importance of a node in the network (Wu, Li, et al., 2019). Centrality analysis has been successfully utilized in different areas (e.g. program dependency networks (Wu et al., 2022), transportation networks (GuimerĂ et al., 2005)). Different types of centrality have been proposed to quantify the importance of a node in a network from different aspects, such as degree centrality (Freeman, 1978), closeness centrality (Freeman, 1978), EigenCentrality (Newman, 2010), and others. According to our empirical experiments, we apply degree centrality as the weight for each node's vector. Because it is efficient and effective. It is defined as Eq. (4), where $deg(i)$ denotes the degree of the ith node and n denotes the number of nodes in a graph.

$$d_i = \frac{deg(i)}{n-1} \tag{4}$$

Finally, the vector of a node is represented by

$$\mathbf{h}_i = d_i * \mathbf{V}^i \tag{5}$$

Where $\mathbf{V}^i$ is a vector obtained by API2vec and opcode2vec, and $\mathbf{h}_i$ is the final vector obtained by node representation in this paper.

3.5. Function call graph learning

After building the sensitive FCGs with the nodes associated with vectors, we further transform the graphs into vectors by a graph embedding algorithm. GNN (Kipf & Welling, 2017) embeds nodes of graphs while considering the topological information of the graph. Different kinds of GNN algorithms have been proposed. In Vinayaka and Jaidhar (2021), the authors compared GCN (Zhang et al., 2022), GraphSAGE (Hamilton et al., 2017), DotGAT (Velickovic et al., 2017), and TAG (Du et al., 2017), and the results show that GraphSAGE performs the best in malware detection. In this paper, we utilize the GraphSAGE for graph embedding.

The main structure of the GraphSAGE based neural network is shown in Fig. 6. Two convolutional layers are used to learn the latent representations of FCGs. The vector obtained by readout is fed into the fully connected layer that follows an output layer. The output can predict the class label of an unknown app. GraphSAGE computes the node embeddings by aggregating the neighbor node's features (Lo et al., 2022), and iteratively updates the node vectors at the object of minimizing the cross entropy loss of predicting the class label of an app. The vector of the ith node at the $(l+1)$ layer is formulated as

$$\mathbf{h}_i^{(l+1)} = \sigma(\mathbf{W}^{(l)}\text{concat}(\mathbf{h}_i^{(l)}, \mathbf{h}_{\mathcal{N}(i)}^{(l+1)})) \tag{6}$$

$$\mathbf{h}_{\mathcal{N}(i)}^{(l+1)} = \text{aggregate}(\mathbf{h}_j^{(l)}, \forall j \in \mathcal{N}(i)) \tag{7}$$

$$\mathbf{h}_i^{(l+1)} = \text{norm}(\mathbf{h}_i^{(l+1)}) \tag{8}$$

$$\text{norm}(\mathbf{h}) = \frac{\mathbf{h}}{\|\mathbf{h}\|_2} \tag{9}$$

$$\mathbf{h}_{\mathcal{G}} = \frac{1}{|\mathcal{V}|}\sum_{v\in\mathcal{V}} \mathbf{h}_v \tag{10}$$

Where $\mathbf{W}^{(l)}$ is the weight matrix, aggregate is the mean of the representation of neighboring nodes, σ is the activation function (i.e. ReLu), the norm is the normalization function of the new node representation. $\mathbf{h}_{\mathcal{G}}$ is the vector of a graph by readout, and it is obtained by the average over vectors of all nodes.

3.6. Model explanation

Inspired by the visualization work in the vulnerability detection research field (Wu et al., 2022), we visualize the node importance of the FCGs in malware detection. The visualization helps to understand the graph learning based malware detection,

After graph embedding, each node is represented by a vector with semantic and topology knowledge. This paper utilizes mean readout for acquiring a vector for each graph. The relation between the node vector and graph vector is shown in Fig. 7. The graph vector is formulated as

$$\mathbf{h}_{\mathcal{G}} = \left[g_1, g_2, \cdots g_k\right]^{\mathsf{T}} = 1/n \begin{bmatrix} x_1^1 + \cdots + x_1^n \\ \cdots \\ x_k^1 + \cdots + x_k^n \end{bmatrix} \tag{11}$$

Where x_i^j denotes the ith vector of the jth node, k is the number of features after graph embedding, and n is the number of nodes in a graph.

The output $\hat{y}$ of a sample is formulated as

$$\hat{y} = \mathbf{W} * \mathbf{H} + b \tag{12}$$

Where $\mathbf{W}$ denotes the weight vector learned by training the graph neural network, as shown in Fig. 7.

This could be further detailed as:

$$\begin{aligned}\hat{y} &= \left[w_1, w_2, \ldots, w_k\right] * \left[g_1, g_2, \ldots g_k\right]^{\mathsf{T}} + b \\ &= \left[w_1, w_2, \ldots, w_k\right] * 1/n \begin{bmatrix} x_1^1 + \cdots + x_1^n \\ \cdots \\ x_k^1 + \cdots + x_k^n \end{bmatrix} + b\end{aligned}$$

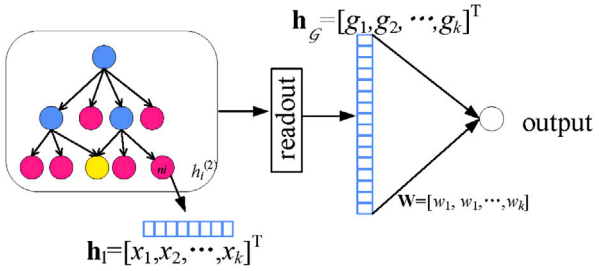


Fig. 7. Relation among the node vector, graph vector and weight vector.

$$= 1/n(w_1 * (x_1^1 + \cdots + x_1^n) + \cdots + w_k * (x_k^1 + \cdots + x_k^n)) + b$$
$$= 1/n((w_1 * x_1^1 + w_2 * x_2^1 + \cdots + w_k * x_k^1)$$
$$+ \ldots + (w_1 * x_1^n + w_2 * x_2^n + \cdots + w_k * x_k^n)) + b \quad (13)$$

According to Eq. (13), $(w_1 * x_1^j + w_2 * x_2^j + \cdots + w_k * x_k^j)$ denotes the contribution of the jth node to the prediction. The higher the value is, the larger the contribution is for malware detection. Therefore, we take the value of $w_i * x_i^j$ to denote the importance of the ith feature of the jth node.

In an FCG, the importance value for each node is calculated in the following three steps.

Step 1: Obtain the weight vector $\mathbf{W} = [w_1, w_2, \ldots, w_k]$ at the output layer.

Step 2: Extract the vector of each node $\mathbf{h}_i = [x_1^i, x_2^i \ldots, x_k^i](i = 1, \ldots, n)$ by performing graph embedding.

Step 3: Calculate the importance vector $[w_1 * x_1^i, \ w_2 * x_2^i, \ldots, w_k * x_k^i]$ $(i = 1, \ldots, n)$ for each node.

4. Experiments

4.1. Datasets

In our experiments, two publicly shared benchmark datasets are applied. They are introduced as below.

4.1.1. CICMal2020 dataset

The samples in CICMal2020 (Mahdavifar et al., 2022) are provided by the CIC institute and can be downloaded from Mahdavifar et al. (2022). The Android apks are from several sources, including VirusTotal service, Contagio security blog, AMD and MalDozer. These malware samples were collected from December 2017 to December 2018. They are from the five categories of Benign, Adware, Banking malware, SMS malware and Riskware. There are respectively 4043, 1511, 2282, 4821 and 3938 apks in the five categories.

4.1.2. MalRadar dataset

MalRadar (Wang et al., 2022) is a growing and up-to-date Android malware dataset. The family labels of some samples have been provided by this dataset. Therefore, we perform the family classification experiments on this dataset. Some malware families contain only a few samples, which are not sufficient for training the graph embedding model. Therefore, in the following experiments, the 15 families with the highest number of samples are chosen for experiments. The details of the MalRadar dataset are shown in Table 1. Since there are only malware samples in this dataset, we further downloaded benign samples from AndroZoo. AndroZoo is also an up-to-date dataset. We downloaded the apks with the VTScan timestamp in the year of 2022. To balance the number of samples between benign and these family classes in MalRadar, we randomly selected about one thousand apks with benign label from AndroZoo. The benign samples (1024 apks) are combined with the malware samples of MalRadar for comparison experiments.

Table 1
The number of samples in each class of MalRadar dataset.

Families	#apps	Families	#apps	Families	#apps
KBuster	54	FAKEBANK	80	GhostClicker	181
ZNIU	59	Lucy	80	HiddenAd	287
SpyNote	63	GhostCtrl	109	LIBSKIN	240
Joker	72	EventBot	124	Xavier	589
FakeSpy	74	MilkyDoor	208	RuMMS	795

4.2. Experiments

To evaluate the performance of SeGDroid, we carry out experiments from the following four aspects.

(1) Ablation experiment: We evaluate the performance of different parts in SeGDroid, specifically analyzing whether graph pruning and node representation are able to improve malware detection performance.

(2) Graph pruning experiment: We check if graph pruning is able to decrease the node imbalance ratio and to decrease the graph learning time.

(3) Comparison experiment: We compare SeGDroid with related works.

(4) Discussion experiment: We mainly discuss the performance of SeGDroid using different graph learning algorithms.

On each dataset, 80% is used as training set, 20% used as testing set. In the training set, 80% is used for training the model and 20% for validation. All experiments are performed on the server with the following environment: (1) operating system: Linux-3.10.0–957.el7.x86_64-x86_64-with-centos-7.6.1810-Core; (2) GPU: Tesla V100-PCIE-32 GB. We adopt Androidguard to build the FCGs and extract the APIs and opcodes from APKs. The DGL library (Wang et al., 2019) is used for implementing graph learning algorithms.

Our graph pruning method aims at building sensitive FCGs. Using graph pruning, the pruned graph may only have a few nodes if the corresponding complete graph originally has a small number of nodes. To preserve nodes for the small graphs, we utilize a threshold λ to check whether graph pruning is performed on a graph or not. The threshold λ is empirically set as 8000. That is, if the number of nodes is higher than 8000 on an FCG, we implement graph pruning; otherwise, we will not perform graph pruning on it. The hyperparameters of GraphSAGE are set as follows: (1) the number of convolutional layers: 2; (2) the number of nodes in each layer [64, 32]; (3) learning rate: 0.001; (4) optimizer: Adam; and (5) the number of epochs: 100.

In the following experiments, the accuracy, F-score, recall, and precision metrics are applied to evaluate the performance of different methods. The Acc. denotes the accuracy, Prec.(m), Rec.(m) and F-score(m) respectively denote the malware class's precision, recall and F-score; Prec.(b), Rec.(b) and F-score(b) respectively denote the benign class's precision, recall and F-score. The best performance is highlighted in bold in the tables of experimental results.

4.2.1. Ablation experiment

Our work's contributions mainly include graph pruning (denoted by P) and node representation, which includes node embedding (denoted by E) and vector weighting with centrality (denoted by W) parts. To analyze the contributions of the three parts, we carried out experiments to evaluate the performance of the variant models of SeGDroid. These variant models are illustrated as below. The symbol "−" means removing. For example, SeGDroid-P-E-W means that the P, E and W parts are removed from the original SeGDroid.

(1) SeGDroid-P-E-W: this model does not apply the three parts of P, E and W. That is, the input data are the complete graphs associated with the raw features for nodes (the occurrence of APIs and opcodes used in Vinayaka and Jaidhar (2021))

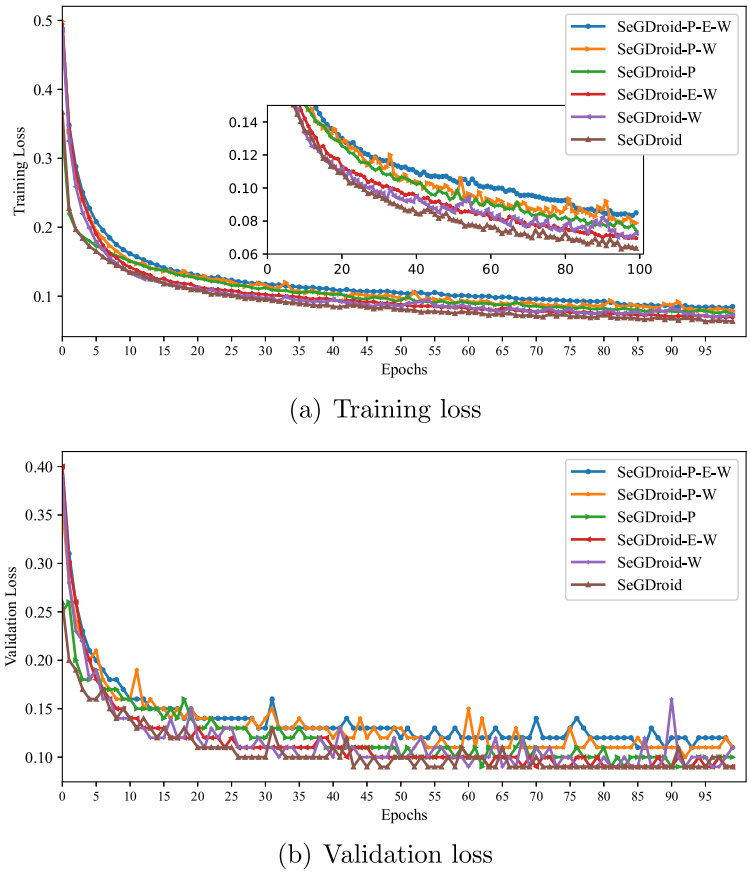


Fig. 8. The training loss and validation loss of variant models.

(2) SeGDroid-P-W: this model only adopts the E part. That is, the input data are the complete graphs associated with node features obtained by our node embedding method.

(3) SeGDroid-P: this model adopts the E and W parts. That is, the input data are the complete graphs associated with node features obtained by our node representation method.

(4) SeGDroid-E-W: this model adopts the P part. That is, the input data are the pruned graphs associated with the raw features for nodes.

(5) SeGDroid-W: this model adopts the P and E parts. That is, the input data are the pruned graphs associated with node feature vectors obtained by our node embedding method.

The training and validation losses of variant models are shown in Fig. 8. Fig. 8(a) shows that SeGDroid obtains the lowest training loss. The models that adopt graph pruning achieve less training loss (approximately 0.064 best among the models taking pruned graphs as input) than those with complete graphs (approximately 0.074 best among the models taking complete graphs as input). Fig. 8(b) shows that SeGDroid obtains the lowest validation loss (approximately 0.09). We can see that the validation loss of SeGDroid is much smaller than that of SeGDroid-W at the first epoch, and the validation loss of SeGDroid is more stable than that of SeGDroid-W when the epochs are higher. This demonstrates that the feature vector weighting with centrality is able to accelerate the convergence of graph learning.

The malware detection performance of different models in terms of accuracy, precision, recall and F-score is shown in Table 2. The experimental results are analyzed from the following three aspects.

(1) When analyzing the performance of the graph pruning of SeGDroid, we compare the two models in each pair of (SeGDroid-P-E-W, SeGDroid-E-W), (SeGDroid-P-W, SeGDroid-W), and (SeGDroid-P, SeGDroid). Between the two models in each pair, the first one does not implement graph pruning, but the second one implements graph pruning. According to Table 2, the model that implements graph pruning can further decrease the loss and improve the malware detection performance in most cases. For example, when compared with SeGDroid-P, SeGDroid improves the F-score of the malware class from 98.39% to 98.71%, and improves the F-score of the benign class from 95.09% to 96.11%.

(2) When analyzing the performance of the API and opcode embedding part, we compare the two models in each pair of (SeGDroid-P-E-W, SeGDroid-P-W) and (SeGDroid-E-W, SeGDroid-W). The results show that the loss of SeGDroid-P-W is less than that of SeGDroid-P-E-W. In addition, the malware F-score of SeGDroid-W (98.46%) is higher than that of SeGDroid-E-W (98.13%).

(3) When analyzing the performance of vector weighting with centrality, we compare the two models in each pair of (SeGDroid-P-W, SeGDroid-P) and (SeGDroid-W,SeGDroid). We observed that SeGDroid outperforms SeGDroid-W in terms of accuracy, precision and F-score. Similarly, when using the complete graph, the model (SeGDroid-P) that relies on the centrality measure outperforms the model(SeGDroid-P-W) without any centrality measures.

4.2.2. *Graph pruning experiment*

This paper proposes a graph pruning method based on sensitive APIs. It aims to preserve the context of invoking sensitive APIs and reduce the number of nodes in a graph. The above section proves that graph pruning is able to improve malware detection performance. This section further evaluates the performance of graph pruning in the case of decreasing node imbalance and decreasing model training time.

Regarding handling the node imbalance, we compare the number of nodes before and after graph pruning, as shown in Fig. 9. Benign-A denotes the results after graph pruning, and Benign-B denotes the results before pruning the graph. The meaning of other *xtick* labels is similar. It shows that the number of nodes of the benign samples

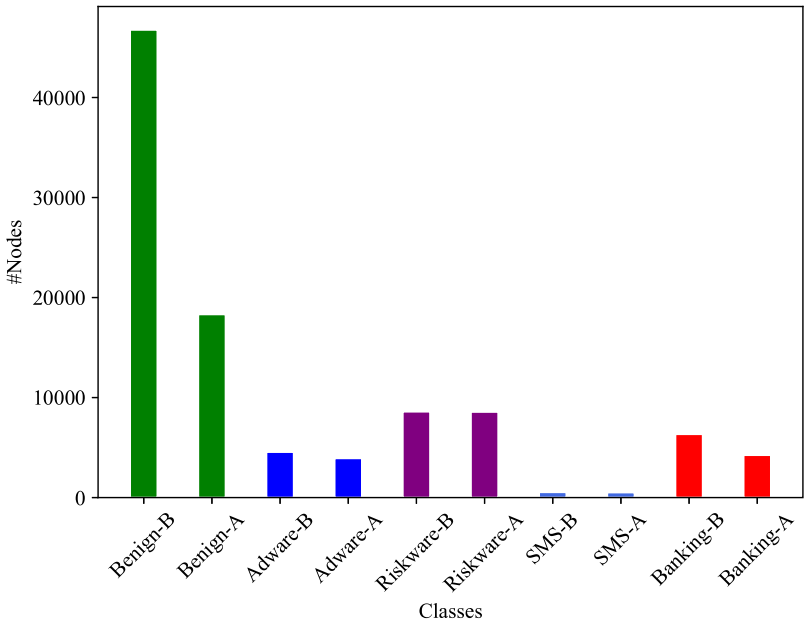


Fig. 9. The number of nodes before and after graph pruning.

Table 2
The malware detection results of variant models.

Methods	Acc.	Prec.(m)	Rec.(m)	F-score(m)	Prec.(b)	Rec.(b)	F-score(b)
SeGDroid-P-E-W	0.9715	0.9808	0.9816	0.9812	0.9423	0.9399	0.9411
SeGDroid-P-W	0.9732	0.9832	0.9815	0.9824	0.9432	0.9482	0.9457
SeGDroid-P	0.9757	0.9874	0.9804	0.9839	0.9408	0.9612	0.9509
SeGDroid-E-W	0.9715	0.9732	**0.9896**	0.9813	**0.9657**	0.9149	0.9396
SeGDroid-W	0.9768	0.9841	0.9852	0.9846	0.9540	0.9054	0.9522
SeGDroid	**0.9807**	**0.9931**	0.9812	**0.9871**	0.9439	**0.9790**	**0.9611**

significantly decreased. Before graph pruning, the benign samples have approximately 46,795 nodes on average. After graph pruning, the benign samples have approximately 18,315 nodes on average. The most significant number of nodes in benign samples decreases from 250,000 to 120,000. The node imbalance ratio decreases from 10.3 to 4.2. The node imbalance ratio is calculated as the ratio between the number of nodes in benign samples and malware samples.

Regarding decreasing the time consumption of graph learning, we conduct our experiments to compare the time consumption of training the GNN on the data with or without graph pruning. The time consumption performance is shown in Fig. 10. We mainly compare the following three pairs of models: (SeGDroid-P-E-W, SeGDroid-E-W), (SeGDroid-P-W, SeGDroid-W), and (SeGDroid-P, SeGDroid). In each pair, the only difference between the two models is if the graph pruning is used. Fig. 10 shows that the graph pruning can further decrease the time consumption of training the graph learning model. This is because the number of nodes is significantly decreased after graph pruning.

Next, we further analyze the performance of graph pruning with different parameter values. We adopt a threshold λ in graph pruning to preserve the node information for apps with a small number of nodes. If the number of nodes in an app is higher than λ, it will be handled by graph pruning. The malware detection performance of the SeGDroid with different thresholds in graph pruning is shown in Table 3. The smaller the value of λ, the more graphs will be pruned. The results show that there is no trend that the smaller the value of λ is, the better of the performance obtained by graph pruning. When further analyzing the results, we found that the graph pruning may decrease the performance on the graphs with a small number of nodes. The model with the λ of 8000 performs the best in terms of accuracy and F-score. Therefore, we empirically set λ as 8000 in our experiments.

Table 3
The results of SeGDroid with different thresholds of graph pruning.

λ	Acc.	Prec.(m)	Rec.(m)	F-score(m)	Prec.(b)	Rec.(b)	F-score(b)
2000	0.9782	0.9817	**0.9896**	0.9856	**0.9667**	0.9426	0.9545
4000	0.9765	0.9907	0.9780	0.9843	0.9345	0.9715	0.9527
6000	0.9780	0.9864	0.9844	0.9854	0.9520	0.9579	0.9550
8000	**0.9807**	**0.9931**	0.9812	**0.9871**	0.9439	**0.9790**	**0.9611**
10000	0.9780	0.9822	0.9888	0.9855	0.9646	0.9442	0.9543
12000	0.9749	0.9844	0.9824	0.9834	0.9458	0.9517	0.9487

4.2.3. Comparison experiments

(1) *Results on CIC dataset*

The novel aspect of SeGDroid is the feature learning based on sensitive FCGs. This section mainly compares the feature vectors obtained by SeGDroid with those obtained by previous works, including Permission, MaMaDroid (Onwuzurike et al., 2019), Malscan (Wu, Li, et al., 2019) and GraphSAGE-Occ (Vinayaka & Jaidhar, 2021). Permission is the feature set of required permissions. Malscan and MaMaDroid are generally used with machine learning algorithms for malware detection. Malscan achieves better performance when it applies 1NN(1 Nearest Neighbor) according to the results in Wu, Li, et al. (2019). MaMaDroid performs better when it adopts random forest according to the results in Onwuzurike et al. (2019). Our empirical results show that Permission achieves better performance when it utilizes 3NN(3 Nearest Neighbor); SeGDroid performs better when it adopts random forest. Therefore, the selected machine learning algorithms are 3NN, random forest, 1NN, random forest for Permission, MaMaDroid, Malscan and SeGDroid respectively. In SeGDroid, we train a classification model on the vectors obtained by graph embedding. The binary classification results on CIC dataset are shown in Table 4.

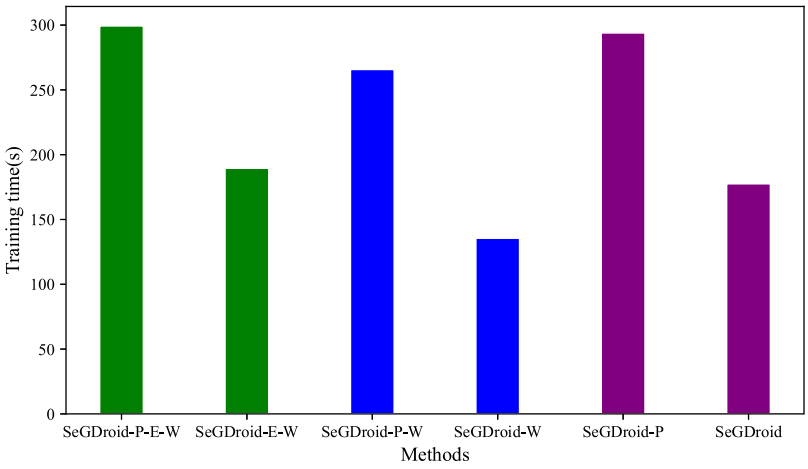


Fig. 10. The training time of different models.

Table 4
Binary classification results obtained by different methods.

Methods	Acc.	Prec.(m)	Rec.(m)	F-score(m)	Prec.(b)	Rec.(b)	F-score(b)
Permission	0.9470	0.8380	0.9690	0.8987	**0.9895**	0.9400	0.9641
MaMaDroid	0.9645	0.9717	0.9817	0.9767	0.9410	0.9107	0.9256
MalScan	0.9789	0.9857	0.9865	0.9861	0.9577	0.9553	0.9565
GraphSAGE-Occ	0.9049	0.9357	0.8696	0.9014	0.8782	0.9402	0.9081
SeGDroid	**0.9837**	**0.9889**	**0.9896**	**0.9892**	0.9677	**0.9653**	**0.9665**

MaMaDroid is also based on the FCG for extracting the feature vectors. MalScan also weights features through centrality measures after combining all sensitive APIs. GraphSAGE-Occ utilizes GraphSAGE to learn the feature vector from the FCG, in which the occurrence of APIs and opcodes represents each node. In addition, GraphSAGE-Occ includes a node balance method with the objective of balancing the number of nodes between benign and malware. Those methods share the source code in public. We implement the experiments based on their public shared codes.

Table 4 shows that SeGDroid performs the best. It obtains 98.37% accuracy, 98.92% F-score for the malware class and 96.65% F-score for the benign class. It improves the performance of GraphSAGE-Occ from 90.49% to 98.37%. We also found that the node balance in GraphSAGE-Occ would decrease the malware detection performance. GraphSAGE-Occ without node balance is the same as SeGDroid-P-E-W, as shown in Table 2. SeGDroid-P-E-W utilizes the complete graph and nodes are characterized by the occurrence of APIs and opcodes, which are the same as Vinayaka and Jaidhar (2021). This implies that the node balance method that removes samples may decrease malware detection performance. MalScan also utilizes the centrality for malware detection but without using graph pruning for simplifying the graphs and the word2vec for semantic node representation. The results in Table 4 show that SeGDroid also outperforms Malscan in malware detection, especially in terms of the F-score of the benign class.

Concerning category classification on the CIC dataset, the classification results are shown in Fig. 11. Similarly, SeGDroid also performs the best among these methods. SeGDroid obtains 95.29% accuracy. It achieves 96.26%, 94.44%, 89.99%, 93.53%, and 98.65% F-score for the Benign, Adware, Banking, Riskware and SMS classes, respectively. When compared with Permission, MaMaDroid,Malscan and GraphSAGE-Occ, SeGDroid improves F-score by about 6.94%, 11.68%, 0.37% and 4.37% respectively on average.

(2) *Results on MalRadar dataset*

To evaluate our method in more cases, we further perform experiments on MalRadar dataset. Similarly, the binary classification and multiclass classification results will be discussed in this section. Also, SeGDroid is compared with Permission, MaMaDroid (Onwuzurike et al., 2019), Malscan (Wu, Li, et al., 2019) and GraphSAGE-Occ (Vinayaka & Jaidhar, 2021).

To carry out the binary classification experiment, all malware samples shown in Table 1 are combined into malware class. The benign samples from AndroZoo are in benign class. The binary classification results are shown in Table 5. The results show that SeGDroid obtains 98.13% accuracy, 98.75% F-score for malware class and 96.33% F-score for benign class. It obtains comparable performance when compared with MalScan. It outperforms SAGEGraph-Occ that also utilizes the graph learning method for Android malware detection.

To further analyze the performance of our method on fine-grained malware detection (i.e. malware family identification). The multiclass classification results on MalRadar dataset are shown in Fig. 12. The results show that SeGDroid achieves the best accuracy (97.01%). There are 15 malware families and a benign class in the MalRadar dataset. There is no model that obtains the best performance on all classes in terms of precision, recall and F-score. The average values for the three metrics are shown in Table 6 to assess the performance of different models on malware family classification. Table 6 shows that SeGDroid obtains the best precision (0.9619) and F-score(0.9548) on average. SeGDroid improves F-score about 8.12%,7.25%,47.68% on average when compared with MaMaDroid, MalScan and GraphSAGE-Occ respectively.

The multiclass classification is more complex than binary classification. The classification performance metrics of all models are lower than those obtained in the case of binary classification. The performance metrics of some methods are reduced much, such as GraphSAGE-Occ. The possible reason is that the class imbalance problem exists among those families. The performance on some families (such as GhostClicker and Joker) is not good. The class imbalance problem in FCG embedding is an interesting work that we will research in the future.

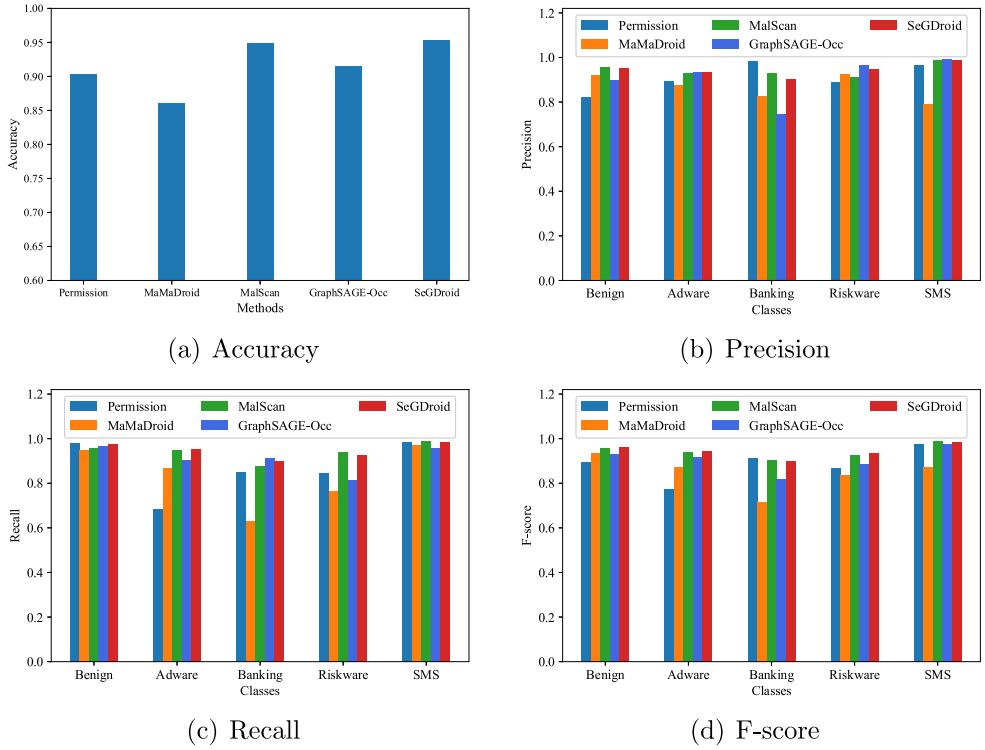


Fig. 11. The results of category classification on CIC.

Table 5
The binary classification results on MalRadar dataset.

Methods	Acc.	Prec.(m)	Rec.(m)	F-score(m)	Prec.(b)	Rec.(b)	F-score(b)
Permission	0.9739	0.9723	**0.9933**	0.9827	**0.9791**	0.9167	0.9468
MaMaDroid	0.9727	0.9866	0.9867	0.9816	0.9333	0.9608	0.9469
MalScan	**0.9813**	0.9867	0.9883	**0.9875**	0.9653	0.9606	0.9630
GraphSAGE-Occ	0.9763	0.9882	0.9800	0.9841	0.9423	0.9655	0.9538
SeGDroid	**0.9813**	**0.9899**	0.9850	**0.9875**	0.9563	**0.9704**	**0.9633**

Table 6
The average metric values among these malware families.

Metrics	Permission	MaMaDroid	MalScan	GraphSAGE-Occ	SeGDroid
Precision	0.9409	0.9087	0.9018	0.5960	**0.9619**
Recall	**0.9633**	0.8562	0.8986	0.4888	0.9527
F-score	0.9511	0.8736	0.8824	0.4768	**0.9548**

4.3. Discussion on SeGDroid using different GNN algorithms

In our experiments, GraphSAGE is used for graph embedding. In this section, we discuss the performance of SeGDroid using different GNN algorithms for graph embedding, including: GCN (Graph Convolutional Networks) (Zhang et al., 2022), GraphSAGE (Hamilton et al., 2017), DotGAT (dot product version of self attention in Graph Attention Network) (Velickovic et al., 2017), TAG (Topology Adaptive Graph) (Du et al., 2017), and SGC (Simplifying Graph Convolutional Networks) (Wu, Souza, et al., 2019). The binary classification results are shown in Table 7. The results are consistent with the results reported in Vinayaka and Jaidhar (2021). GraphSAGE performs the best among the GNN algorithms. GraphSAGE obtains 98.07% accuracy, 98.71% F-score for malware and 96.11% F-score for benign apps.

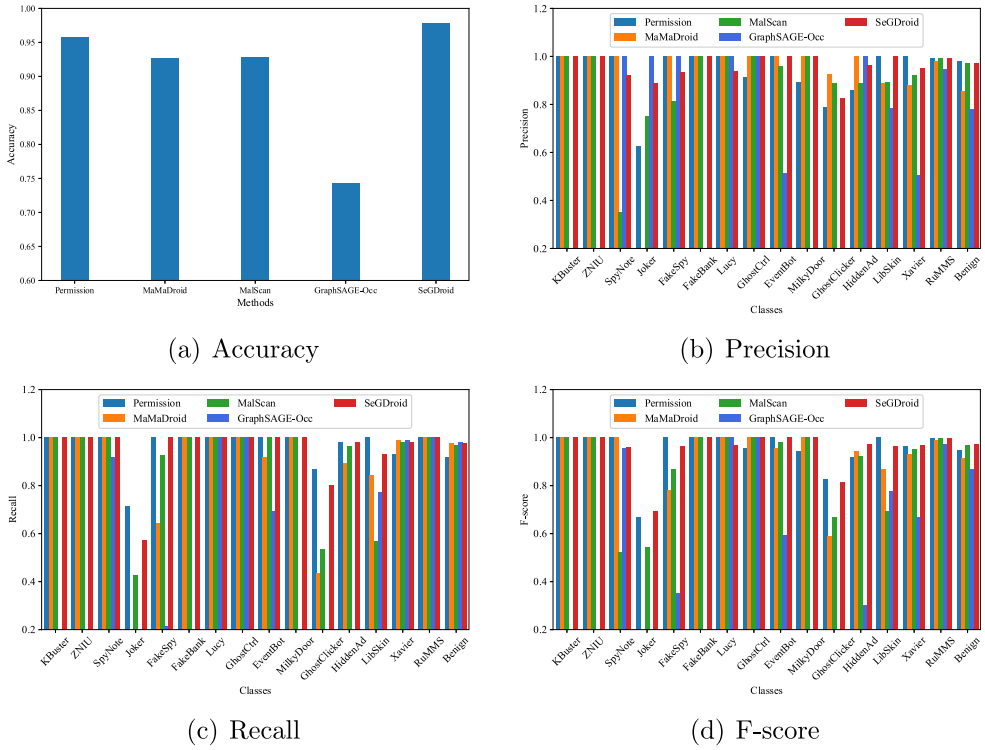
Next, we carry out experiments of category classification on the CIC datasets, as shown in Fig. 13. The results show that GraphSAGE also achieves the best accuracy (92.3%), 95.24%, 92.71%, 81.21%, 89.78% and 97.63% F-scores for the Benign, Adware, Banking, Riskware and SMS classes, respectively. This is because GraphSAGE is an inductive framework that leverages node attribute information to generate representations on previously unseen data efficiently. With the dynamic nature of Android malware, the graphs in testing data may have some unseen nodes. Therefore, this paper utilizes GraphSAGE for graph embedding of FCGs in our experiments.

4.4. Model explanation

To explain SeGDroid model, we propose a method to visualize the importance of each graph node (the functions in an app). According to the model visualization method proposed in Section 3.6, the visualization first depicts the nodes' importance and then summarizes the functions with high importance values.

On the CIC dataset, taking SMS malware as an example, the importance vector of each node is shown in Fig. 14. The results show that the android.telephony.SmsMananger.getDefault and android.telephony.SmsManager.sendTextMessage achieve higher values than other functions. This denotes that the two functions are more likely to relate to malicious behavior.

We summarize the functions of the nodes with higher importance values in SMS malware, as shown in Table 8. First, we rank the nodes according to the importance value and select the top 10 nodes in each malware. We further rank the selected nodes according to their selected frequency, and then the top 10 nodes are summarized in Table 8. The



(a) Accuracy (b) Precision

(c) Recall (d) F-score

Fig. 12. The family classification results on the MalRadar dataset.

Table 7
The binary classification results obtained by different graph learning algorithms.

Methods	Acc.	Prec.(m)	Rec.(m)	F-score(m)	Prec.(b)	Rec.(b)	F-score(b)
GCN	0.9710	0.9882	0.9732	0.9807	0.9208	0.9641	0.9420
DotGAT	0.9257	0.9800	0.9205	0.9493	0.7927	0.9418	0.8609
TAG	0.9586	0.9736	0.9716	0.9726	0.9127	0.9183	0.9155
SGC	0.9372	0.9682	0.9481	0.9580	0.8488	0.9035	0.8753
GraphSAGE	**0.9807**	**0.9931**	**0.9812**	**0.9871**	**0.9439**	**0.9790**	**0.9611**

Table 8
The APIs of the top 10 nodes.

ClassName	FunctionName	Explanation
Landroid/telephony/SmsManager	sendTextMessage	Send a text-based SMS.
Landroid/telephony/SmsManager	getDefault	Get the SmsManager associated with the default subscription id.
Landroid/view/animation/Animation	setDuration	Sets the duration of the animation
Landroid/view/animation/TranslateAnimation	<init>	Initialize this animation
Landroid/telephony/TelephonyManager	getLine1Number	Get the phone number
Landroid/telephony/TelephonyManager	getNetworkOperator	Get the MCC+MNC of current registered operator
Landroid/telephony/TelephonyManager	getSimOperator	Get the MCC+MNC of the provider of the SIM
Landroid/telephony/SmsMessage	createFromPdu	Create a SmsMessage from a raw PDU
Landroid/telephony/SmsMessage	getOriginatingAddress	Get the originating address(sender) of this SMS message
Landroid/widget/Toast	makeText	Make a standard toast

description of these APIs is available from their documentation (Google, 2022).

This benefits tracing the malicious functions of Android malware. The sendTextMessage of the SmsManager class is usually used to send a text-based SMS. SMS malware is any malicious software delivered to victims by text messaging. It involves the malicious usage of TelephonyMnager (acquiring the SIM information of mobile devices) and SmsManager (sending malicious messages such as malicious links to mobile devices). In addition, the SmsManager and TelephonyManager with high occurrence frequency in Table 8 are also the sensitive APIs found by PScout (Androguard, 2022). This further demonstrates that malicious behaviors invoke sensitive APIs with high probability.

5. Conclusion and future work

This paper proposes a novel android malware detection method named SeGDroid. It aims at learning semantic features from the sensitive FCGs. Our method firstly builds the FCG from the Smali codes, and then extracts the sensitive FCG using our proposed graph pruning method. Regarding node representation, we build API2vec and opcode2vec to embed the external and internal nodes, respectively, and to weight feature vectors at the base of the centrality measure. The resulting sensitive FCGs associated with its node features are embedded by the GraphSAGE algorithm. To provide a mechanism for understanding malicious behaviors, we propose an explanation method for our model.

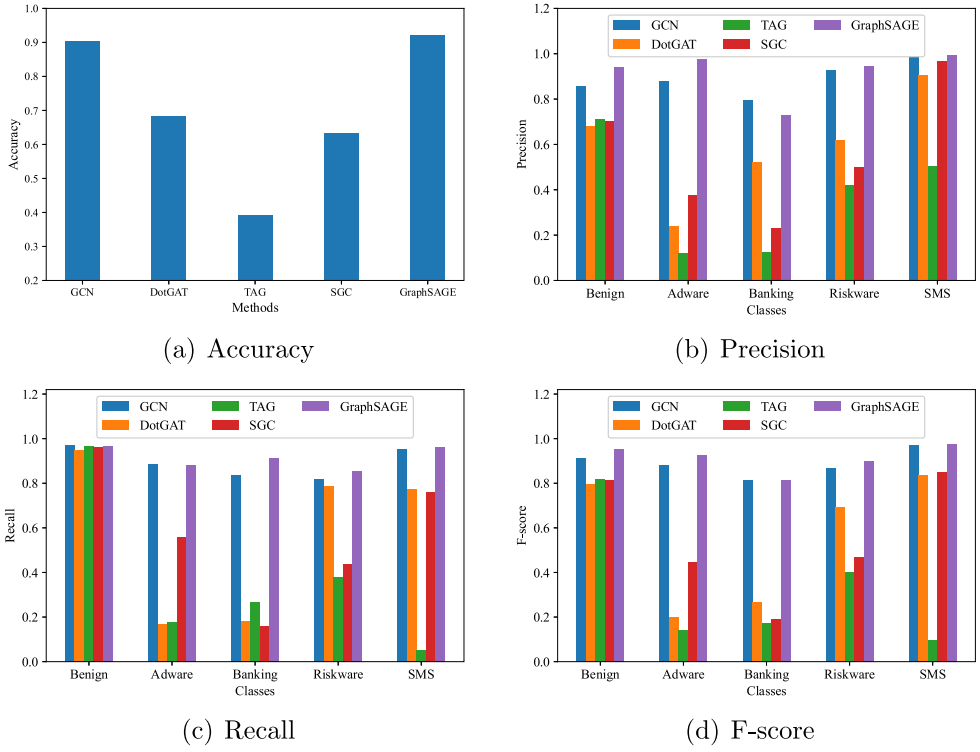


Fig. 13. The category classification results obtained by different graph learning algorithms.

Fig. 14. The visualization of the importance of each graph node in a SMS malware.

The experimental results on the CICMal2020 and MalRadar datasets are summarized below.

(1) The ablation experiments show that each of our proposals – graph pruning and node representation (node embedding and vector weighting) – can further improve malware detection performance.

(2) The graph pruning experiments show that the graph pruning indeed decreases the node imbalance ratio and decreases the time consumption of training the graph embedding model.

(3) The comparison results show that SeGDroid outperforms previous works in terms of accuracy and F-score. It achieves an F-score of 98% in the case of malware detection, and 96% in the case of family classification on average.

(4) Using our model explanation method, we visualized the importance values of FCG nodes. The malicious APIs can be highlighted by our method. It is helpful for understanding the malicious behavior.

A limitation of this paper is that it does not consider the concept drift and class imbalance problems in FCG learning. In future work, we will further research how to improve the performance of SeGDroid when confronting these two problems.

CRediT authorship contribution statement

Zhen Liu: Conceptualization, Methodology, Writing – original draft, Formal analysis, Software, Writing – review & editing. **Ruoyu Wang:** Conceptualization, Methodology, Visualization, Software, Writing – review & editing. **Nathalie Japkowicz:** Conceptualization, Writing – review & editing. **Heitor Murilo Gomes:** Conceptualization, Writing – review & editing. **Bitao Peng:** Funding acquisition, Writing – review & editing. **Wenbin Zhang:** Resources, Writing – review & editing.

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.

Data availability

Our experimental datasets have been shared in public by other research institutes. The source of these datasets can be found in our manuscript.

Acknowledgments

We thank the anonymous reviewers for their constructive comments. This work is supported by the Starting Research Fund from the Guangdong University of Foreign Studies [Grant No. 2022RC049], Science and Technology Projects of Guangzhou [Grant No. 202201010100], Key Research Platforms and Projects of Colleges and Universities in Guangdong Province [Grant No. 2020ZDZX3060], National Natural Science Foundation of China [Grant No. 61501128].

References

Alhanahnah, M., Yan, Q., Bagheri, H., Zhou, H., Tsutano, Y., Srisa-an, W., & Luo, X. (2020). DINA: Detecting hidden android inter-app communication in dynamic loaded code. *IEEE Transactions on Information Forensics and Security*, *15*, 2782–2797.

Alzaylaee, M. K., Yerima, S. Y., & Sezer, S. (2020). DL-droid: Deep learning based android malware detection using real devices. *Computers & Security*, *89*.

Ananya, A., Aswathy, A., Amal, T. R., Swathy, P. G., Vinod, P., & Shojafar, M. (2020). SysDroid: A dynamic ML-based android malware analyzer using system call traces. *Cluster Computing*, *23*(4), 2789–2808.

Androguard (2022). Androguard. https://github.com/androguard/androguard.

Android Statistics (2022). Android statistics. https://www.businessofapps.com/data/android-statistics/.

Arora, S., Liang, Y., & Ma, T. (2017). A simple but tough-to-beat baseline for sentence embeddings. In *5th International conference on learning representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference track proceedings* (pp. 1–16). OpenReview.net.

Au, K. W. Y., Zhou, Y. F., Huang, Z., & Lie, D. (2012). PScout: Analyzing the android permission specification. In T. Yu, G. Danezis, & V. D. Gligor (Eds.), *The ACM conference on computer and communications security* (pp. 217–228). ACM.

Badhani, S., & Muttoo, S. K. (2019). CENDroid - A cluster-ensemble classifier for detecting malicious android applications. *Computers & Security*, *85*, 25–40.

Cai, M., Jiang, Y., Gao, C., Li, H., & Yuan, W. (2021). Learning features from enhanced function call graphs for Android malware detection. *Neurocomputing*, *423*, 301–307.

D'Angelo, G., Ficco, M., & Palmieri, F. (2020). Malware detection in mobile environments based on autoencoders and API-images. *Journal of Parallel and Distributed Computing*, *137*, 26–33.

Du, J., Zhang, S., Wu, G., Moura, J. M. F., & Kar, S. (2017). Topology adaptive graph convolutional networks. CoRR abs/1710.10370.

Freeman, L. C. (1978). Centrality in social networks conceptual clarification. *Social Networks*, *1*(3), 215–239.

Gao, H., Cheng, S., & Zhang, W. (2021). GDroid: Android malware detection and classification with graph convolutional network. *Computers & Security*, *106*, Article 102264.

Google (2022). Android developers. https://developer.android.google.cn/.

Grace, M. C., Zhou, Y., Zhang, Q., Zou, S., & Jiang, X. (2012). RiskRanker: Scalable and accurate zero-day android malware detection. In N. Davies, S. Seshan, & L. Zhong (Eds.), *The 10th international conference on mobile systems, applications, and services* (pp. 281–294).

Guerra-Manzanares, A., Bahsi, H., & Nõmm, S. (2021). KronoDroid: Time-based hybrid-featured dataset for effective android malware detection and characterization. *Computers & Security*, *110*, Article 102399.

Guerra-Manzanares, A., Luckner, M., & Bahsi, H. (2022). Concept drift and cross-device behavior: Challenges and implications for effective android malware detection. *Computers & Security*, *120*, Article 102757.

GuimerÃ, R., Mossa, S., Turtschi, A., & Amaral, L. A. N. (2005). The worldwide air transportation network: Anomalous centrality, community structure, and cities' global roles. *Proceedings of the National Academy of Sciences*, *102*(22), 7794–7799.

Hamilton, W. L., Ying, Z., & Leskovec, J. (2017). Inductive representation learning on large graphs. In I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, & R. Garnett (Eds.), *Advances in neural information processing systems 30: Annual conference on neural information processing systems 2017* (pp. 1024–1034).

Hou, S., Ye, Y., Song, Y., & Abdulhayoglu, M. (2017). 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* (pp. 1507–1515). ACM.

Kabakus, A. T. (2022). DroidMalwareDetector: A novel android malware detection framework based on convolutional neural network. *Expert Systems with Applications*, *206*, Article 117833.

Khan, K. N., Khan, M. S., Nauman, M., & Khan, M. Y. (2022). Op2Vec: An opcode embedding technique and dataset design for end-to-end detection of android malware. *Security and Communication Networks*, *2022*, Article 3710968.

Kipf, T. N., & Welling, M. (2017). Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings* (pp. 1–14). OpenReview.net.

Kong, K., Zhang, Z., Yang, Z., & Zhang, Z. (2022). FCSCNN: Feature centralized Siamese CNN-based android malware identification. *Computers & Security*, *112*, Article 102514.

Lei, T., Qin, Z., Wang, Z., Li, Q., & Ye, D. (2019). EveDroid: Event-aware android malware detection against model degrading for IoT devices. *IEEE Internet of Things Journal*, *6*(4), 6668–6680.

Li, Q., Chen, Z., Yan, Q., Wang, S., Ma, K., Shi, Y., & Cui, L. (2018). MulAV: Multilevel and explainable detection of android malware with data fusion. In J. Vaidya, & J. Li (Eds.), *Lecture notes in computer science*: *vol. 11337*, *Algorithms and architectures for parallel processing - 18th International conference, ICA3PP 2018, Guangzhou, China, November 15-17, 2018, proceedings, part IV* (pp. 166–177). Springer.

Lin, K., Xu, X., & Xiao, F. (2022). MFFusion: A multi-level features fusion model for malicious traffic detection based on deep learning. *Computer Networks*, *202*, Article 108658.

Liu, Z., Wang, R., Japkowicz, N., Tang, D., Zhang, W., & Zhao, J. (2021). Research on unsupervised feature learning for Android malware detection based on restricted Boltzmann machines. *Future Generation Computer Systems*, *120*, 91–108.

Lo, W. W., Layeghy, S., Sarhan, M., Gallagher, M., & Portmann, M. (2022). Graph neural network-based Android malware classification with jumping knowledge. In *2022 IEEE conference on dependable and secure computing* (pp. 1–9).

Mahdavifar, S., Alhadidi, D., & Ghorbani, A. A. (2022). Effective and efficient hybrid android malware classification using pseudo-label stacked auto-encoder. *Journal of Network and Systems Management*, *30*(1), 22.

Martín, A., Lara-Cabrera, R., & Camacho, D. (2019). Android malware detection through hybrid features fusion and ensemble classifiers: The AndroPyTool framework and the OmniDroid dataset. *Information Fusion*, *52*, 128–142.

Meng, G., Feng, R., Bai, G., Chen, K., & Liu, Y. (2018). DroidEcho: An in-depth dissection of malicious behaviors in Android applications. *Cybersecurity*, *1*(1), 1–17.

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013a). Efficient estimation of word representations in vector space. In Y. Bengio, & Y. LeCun (Eds.), *1st International conference on learning representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop track proceedings* (pp. 1–12).

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013b). Efficient estimation of word representations in vector space. In Y. Bengio, & Y. LeCun (Eds.), *1st International conference on learning representations* (pp. 1–12).

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. In C. J. C. Burges, L. Bottou, Z. Ghahramani, & K. Q. Weinberger (Eds.), *Advances in neural information processing systems 26: 27th Annual conference on neural information processing systems* (pp. 3111–3119).

Naït-Abdesselam, F., Darwaish, A., & Titouna, C. (2020). An intelligent malware detection and classification system using apps-to-images transformations and convolutional neural networks. In *16th International conference on wireless and mobile computing, networking and communications* (pp. 1–6). IEEE.

Newman, M. E. J. (2010). *Networks: An introduction*. Oxford University Press.

Onwuzurike, L., Mariconti, E., Andriotis, P., Cristofaro, E. D., Ross, G. J., & Stringhini, G. (2019). MaMaDroid: Detecting android malware by building Markov chains of behavioral models (extended version). *ACM Transactions on Privacy and Security*, *22*(2), 14:1–14:34.

Ou, F., & Xu, J. (2022). S^3feature: A static sensitive subgraph-based feature for android malware detection. *Computers & Security*, *112*, Article 102513.

Qiu, J., Han, Q., Luo, W., Pan, L., Nepal, S., Zhang, J., & Xiang, Y. (2023). Cyber code intelligence for android malware detection. *IEEE Transactions on Cybernetics*, *53*(1), 617–627. http://dx.doi.org/10.1109/TCYB.2022.3164625.

Razgallah, A., Khoury, R., Hallé, S., & Khanmohammadi, K. (2021). A survey of malware detection in Android apps: Recommendations and perspectives for future research. *Computer Science Review*, *39*, Article 100358.

Rong, X. (2014). Word2vec parameter learning explained. CoRR abs/1411.2738.

Scalas, M., Maiorca, D., Mercaldo, F., Visaggio, C. A., Martinelli, F., & Giacinto, G. (2019). On the effectiveness of system API-related information for Android ransomware detection. *Computers & Security*, *86*, 168–182.

Shar, L. K., Demissie, B. F., Ceccato, M., & Minn, W. (2020). Experimental comparison of features and classifiers for Android malware detection. In *Proceedings of the IEEE/ACM 7th international conference on mobile software engineering and systems* (pp. 50–60).

Sun, B., Ban, T., Chang, S., Sun, Y. S., Takahashi, T., & Inoue, D. (2019). A scalable and accurate feature representation method for identifying malicious mobile applications. In C. Hung, & G. A. Papadopoulos (Eds.), *Proceedings of the 34th ACM/SIGAPP symposium on applied computing* (pp. 1182–1189). ACM.

Tang, J., Li, R., Jiang, Y., Gu, X., & Li, Y. (2022). Android malware obfuscation variants detection method based on multi-granularity opcode features. *Future Generation Computer Systems*, *129*, 141–151.

Vasan, D., Alazab, M., Wassan, S., Naeem, H., Safaei, B., & Zheng, Q. (2020). IMCFN: Image-based malware classification using fine-tuned convolutional neural network architecture. *Computer Networking*, *171*, Article 107138.

Velickovic, P., Cucurull, G., Casanova, A., Romero, A., Lio', P., & Bengio, Y. (2017). Graph attention networks. ArXiv abs/1710.10903.

Vinayaka, K. V., & Jaidhar, C. D. (2021). Android malware detection using function call graph with graph convolutional networks. In *2021 2nd International conference on secure cyber computing and communications* (pp. 279–287). http://dx.doi.org/10.1109/ICSCCC51823.2021.9478141.

Wang, S., Chen, Z., Yan, Q., Ji, K., Peng, L., Yang, B., & Conti, M. (2020). Deep and broad URL feature mining for android malware detection. *Information Sciences*, *513*, 600–613. http://dx.doi.org/10.1016/j.ins.2019.11.008.

Wang, L., Wang, H., He, R., Tao, R., Meng, G., Luo, X., & Liu, X. (2022). MalRadar: Demystifying android malware in the new era. *Proceedings of ACM Measurement and Analysis of Computing*, *6*(2), 40:1–40:27.

Wang, M., Zheng, D., Ye, Z., Gan, Q., Li, M., Song, X., Zhou, J., Ma, C., Yu, L., Gai, Y., Xiao, T., He, T., Karypis, G., Li, J., & Zhang, Z. (2019). Deep graph library: A graph-centric, highly-performant package for graph neural networks. arXiv: Learning.

Wasserman, S., & Faust, K. (1994). *Social network analysis: Methods and applications*. Cambridge University Press.

Wu, Y., Li, X., Zou, D., Yang, W., Zhang, X., & Jin, H. (2019). MalScan: Fast market-wide mobile malware scanning by social-network centrality analysis. In *34th IEEE/ACM international conference on automated software engineering* (pp. 139–150). IEEE.

Wu, F., Souza, A. H., Jr., Zhang, T., Fifty, C., Yu, T., & Weinberger, K. Q. (2019). Simplifying graph convolutional networks. In K. Chaudhuri, & R. Salakhutdinov (Eds.), *Proceedings of machine learning research*: *vol. 97*, *Proceedings of the 36th international conference on machine learning* (pp. 6861–6871).

Wu, Y., Zou, D., Dou, S., Yang, W., Xu, D., & Jin, H. (2022). VulCNN: An image-inspired scalable vulnerability detection system. In *44th IEEE/ACM 44th international conference on software engineering* (pp. 2365–2376). ACM.

Xu, P., Eckert, C., & Zarras, A. (2021). Detecting and categorizing android malware with graph neural networks. In *SAC '21: The 36th ACM/SIGAPP symposium on applied computing* (pp. 409–412).

Zhang, H., Lu, G., Zhan, M., & Zhang, B. (2022). Semi-supervised classification of graph convolutional networks with Laplacian rank constraints. *Neural Processing Letters*, *54*(4), 2645–2656.

Zhang, N., an Tan, Y., Yang, C., & Li, Y. (2021). Deep learning feature exploration for android malware detection. *Applied Soft Computing*, *102*, Article 107069. http://dx.doi.org/10.1016/j.asoc.2020.107069.

Zheng, M., Sun, M., & Lui, J. C. S. (2013). Droid analytics: A signature based analytic system to collect, extract, analyze and associate android malware. In *12th IEEE international conference on trust, security and privacy in computing and communications* (pp. 163–171).