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.

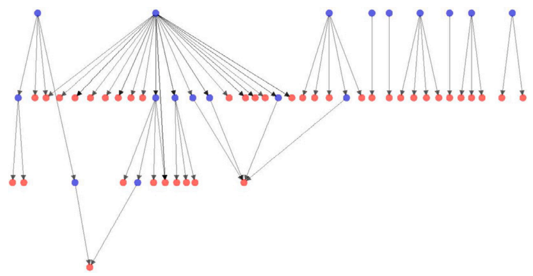
Fig. 1. The example of an FCG.

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,

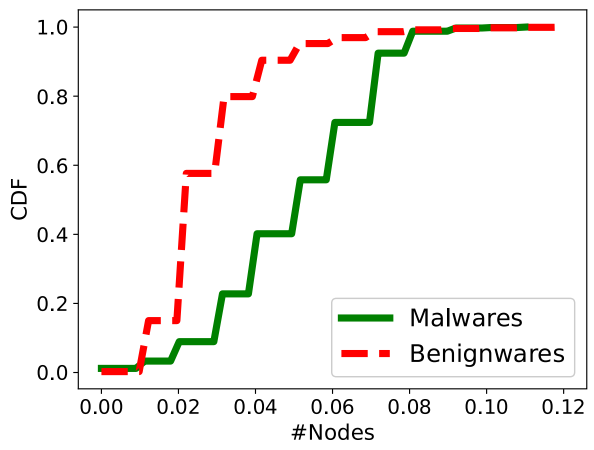

Fig. 2. The CDFs of the sensitive API ratios in malwares and benign apps.

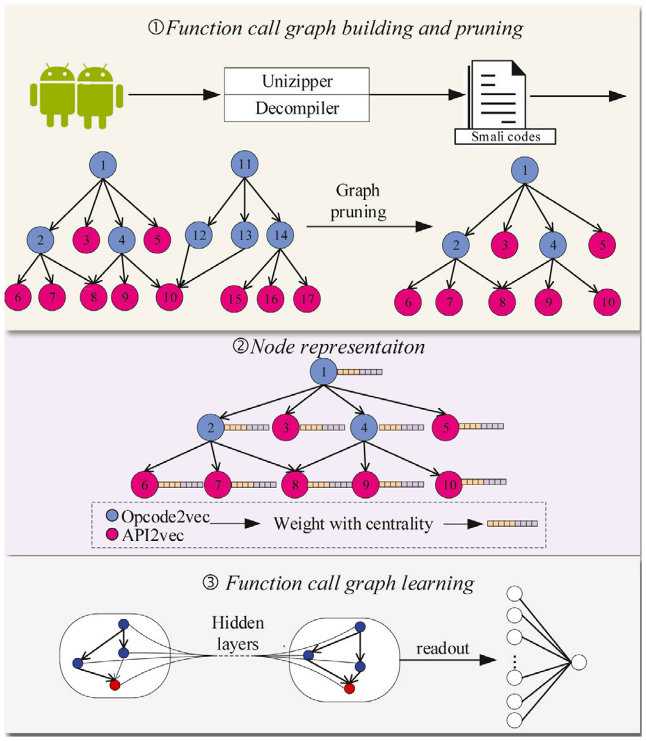

Fig. 3. The framework of SeGDroid.

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.

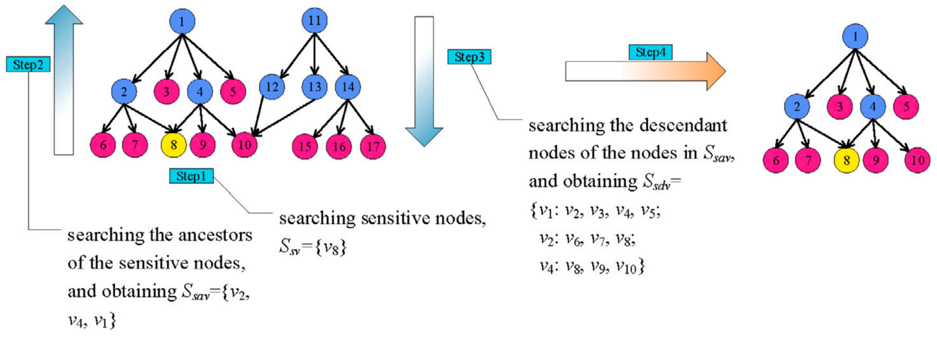
Step2
Step1
Step3
Step4
searching the ancestors of the sensitive nodes, and obtaining S_{sav}={v_2, v_4, v_1}
searching sensitive nodes, S_{sv}={v_8}
searching the descendant nodes of the nodes in S_{sav}, and obtaining S_{sdv}= {v_1: v_2, v_3, v_4, v_5; v_2: v_6, v_7, v_8; v_4: v_8, v_9, v_{10}}

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 $\mathcal{SG} = (\mathcal{V}, \mathcal{E})$
1:  for $v_i$ in $\mathcal{V}$ do
2:      if $v_i \in S_{api}$ then
3:          $S_{sv} \leftarrow S_{sv} \cup v_i$
4:      end if
5:  end for
6:  for $sv_i$ in $S_{sv}$ do
7:      addAncestor($sv_i$, $S_{sav}$, $\mathcal{G}$)
8:  end for
9:  for $sav_i$ in $S_{sav}$ do
10:     addDescendant($sav_i$, $S_{sdv}$, $\mathcal{G}$)
11: end for
12: $\mathcal{SG}$=obtainPrunedGraph($\mathcal{G}$, $S_{sdv}$,$S_{sav}$)
```

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

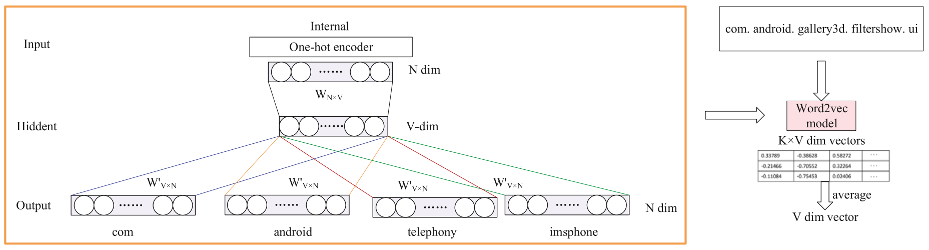

Fig. 5. The architecture of API2vec model.

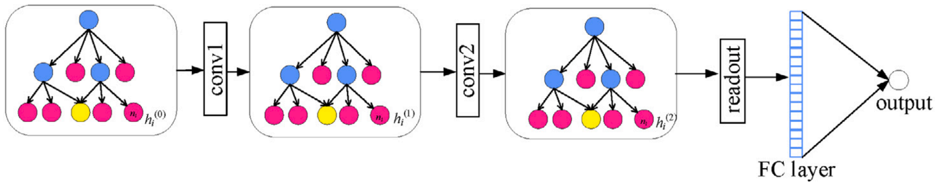

Fig. 6. The architecture of graph convolutional network model.

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

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

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

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

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

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

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

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

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:

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

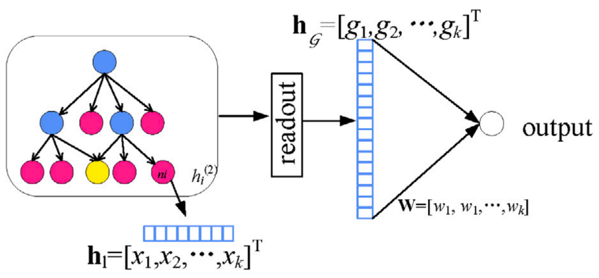
$\mathbf{h}_{\mathcal{G}}=[g_1,g_2,\cdots,g_k]^T$
readout
output
$h_i^{(2)}$
ni
$\mathbf{W}=[w_1, w_1,\cdots,w_k]$
$\mathbf{h}_1=[x_1,x_2,\cdots,x_k]^T$

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

$$\begin{aligned}&= 1/n(w_1 * (x_1^1 + \cdots + x_1^n) + \cdots + w_k * (x_k^1 + \cdots + x_k^n)) + b\\ &= 1/n((w_1 * x_1^1 + w_2 * x_2^1 + \cdots + w_k * x_k^1)\\ &+ \ldots + (w_1 * x_1^n + w_2 * x_2^n + \cdots + w_k * x_k^n)) + b \qquad (13)\end{aligned}$$

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))