Computers, Materials & Continua

Tech Science Press

DOI: 10.32604/cmc.2024.058168

ARTICLE

A Novel Approach for Android Malware Detection Based on Intelligent Computing

Manh Vu Minh* **and Cho Do Xuan**

Faculty of Information Security, Posts and Telecommunications Institute of Technology, Hanoi, 100000, Vietnam

*Corresponding Author: Manh Vu Minh. Email: manhvm@ptit.edu.vn

Received: 05 September 2024 Accepted: 28 October 2024 Published: 19 December 2024

ABSTRACT

Detecting malware on mobile devices using the Android operating system has become a critical challenge in the field of cybersecurity, in the context of the rapid increase in the number of malware variants and the frequency of attacks targeting Android devices. In this paper, we propose a novel intelligent computational method to enhance the effectiveness of Android malware detection models. The proposed method combines two main techniques: (1) constructing a malware behavior profile and (2) extracting features from the malware behavior profile using graph neural networks. Specifically, to effectively construct an Android malware behavior profile, this paper proposes an information enrichment technique for the function call graph of malware files, based on new graph-structured features and semantic features of the malware's source code. Additionally, to extract significant features from the constructed behavior profile, the study proposes using the GraphSAGE graph neural network. With this novel intelligent computational method, a variety of significant features of the malware have been effectively represented, synthesized, and extracted. The approach to detecting Android malware proposed in this paper is a new study and has not been explored in previous research. The experimental results on a dataset of 40,819 Android software indicate that the proposed method performs well across all metrics, with particularly impressive accuracy and recall scores of 99.03% and 99.19%, respectively, which outperforms existing state-of-the-art methods.

KEYWORDS

Android malware detection; malware behavior profile; function call graph; graph neural network; graph-structured features; semantic features

1 Introduction

1.1 Challenges and Difficulties in Detecting Android Malware

According to statistics in the fourth quarter of 2023, Android is the most popular mobile operating system, accounting for about 70.4% of the mobile operating system market share worldwide [1]. Because of its popularity and open-source properties, the Android operating system is also a primary target for most mobile malware [2]. According to a report by Kaspersky [3], 8,346,169 malware attacks were detected and prevented on Android devices in the third quarter of 2023. Android malware often disguises itself as benign applications, allowing it to bypass the filtering or review mechanisms of the Google Play Store and other third-party application markets such as Huawei, Amazon Appstore, and

APKMirror. After successfully deceiving users into installing the malicious application, the malware proceeds to steal sensitive information and gain unauthorized control over their devices. As a result, the challenge of effectively detecting Android malware has become a pressing concern for researchers today.

There are three primary approaches to detecting Android malware: (1) signature-based detection, (2) behavior-based detection, and (3) machine learning-based detection. Signature-based methods rely on a sequence of bytes extracted from known malware files to create a unique signature for detection. The main advantage of this approach is its high accuracy in identifying known malware samples. However, it is limited by its inability to detect new or previously unknown malware variants. Behavior-based methods require file execution in a monitored environment, where behaviors will be captured and examined to classify the file as benign or malware. This method has the advantage of being able to detect zero-day malware or unknown malware, but the disadvantage of this method is that it is complex and requires a lot of computing resources.

Recently, machine learning-based malware detection methods have been promising solutions for detecting new malware samples and adapting to the rapid evolution and diversification of malware [4]. Based on the extracted features, this method employs a classifier to determine whether a file is malware or benign. Thus, it can be seen that features are considered important factors that determine the performance of machine learning-based malware detection methods. There are three primary types of features commonly used in machine learning models for malware detection: static features, dynamic features, and hybrid features. According to our survey, recent research often relies on static features because they do not require application execution, thereby conserving resources and reducing computational overhead while still achieving effective malware detection performance.

Among static features, features based on function call graph (FCG) have recently been widely used and have shown high efficiency in detecting malware because this graph structure has good representation ability. The relationships between components in the file also provide a lot of information about the function and behavior of the file to be analyzed. An FCG is a directed graph where each node represents a function, and each directed edge represents a function-calling relationship in the program file. Based on the constructed FCG, various graph data mining techniques, such as Graph Convolutional Networks (GCN), Graph Attention Networks (GAT), Deep Graph Convolutional Neural Networks (DGCNN), and Topology Adaptive Graph (TAG), are employed to extract significant features from the FCG. These features are then used as inputs for machine learning and deep learning classifiers to detect malware.

When surveying and evaluating recent approaches to Android malware detection, we identified several issues in need of improvement, as outlined below:

(1) Method for Representing and Synthesizing Android Malware Features: Malicious application files are typically decompiled into Smali code for analysis. Subsequently, the FCG representation technique is employed to model the functions and their interrelationships within the source code. Next, this graph data will be synthesized and feature extracted by graph neural networks. However, this approach can lead to the loss of crucial and meaningful features of Android malware, which can reduce the effectiveness of prediction models. Therefore, it is necessary to have methods to enrich and highlight important and meaningful information from the components in the FCG.

(2) Feature Extraction Method from Graphs: Recent studies often employ various graph neural networks, such as GCN [5,6] and GAT [7,8], to extract features from FCG-based Android malware. Although these techniques have shown promising results, we recognize that to effectively apply a graph neural network for feature extraction, it is essential to first understand and analyze the structural

properties of the graph. This step is crucial for selecting the most appropriate GNN model; otherwise, critical information may be overlooked during the graph embedding process, which could reduce the effectiveness of the classification models.

1.2 Propose a Solution

To address the two problems outlined above, this paper proposes a new approach that combines multiple intelligent computing techniques, as detailed below:

(1) To address the issue of the "Method for Representing and Synthesizing Android Malware Features", this paper proposes a new technique for constructing a malware behavior profile. First, the application file is decompiled and represented as a FCG. Next, the nodes within the FCG are enriched with meaningful features to enhance their informational content. Based on the analysis of functions and function-call relationships in the decompiled source code, we recognized several important features that have either been neglected or not effectively combined in previous studies.

Firstly, this paper proposes five graph-structured features to enhance the informational content of the nodes in the FCG. These features include the in-degree measure, out-degree measure, closeness measure, Katz measure, and clustering coefficient measure. These features are crucial as they represent both the structural properties of the FCG and the importance of each function within the application file. Incorporating these five structural features into the nodes will improve the ability of graph neural networks to perform effective graph representation learning.

Secondly, each node in the FCG represents a function within the application file; however, these functions are not independent but are defined within specific classes and packages. Moreover, functions with the same name but defined in different classes or packages may have different functionalities. Therefore, a method that relies only on embedding function names and using these embeddings as node features may lead to the loss of critical information. This paper proposes a more comprehensive semantic representation method for the FCG nodes by considering the semantics of function names, class names, and package names. These semantic features are obtained using the Word2vec embedding method.

Thus, to enhance the information for the FCG nodes, the paper proposes combining both graph-structured features and semantic features of function names, class names, and package names.

(2) To address the issue of "Feature Extraction Method from Graphs", this paper proposes using a new graph neural network called GraphSAGE to efficiently extract behavior profiles represented as graphs. This advanced graph neural network offers several superior advantages over traditional neural networks.

1.3 Contributions of This Paper

Some scientific contributions of this research include:

(1) Proposing a new model for detecting Android malware. This model is based on advanced computational techniques to improve the effectiveness of Android malware detection and prediction.

(2) The paper proposes a new method for constructing malware behavior profiles. Specifically, it introduces a technique to enrich the information of nodes in the FCG by combining five new graph-structured features with semantic features related to function names, class names, and package names.

(3) Proposes the use of GraphSAGE for the task of extracting features from behavioral profiles represented as graphs.

2 Related Works

This section surveys the studies most closely related to our work. In the problem of detecting Android malware based on machine learning, the features used to detect malware include: static features, dynamic features, and hybrid features.

2.1 Static Features

Static features are obtained through static analysis of APK files without having to execute the files. Static features commonly used to identify Android malware include permissions [9,10], intent actions [11,12] and APIs [13–15].

Permissions are important in the Android operating system's security policy, determining what applications can do and access the resources. Permissions are extracted directly from the AndroidManifest.xml file. In Reference [9], 22 significant permissions were extracted and subsequently used with a Support Vector Machine (SVM) classifier, resulting in malware detection accuracy of over 90%. In Reference [10], a method was proposed to select the most critical permissions based on linear regression instead of all permissions. Experimental results showed that this method achieved an F1 score of 96.1%. However, the approach relying solely on permission features has limitations, as some malware does not need to request sensitive permissions to perform unauthorized actions [16]. Additionally, the malware detection system can be compromised if someone deliberately creates disturbances by declaring numerous permissions in the AndroidManifest.xml file. At the same time, the program's source code does not use functionalities related to these permissions [17].

Intent actions describe the abstract actions that an application intends to perform. This feature can describe sensitive features of malware; however, it is often used in combination with other features to enhance malware detection effectiveness [11,12]. In Reference [11], a solution combining intent action features and permission features was proposed, with experimental results showing a malware detection rate of 95.5%, which is more effective than using only intent action features, which had a detection rate of 91%. In Reference [12], a combination of permission features and intent action features extracted from APK files was also used, achieving a detection accuracy of 99.8%. However, they used a relatively small experimental dataset with only 1745 APK files.

API calls are also commonly used features in Android malware detection because to interact with Android devices, such as performing actions like reading and writing files, retrieving device information, accessing SMS, contacts, camera, etc., need API calls. Therefore, API calls can reflect the behavior of malware. In Reference [17], a method was proposed that uses only a subset of APIs frequently used by malware as features to distinguish between malware and benign applications. However, this approach has the drawback of requiring continuous retraining of the model due to the constant changes in Android API versions and the evolution of malware. Additionally, the feature vector can be large and sparse because each version of the Android operating system provides around 32,000 API functions for application developers to use, which can introduce noise and reduce the effectiveness of malware detection. To address these drawbacks, Reference [14] proposed an alternative approach by abstracting individual API functions prone to changes to higher levels such as family, package, and class levels and modeling the sequence of these abstract API calls as Markov chains to capture malware behavior. However, this approach is ineffective when many different API functions at the same abstraction level have completely different functionalities. Additionally, this method cannot capture API functions when changing package names. In Reference [15], a more advanced method was proposed by clustering APIs based on their semantic distances. They designed API sentences to summarize the important features of APIs and then used the BERT model to extract the semantics of the API sentences. This approach allows their model to effectively handle changes in package names, class names, and API function names across different Android operating system versions.

In addition to the traditional static features considered above, recent research has paid more attention to static features represented in graph form because the graph data structure can model the relationships between objects. Component in Android application files and provides more information about how the application works. There are many different types of graph-based features. Still, in our opinion, FCG features are often used because this representation can represent functions and purposes application goals and shows how functions in an application interact with each other. In FCG, each node represents a function, and each directed edge represents the function-calling relationship in the application program. In Reference [18], FCGs are extracted from APK files using the Androguard tool. Each node in the FCG is assigned a feature vector related to the opcode list and the API packet list. They proposed a node balancing technique to address the imbalance of the number of nodes between benign FCGs and toxic FCGs. Next, the FCGs are fed into graph neural networks (GNNs) to embed the graph and obtain a feature vector for each APK application. Experimental results show that the detection accuracy reaches 92.29%. However, their node representation method only considers the existence of opcodes and API functions but ignores the semantic information of API functions and opcode strings. In addition, their node balancing technique can eliminate many important data samples for malware detection. In Reference [19], a method is proposed to add information to the nodes of FCG by extracting function features, including request permissions, security level, and code features. Next, the function call graphs are fed into a GNN to generate the embedding vector; then finally, the embedding vectors are classified using MLP. Experiments show that the F1 score is 8% higher than other methods. In Reference [20], word embedding technique is applied to convert opcodes from text to vector using the Skip-gram method. These embeddings are used as features for the nodes in the FCG; next, a GNN is applied to embed the graph into a feature vector. To classify malware they used a 2-layer MLP, which experimentally showed a detection accuracy of 99.6%. In Reference [5], a method is proposed to enrich node features in FCG using Word2vec. Specifically, the CBOW model is employed to embed function names into feature vectors, which are then used to enhance the node features. However, this approach has several limitations: First, for each APK file, a list of function names is extracted in the order found in the decompiled source code and considered as a sentence in natural language. This approach is ineffective if malware employs code obfuscation techniques such as reordering the functions in the source code or changing function names. Second, relying solely on the semantic features of function names leads to the loss of important information, as the same function name may have different functionalities when defined in different classes and packages. Third, the approach in the paper does not consider graph-structured features, which are crucial for enhancing the effectiveness of GNNs. In Reference [21], a method is proposed to enrich the features of nodes in the FCG by training two models based on Word2vec. Specifically, if a node represents a system API function, the API2vec model is applied to embed the package name of the corresponding API. Conversely, if the node represents a user-defined function, the Opcode2vec model is used to embed the opcode sequence extracted from that function. Additionally, this study incorporates node degree as weights for the feature vectors of the nodes. However, this approach is limited because it only captures the semantic features of the package name, leading to the loss of important information since the functions within the same package can have relatively different functionalities. Moreover, their approach requires considerable time to extract opcode sequences from the functions. In Reference [22], a method is proposed that combines pretrained CodeBERT and TextCNN models to embed opcode sequences from functions into vectors, thereby enriching the features of the corresponding nodes in FCG. However, this approach has several limitations: First, it does not consider graph-structured features. Second, it relies on the large language model CodeBERT, which was trained on an incomplete dataset. As a result, this model may not fully capture the semantic features of opcode sequences from Android applications that are not present in the pretrained dataset.

2.2 Dynamic Features

Besides static features, dynamic features are often used in Android malware detection models based on machine learning. Dynamic features are obtained through dynamic analysis, which means executing APK application files in a simulated environment or actual device and then collecting and extracting features representing application behavior during execution. Although this method has the disadvantage of consuming time and resources, it helps better understand the behavior of malware, especially malware that uses code obfuscation techniques or dynamic code loading techniques. Common dynamic features include system calls, network operations, and inter-component communication intents.

System calls are a feature commonly used to detect malware because they represent the application's interactions with the device. In Reference [23], a method is proposed to collect system call traces of all benign and malware applications at runtime and then consider the frequency of system calls made by each application as a set of features in classifying applications as malware or benign. However, experimental results show that the accuracy is not high. Specifically, the detection accuracy with Decision Tree (DT) and Random Forest algorithms is 85% and 88%, respectively. In Reference [24], a method to detect malware based on the features of system calls and API function calls was proposed, and the test results achieved an accuracy of 96.66%. However, their approach fails to monitor malware behavior in some cases, such as when an application fails to connect to the Internet. The model stops executing the application without collecting any additional information.

Malware detection models based on system calls face limitations with malware that employs system call obfuscation techniques. In Reference [25], a dynamic feature set based on method calls and inter-component communication intents was proposed. With these new features, their detection model demonstrated effectiveness in identifying malware that uses a system called obfuscation techniques.

Network activities are also an important feature in malware detection because in order to steal private information or download additional malware, the malware application needs to go through network activities. In Reference [26], a malware detection model was proposed based on analyzing network traffic when the application executes, specifically they used seven features including Average Packet Size, Average Number of Packets Sent per Flow, Average Number of Packets Received per Flow, Average Number of Bytes Sent per Flow, Average Number of Bytes Received per Flow, Ratio of Incoming to Outgoing Bytes, Average Number of Bytes Received per Second, to train the Decision Tree algorithm, experimental results for 98.4% detection accuracy. With a similar approach, Reference [27] focused on analyzing TCP session flows in network traffic to extract important features such as the duration of TCP sessions and the total number of packets uploaded or downloaded for each session. TCP session, the average size of each uploaded or downloaded packet, etc., to train the detection model.

In addition to the limitations of requiring large computational resources and computing power, dynamic analysis methods also face challenges in collecting valid features because applications' malware behaviors may not be triggered, or some malware may not execute malware behaviors in simulated environments.

2.3 Hybrid Features

Hybrid features combine static and dynamic features, which can provide a more comprehensive description of malware behaviors. However, this fusion requires large computational resources and computing power. In Reference [28], a mechanism for detecting malware is proposed by combining three features: API calls, application permissions, and system calls. They used the Tree Augmented Naive Bayes (TAN) classification algorithm, and the experimental results showed a detection accuracy of 97%. In Reference [29], dynamic features such as system calls related to file and network operations during application execution are combined with static features, including application permissions, intent actions, suspicious and restricted APIs, application components, and required hardware components. These features are used as input for the SVM classification algorithm, resulting in a detection accuracy of 98.97%. In Reference [30], static features such as permissions and sensitive API calls are extracted from the source code. Dynamic features such as network activities, file system access, interaction with the operating system, etc., are extracted during application execution. All these features are utilized in a machine learning classifier to classify applications as benign or malware. In Reference [31], a CNN-Ensemble model is proposed that integrates static and dynamic features for Android malware detection. Static features are obtained by converting DEX files into greyscale images, processed by a CNN for extraction. Dynamic features are collected through dynamic analysis using the Monkey and strace tools, capturing system call counts. These static and dynamic feature vectors are concatenated and fed into a gradient boosting classifier for final prediction. In Reference [32], a hybrid analysis method is proposed, where static features are obtained by converting malware binaries into grayscale images for analysis with CNN-LSTM networks. Dynamic features are extracted, ranked, and reduced using Principal Component Analysis. The approach integrates static and dynamic features, which are then classified using various classifiers combined through a voting scheme.

In general, static, dynamic, and hybrid feature types have advantages and disadvantages. Still, static features are often used in machine learning models to detect malware because this method does not require application execution and saves resources and computational resources while still providing good malware detection performance.

3 Proposed Model

3.1 Model Architecture

The overall architecture of our proposed Android malware detection model is shown in Fig. 1.

Details of the process and steps in our model are as follows:

(1) Android application dataset block: This includes malware and benign application files in APK format. These files serve as the data for training and detecting Android malware.
(2) Constructing behavior profiles for application files block: As illustrated in Fig. 1, the process of constructing behavior profiles for application files consists of three primary sub-blocks, specifically as follows:
(a) Unzip and decompile block: Unzips APK files and decompiles them into Smali code format, using the Androguard [33] tool.
(b) Building the FCG block: This block constructs the FCG from the decompiled source code obtained in the previous block, using the Androguard tool.
(c) Enriching node features in the FCG block: This block enhances the node features in the FCG by combining each node's graph-structured features with the semantic features

of function names, class names, and package names, obtained through natural language processing techniques. The output of this block is FCGs with enriched node features.

(3) Feature extraction block: This block is responsible for extracting features from the behavior profile of the application file using the GraphSAGE model. The output of this block is graph embedding vectors.

(4) Android malware detection block: This block classifies benign and malicious APK files using the Random Forest algorithm.

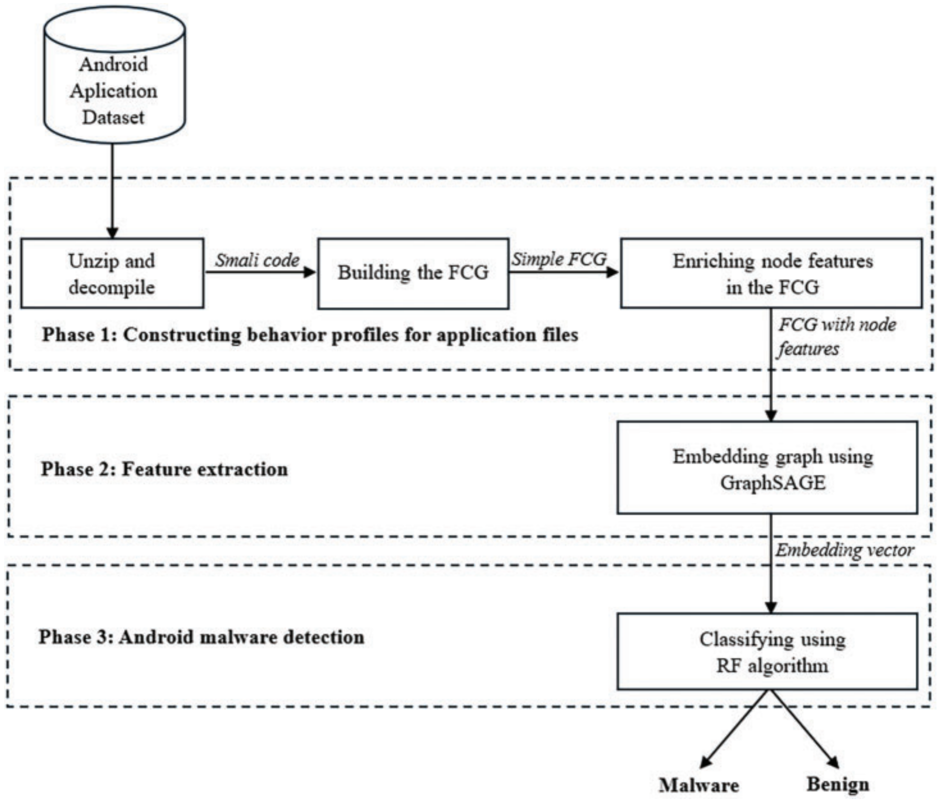
Android Aplication Dataset
Unzip and decompile
Smali code
Building the FCG
Simple FCG
Enriching node features in the FCG
Phase 1: Constructing behavior profiles for application files
FCG with node features
Phase 2: Feature extraction
Embedding graph using GraphSAGE
Embedding vector
Phase 3: Android malware detection
Classifying using RF algorithm
Malware
Benign

Figure 1: Architecture of the Android malware detection model

3.2 Method for Constructing Behavior Profiles

We observe that the FCG extracted from an APK file can model the relationships between the components within the file and illustrate the program's behavior. However, the initial FCG lacks features to represent each node in the graph, which diminishes the effectiveness of graph neural networks in extracting important features from the FCG. Furthermore, this limitation leads to the loss of crucial and meaningful information about the program's source code. Therefore, in this section, we propose a method for enriching the information in the FCG by enhancing the node features through a combination of graph-structured features and semantic features of function names, class names, and package names.

3.2.1 Proposed Graph-Structured Features for Nodes

FCG is a directed graph, and as such, it inherits the properties of general graphs. According to graph theory, nodes in a graph have features that reflect their different levels of importance. For example, we can evaluate a node's importance based on the number of direct connections it has with neighboring nodes. Therefore, a node with more connections to other nodes is considered more important than nodes with fewer connections. In a FCG, a function that is called by many other functions may indicate a central access point for malware, suggesting that this function is a critical part of the malware and is often used to execute harmful actions.

There are several metrics to measure the important role of each node within the graph structure. In this study, we propose using four measures to express the role of nodes, including in-degree, out-degree, closeness, and Katz measures. In addition, to capture the community structure or clustering features around a node, we use the clustering coefficient measure. These five proposed node features will help graph neural networks more effectively learn and represent the graph structure. Detailed descriptions of the proposed measures are as follows:

- **In-degree measure** [34]: The in-degree measure of a node reflects the number of incoming edges from neighboring nodes to the node. The formula to calculate the in-degree measure of a node in the normalized form is as follows:

$$M_{in-D}(v) = \frac{d_{in}(v)}{N-1}, \tag{1}$$

where N is the number of nodes in the graph, $d_{in}(v)$ is the number of incoming edges of node v.

- **Out-degree measure** [34]: The out-degree measure of a node represents the number of outgoing edges from the node to its neighboring nodes. The formula to calculate the out-degree measure of a node in the normalized form is as follows:

$$M_{out-D}(v) = \frac{d_{out}(v)}{N-1}, \tag{2}$$

where N is the number of nodes in the graph, $d_{out}(v)$ is the number of outgoing edges of node v.

- **Closeness measure** [34]: The closeness measure of a node represents the average length of the shortest paths from the node to all other nodes in the graph. The formula to calculate the closeness measure of a node in the normalized form is as follows:

$$M_C(v) = \frac{N-1}{\sum_u d(u,v)}, \tag{3}$$

where N is the number of nodes in the graph, $d(u, v)$ is the shortest path distance from node u to node v.

- **Katz measure** [35]: The Katz measure is a general degree measure that incorporates the importance of neighboring nodes. The Katz measure of a node calculates the influence of a node in the graph by measuring the number of neighbors and all other nodes in the network connected to the node under consideration through these neighbors. The formula to calculate the Katz measure of a node in the normalized form is as follows:

$$M_K(v) = \alpha \sum_u A^T M_K(u) + \beta, \tag{4}$$

where A is the adjacency matrix of the graph with eigenvalues λ, A^T is the transpose matrix of A. α and β are constant, where the value α is usually chosen to satisfy the condition $0 < \alpha < \frac{1}{\lambda_{max}}$ ($\lambda_{\max}$ is the largest eigenvalues of matrix A).

- **Clustering coefficient measure** [36]: The clustering coefficient of a node measures the level of interconnectedness among its neighboring nodes. Therefore, it reflects the level to which these neighbors form clusters around the node under consideration. The formula to calculate the clustering coefficient of a node in the normalized form is as follows:

$$M_{CC}(v) = \frac{|\{e_{zu}\}|}{k_v(k_v - 1)} : z, u \in N_v, e_{zu} \in E, \tag{5}$$

where N_v is the set of neighbor nodes of v, k_v is the size of the set N_v, e_{zu} is the edge connecting node z and u.

Thus, in this paper, to enrich the graph-structured information in the FCG, each node v will be enhanced with a feature vector defined as follows:

$$m_v = [M_{in-D}(v), M_{out-D}(v), M_C(v), M_K(v), M_{CC}(v)], \tag{6}$$

3.2.2 Proposed Semantic Features for Nodes

Each node in the FCG represents a function in an Android application file. A function's name can provide information about its functionality. This meaningful information should be represented as vectors and utilized to enrich the nodes in the FCG. For example, function names such as "sendTextMessage", "getDeviceId" and "onConfigurationChanged" describe meaningful information about their respective functionalities. Furthermore, in an Android application, each function is defined within a specific class and package, thereby establishing a semantic relationship between the function name, class name, and package name. Moreover, functions defined within the same class or package generally exhibit greater similarity compared to functions defined in different classes or packages. Additionally, it is clear that functions with the same name but located in different classes or packages may have different functionalities.

Previous studies have often focused on representing the semantic features of function names without considering the class and package in which the function is defined. This approach can result in a loss of contextual information regarding the function's relationship with its class and package. Furthermore, this method has the drawback that malware employing code obfuscation techniques can easily alter function names to bypass detection models.

This paper proposes a method for representing the semantic features of package names, class names, and function names using natural language processing techniques. These semantic features are then used to enrich the information of the nodes in the FCG. Our approach has the advantage of incorporating the semantics of functions in relation to the class and package in which they are defined. This addresses the issue of contextual information loss and performs effectively against malware using code obfuscation techniques, such as function renaming. Although function names can be easily changed, altering the class and package names is more challenging. As a result, the semantic features of the function are maintained when considered within the context of its class and package.

Word2vec [37] is an effective method for representing words in natural language as low-dimensional vectors, where semantically similar words are positioned close to each other in the vector space. Building on this idea, we treat package names, class names, and function names as words in natural language and apply the Skip-gram model [38,39] to embed these entities and obtain their corresponding feature vectors. In this paper, we use the Skip-gram model instead of CBOW [38] because the Skip-gram model is better suited for learning representations of less common words. In the context of malware detection, many rare and infrequent package names, class names, and function names can play a crucial role in identifying malicious behavior. The Skip-gram model effectively embeds these infrequent names into the vector space.

Fig. 2 illustrates the architecture of the embedding model for package names, class names, and function names. First, each application file in the dataset is unzipped and decompiled. Next, package names, class names, and function names are extracted to construct sentences. Note that in a sentence $s_1 = \{p_1, c_1, f_1\}$, the function f_1 is defined within the class c_1, and the class c_1 belongs to the package p_1. Fig. 3 illustrates a list of sentences extracted from an Android application file. Thus, for each application file, a list of sentences is extracted. Next, we construct a corpus by combining all the sentence lists from the application files. Using the Skip-gram method, we train a fully-connected neural network to embed each package name, class name, and function name into an N-dimensional vector. During training, each object name is represented as a one-hot vector with V dimensions, where V is the size of the vocabulary. The goal of training the Skip-gram model is to maximize the probability of predicting context words based on a given target word. For a sequence of words $\{w_1, w_2, \ldots, w_K\}$ and a window size of $2m + 1$, the objective function can be expressed as the average log probability as follows:

$$J(w) = \frac{1}{K} \sum_{t=1}^{K} \sum_{-m \le j \le m} \log P(w_{t+j} | w_t), \tag{7}$$

$P(w_{t+j}|w_t)$ is defined as follows:

$$P\left(w_{t+j}|w_t\right) = \frac{\exp(e_{w_{t+j}}^T e_{w_t})}{\sum_{i=1}^{V} \exp(e_{w_i}^T e_{w_t})}, \tag{8}$$

here $e_{w_{t+j}}$, e_{w_t} is the vector representation of w_{t+j} and w_t. $P(w_{t+j}|w_t)$ is defined by the *softmax* function.

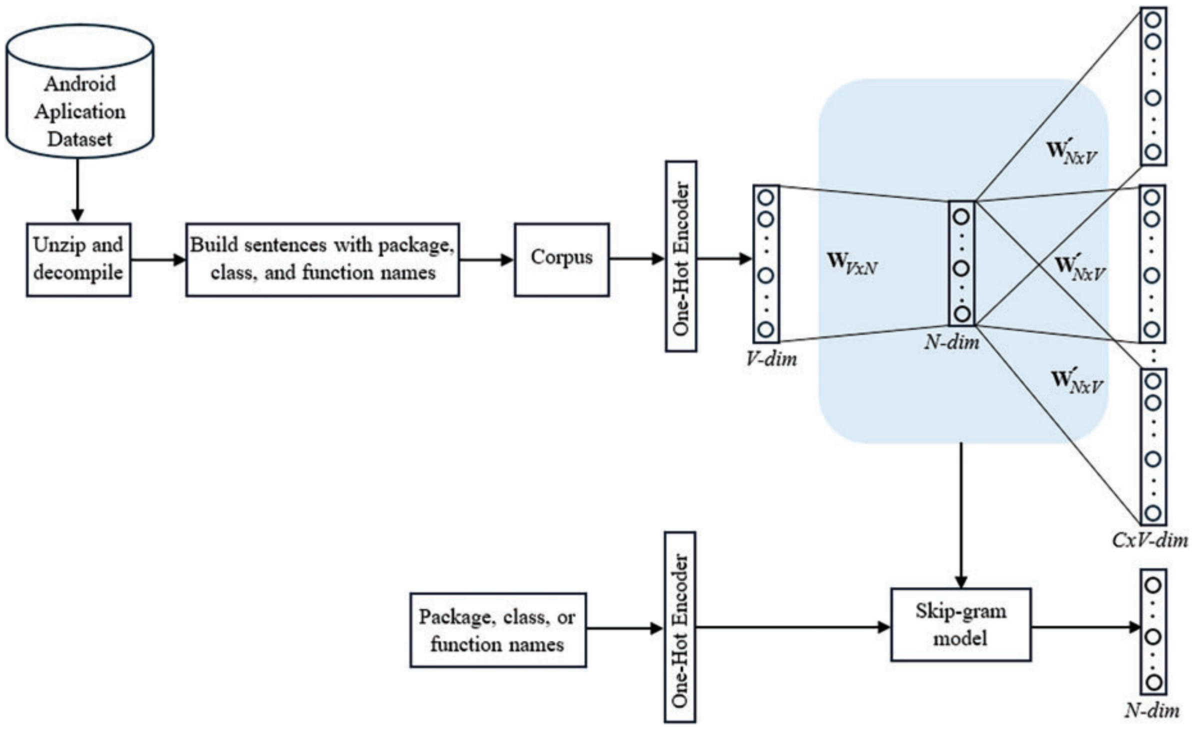

Figure 2: Architecture of the embedding model for package, class, and function names

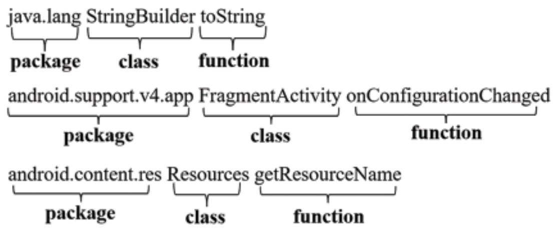
java.lang StringBuilder toString
package
class
function
android.support.v4.app FragmentActivity onConfigurationChanged
package
class
function
android.content.res Resources getResourceName
package
class
function

Figure 3: Illustrates the list of sentences extracted from an application file

For each node v in the FCG, let $e_{package}(v)$, $e_{class}(v)$, and $e_{function}(v)$ denote the embedding vectors for the package name, class name, and function name associated with node v, respectively. These embedding vectors are obtained from the output of the trained Skip-gram model, as illustrated in Fig. 2. To enrich the semantic information of package names, class names, and function names in the FCG, each node v will be enhanced with a feature vector obtained by averaging the embedding vectors for the package name, class name, and function name, as defined below:

$$e_v = Average\left(e_{package}(v), e_{class}(v), e_{function}(v)\right), \tag{9}$$

3.2.3 Combining Graph-Structured and Semantic Features for Nodes

To comprehensively capture both graph-structured and semantic features, each node v in the FCG is enriched with a feature vector x_v, defined as follows:

$$x_v = m_v || e_v, \tag{10}$$

where m_v is the graph-structured feature vector of node v defined by Eq. (6), e_v is the semantic feature vector of node v defined by Eq. (9), and || denotes horizontal concatenation.

Accordingly, the FCG with each node v represented by the feature vector x_v forms the behavior profile of the APK application file.

3.3 Methods for Extracting Features from Behavior Profiles

As mentioned above, the behavior profile of each APK application file is represented as an FCG, including important information: graph structure, node properties, and connections between nodes. However, traditional classification models such as RF, DT, SVM, etc., cannot work with non-Euclidean data. To address this issue, we propose to use graph neural networks to learn features from behavior profile, or in other words, to embed the behavior profile into a feature vector. There are many graph neural network models used in this task, specifically in studies [7,18,40,41] that used GCN, DGCNN, GAT, and TAG to embed graphs. In this paper, we propose using the GraphSAGE model for extracting features from behavior profiles.

3.3.1 Introducing the GraphSAGE Model

The GraphSAGE network is a spatial graph neural network introduced in Reference [42], and it is a type of graph neural networks. Graph neural networks are currently widely applied in various fields, including link prediction [43], node classification [44], community detection [45], graph classification [46], and graph embedding [47].

The GraphSAGE model consists of convolutional layers. On each convolutional layer, each node will create a message and spread it to its neighboring nodes. Next, the model aggregates the messages received from neighboring nodes. Finally, the node embedding vector of each node will be calculated by combining the information of the neighbors and the previous embedding results of this node. So, it can be seen that GraphSAGE operates based on the mechanisms of message propagation and aggregation. Eq. (11) below represents the node embedding process at a convolutional layer in the GraphSAGE model.

$$h_v^{(l)} = \sigma\left(W^{(l)}.concat\left(h_v^{(l-1)}, aggregate\left(\left\{h_u^{(l-1)}, \forall u \in N(v)\right\}\right)\right)\right), \quad (11)$$

$$h_v^{(l)} = norm\left(h_v^{(l)}\right), \quad (12)$$

in there:

- $W^{(l)}$: weight matrix used for each layer of the model.
- $N(v)$: set of neighbors of node v.
- *aggregate*: the function used to combine messages from neighboring nodes.
- σ: the activation function.
- *norm*: the normalization function.
- $h_v^{(l)}$: the feature vector of node v in layer l.
- $h_v^{(0)} = x_v$.
- x_v: the feature vector of node v, as defined in Eq. (10).

Typically, the following parameters are commonly used: $aggregate = mean$, $\sigma = ReLu$, $norm\left(h_v^{(l)}\right) = \frac{h_v^{(l)}}{||h_v^{(l)}||_2}$).

3.3.2 *Proposed GraphSAGE Architecture for Extracting Features*

We propose the GraphSAGE network architecture, as shown in Fig. 4 below, to apply the GraphSAGE model for extracting features from the behavior profile of the application file.

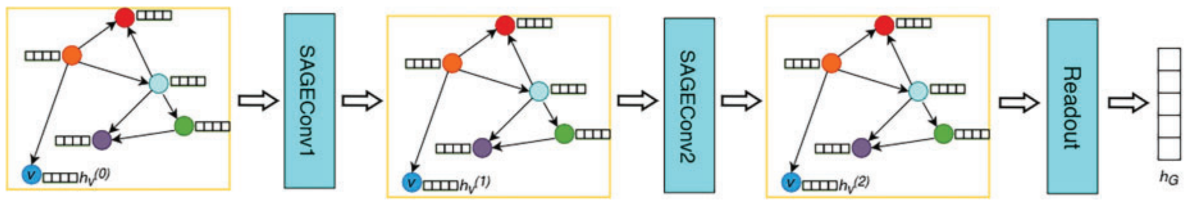

Figure 4: The GraphSAGE model architecture for graph feature extraction

From Fig. 4, we see that there are two main components used in the GraphSAGE model, including two GraphSAGE layers and a Readout layer. The process of extracting features from the FCG graph is as follows:

- The input to the GraphSAGE model is an FCG graph with node features. Specifically, the model receives an adjacency matrix A as input to represent the graph structure and a feature matrix X in which each row is a feature vector of a node.

- In the two convolutional layers, the SAGEConv1 and the SAGEConv2, feature updates will be performed on each node according to the propagation and aggregation rules shown in Eq. (11). The output of the GraphSAGE layers is the node embedding vector of each node.

- In the Readout step, the node embedding vectors of all nodes will be combined to create the graph embedding vector, which is considered the result of the FCG graph feature extraction process. The pooling operations typically used are *mean*, *sum*, or *max*; in this model, we use *max* pooling.

3.4 Method to Detect Android Malware

Supervised machine learning algorithms can be used to classify the feature vectors obtained from the behavior profile feature extraction block for Android malware detection. Specifically, we propose to use a basic and popular machine learning algorithm for the classification task, algorithm RF. The RF algorithm has shown its effectiveness when used in our model compared with some other traditional classification algorithms such as SVM and DT, and the experiments of this paper also prove RF for file classification results with high and stable measurements.

4 Experiment and Evaluate

4.1 Experimental Dataset and Environment

In this study, we use a large dataset with 40,819 APK files, including 20,406 benign files and 20,413 malware files. The benign application files were obtained from Androzoo [48] and the malware application files were obtained from Virusshare [49], and all these files were collected from January 2018 to December 2023. To ensure the dataset's reliability, all APK files were verified and labeled using the VirusTotal API. Details of the experimental dataset are shown in Table 1 below.

Table 1: Summary statistics of the experimental datasets

Class	Dataset	Date range	Number of samples
Malware	Virusshare	2018.01–2023.12	20,413
Benign	Androzoo	2018.01–2023.12	20,406

The dataset is divided into 80% used for model training and 20% used for validation.

All experiments were conducted on a server with the following environment: (1) Operating system: Linux-5.15.154-x86_64 × 86_64-with-Ubuntu-22.04; (2) GPU: Tesla T4-PCIE-16 GB (×2); (3) Software installs: Python 3.10.14, PyTorch 2.4.0, and Dgl 2.0.0.

4.2 Evaluation Criteria and Scenarios

4.2.1 Evaluation Criteria

To evaluate the effectiveness of the model, we use four metrics described below:

- Accuracy: The ratio between the number of correctly classified samples and the total number of samples. Accuracy is calculated according to the following formula:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} \times 100\%, \tag{13}$$

- Precision: The ratio between the number of samples correctly predicted positive and the total number of samples predicted positive (including correct and incorrect predictions). A high Precision value means the model can predict malware samples accurately and is less likely to be confused with benign samples. Precision is calculated according to the following formula:

$$precision = \frac{TP}{TP + FP} \times 100\%, \tag{14}$$

- Recall: The ratio between the number of correctly predicted positive samples and the total number of malware samples. A high Recall value means the model can detect more malware cases.

$$\text{re}call = \frac{TP}{TP + FN} \times 100\%, \tag{15}$$

- F1 score: The harmonic average between precision and recall measures. A high $F1$ value shows that the model performs well in detecting malware.

$$F1 = \frac{2 \times precision \times recall}{precision + recall}. \tag{16}$$

In there: TP—True Positive: Number of malware files correctly classified; FN—False negative: Number of malware files classified as benign files; TN—True negative: Number of benign files classified correctly; FP—False Positive: Number of benign files classified as malware files.

4.2.2 Evaluation Scenarios

In this paper, to evaluate the effectiveness of the proposed method, we conducted the following experimental scenarios:

- Scenario 1: Testing and evaluating the proposed model based on changing some parameters in the model.

- Scenario 2: Evaluating the effectiveness of the GraphSAGE graph embedding method by comparing it with other graph embedding methods such as GCN, GAT, and TAG.

- Scenario 3: Evaluating the effectiveness of the RF classification algorithm. In this scenario, the paper will replace the RF algorithm with other classifiers, such as SVM and DT. Besides, we also conducted an additional situation without using machine learning algorithms for classification but using the classification model in the GraphSAGE model.

- Scenario 4: Comparing and evaluating our proposed method with several other studies on the same experimental dataset.

4.3 Experimental Results

4.3.1 Experimental Results According to Scenario 1

As mentioned above, in this experimental scenario, we evaluated the effectiveness of our proposed model for classifying malware and benign files. The results in Scenario 1 are shown in Table 2 below. In the table, the best results for the model using the proposed enriched node features are highlighted in bold, while the best results for the model without these features are highlighted in red and bold.

From Table 2, it can be seen that changing the number of layers of GraphSAGE and the number of trees in the RF algorithm yields different results for our proposed model. With 2 layers of the GraphSAGE and 100 trees in the RF algorithm, the proposed malware detection model achieves the highest performance across most metrics, specifically accuracy = 99.03%, precision = 98.87%, recall = 99.19%, and F1 = 99.03%. The results in Table 2 also show that although the number of trees changes rapidly from 10 to 100 and the number of the GraphSAGE layers changes from 1 to 3, the metrics do not vary significantly. Specifically, the differences between the best and worst results for each metric are as follows: accuracy varies by 0.5%, precision by 0.98%, recall by 0.49%, and F1 score by 0.51%. These differences indicate that the RF algorithm provides good and stable classification performance despite changes in the number of trees. Additionally, this demonstrates the stability of the GraphSAGE embedding algorithm.

Table 2: Comparison of application file classification with and without proposed enriched node features

Approach	Number of GraphSAGE layers	Parameters of the RF algorithm	Accuracy	Precision	Recall	F1
File classification with proposed features	1	Number of trees = 10	98.82	98.45	99.18	98.82
		Number of trees = 50	98.92	98.87	98.97	98.92
		Number of trees = 100	**99.03**	**98.92**	99.14	**99.03**
	2	Number of trees = 10	98.53	97.94	99.11	98.52
		Number of trees = 50	98.91	98.67	99.14	98.90
		Number of trees = 100	**99.03**	98.87	**99.19**	**99.03**
	3	Number of trees = 10	98.59	98.28	98.89	98.58
		Number of trees = 50	98.73	98.53	98.91	98.72
		Number of trees = 100	98.69	98.67	98.70	98.69
File classification without proposed features	1	Number of trees = 10	85.49	85.48	85.42	85.45
		Number of trees = 50	85.14	85.97	84.50	85.23
		Number of trees = 100	85.06	**86.05**	84.31	85.17
	2	Number of trees = 10	85.15	85.48	84.86	85.17
		Number of trees = 50	85.24	85.92	84.70	85.31
		Number of trees = 100	85.14	86.02	84.47	85.24
	3	Number of trees = 10	**85.64**	84.99	**86.05**	**85.52**
		Number of trees = 50	85.22	85.95	84.64	85.29
		Number of trees = 100	85.24	**86.05**	84.61	85.32

Moreover, the experimental results show that using our proposed features to enrich the nodes of the FCG brings significant improvements compared to those without using the proposed features. Specifically, the method using the features in our paper to enhance the FCG yields better results across all metrics by 13.86% to 14.88%. The reason is that the additional features for the nodes in the FCG provide both structural information about each node in the graph and semantic information related to the package names, class names, and function names associated with those nodes. These important pieces of information help graph neural network models understand the graph structure, distinguish the importance of each node in the graph structure, and comprehend the semantics of the nodes in the FCG, thereby enabling the neural network model to learn the most critical features of the FCG.

Fig. 5 below describes the experimental results in the form of a confusion matrix.

Fig. 5a presents the model's confusion matrix results using the proposed features. In contrast, Fig. 5b shows the confusion matrix results of the model without the proposed features, with the parameters yielding the best results, as seen in Table 2. From Fig. 5a, it is evident that for malware classification, the model using the proposed features correctly classifies 4038 out of 4071 malware files, resulting in a correct classification rate of 99.19% and a false negative rate of only 0.81%. For benign files, the model correctly classifies 98.88% of them, with a false positive rate of only 1.12%. This can be considered a good classification result for the malware detection task.

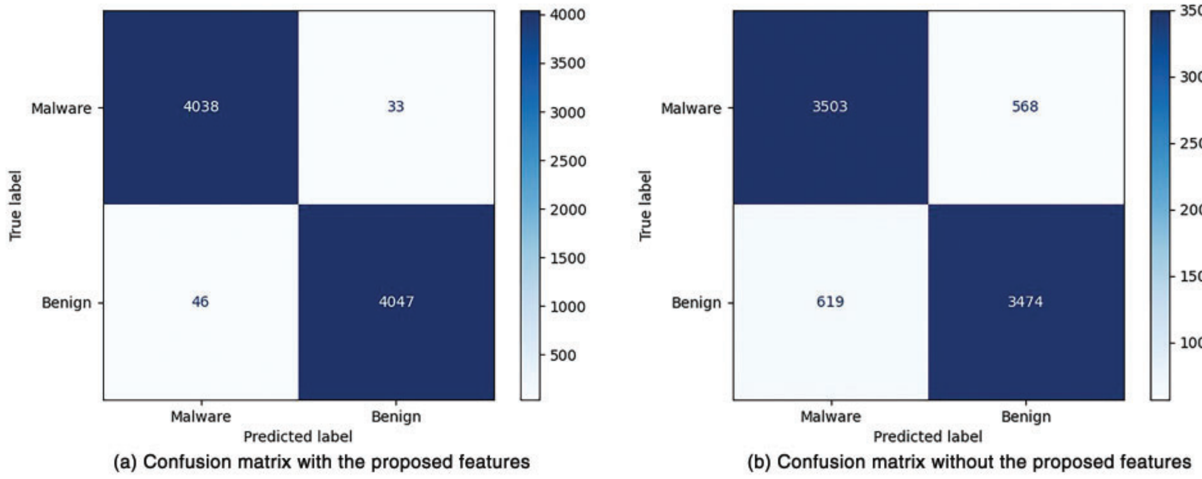

Figure 5: Confusion matrix of the best results for the approaches with and without the proposed features

Comparing the confusion matrices in Fig. 5a,b, it can be seen that the model using the proposed features significantly improves the detection of malware files compared to the model without these features. Specifically, the model using the proposed features correctly detects 4038 out of 4071 malware files, while the model without these features correctly detects only 3503 malware files. Thus, in terms of enhancing the accurate detection of Android malware, the model we proposed (using the features to enrich the FCG) performs significantly better than the model that does not use the proposed features. Furthermore, from Fig. 5a,b, it is evident that the model using our proposed features only misclassifies 46 benign files as malware. In contrast, the model without these features misclassifies a much larger number of benign files, with 619 benign files being incorrectly classified as malware. This result further demonstrates the reliability of our model.

4.3.2 Experimental Results According to Scenario 2

In this scenario, we want to compare and evaluate the effectiveness of the GraphSAGE graph embedding method used in our model with other graph embedding methods, including GCN [50], GAT [51], TAG [52]. These are all graph neural networks that are effective in synthesizing and extracting graph information. Table 3 below shows details of model experimental results when replacing GraphSAGE with graph embedding methods GCN, GAT, and TAG.

Table 3: Experimental results evaluate the role of the GraphSAGE graph embedding model

Approach	Numbers of layer	Embedding graph method	Accuracy	Precision	Recall	F1
File classification with proposed features	1	GCN	98.51	98.35	98.64	98.50
		GAT	98.48	98.40	98.55	98.48
		TAG	98.21	98.13	98.28	98.21
	2	GCN	**98.76**	**98.80**	98.72	**98.76**
		GAT	98.74	98.72	**98.75**	98.73

(Continued)

Table 3 (continued)

Approach	Numbers of layer	Embedding graph method	Accuracy	Precision	Recall	F1
		TAG	98.58	98.62	98.53	98.58
	3	GCN	98.59	98.65	98.53	98.59
		GAT	98.65	98.60	98.70	98.65
		TAG	97.92	98.21	97.63	97.92
File classification without proposed features	1	GCN	93.37	90.84	95.65	93.18
		GAT	85.06	85.80	84.47	85.13
		TAG	92.94	91.87	92.72	91.75
	2	GCN	**94.23**	92,41	**95.87**	**94.11**
		GAT	85.04	86.02	84.30	85.16
		TAG	91.95	89.56	94.02	91.73
	3	GCN	94.18	**92.58**	95.61	94.07
		GAT	85.11	85.92	84.47	85.19
		TAG	93.66	91.62	95.47	93.51

From the results in Table 3, considering the approach using the proposed features in the paper, we see that among the three graph embedding methods, GCN, GAT, and TAG, GCN provided the best and most stable results. Specifically, the GCN graph embedding method with 2 layers gives the best classification results with an accuracy of 98.76%, precision of 98.80%, recall of 98.72%, and F1 score of 98.76%. However, compared to the GraphSAGE embedding results in Table 2, the GraphSAGE performed better with an increase of 0.27% accuracy, 0.07% precision, 0.47% recall, and 0.27% F1 score. Moreover, from the results in Table 3, we also see that the three graph embedding methods, GCN, GAT, and TAG, all provide relatively good and stable results, especially with precision values ranging from 98.13% to 98.80%. This indicates that the proposed features in the paper play an important role and help the graph embedding models extract graph information, thereby enabling the model to accurately predict malware samples and minimize false positives with benign samples. To make this statement more specific, we look at Fig. 6, which shows the confusion matrix of the GCN model below.

Fig. 6 shows the confusion matrix for the best model when GraphSAGE is replaced with GCN, GAT, and TAG, respectively. Comparing the results in Figs. 5a and 6, we see that the GraphSAGE graph embedding method correctly detected 4038 malware files, while the GCN graph embedding method only detected 4019 malware files. In addition, the GraphSAGE graph embedding method only misclassified 46 benign files as malware, while the GCN graph embedding method gave a higher misclassification result with a number of 49 files. GraphSAGE proves to be more effective than other graph embedding methods such as GCN, GAT, and TAG.

From the results in Table 3 with the approach not using the proposed features, we see that all three methods, GCN, GAT, and TAG, are less effective than those using the proposed features. In particular, the GAT graph embedding method with an approach without using the proposed features is much less effective than an approach that uses the proposed features. Specifically, the accuracy is lower from 13.44% to 13.63%, precision is lower from 12.60% to 12.70%, recall is lower from 14.25% to 14.28%, and F1 score is lower from 13.35% to 13.54%. For the GCN and TAG graph embedding methods, it is less affected by the approach that does not use the proposed features, so accuracy is lower from 4.53% to 5.97%, precision is lower from 6.22% to 8.57%, recall lower from 2.92% to 4.91% and F1 score lower from 4.65% to 6.19%.

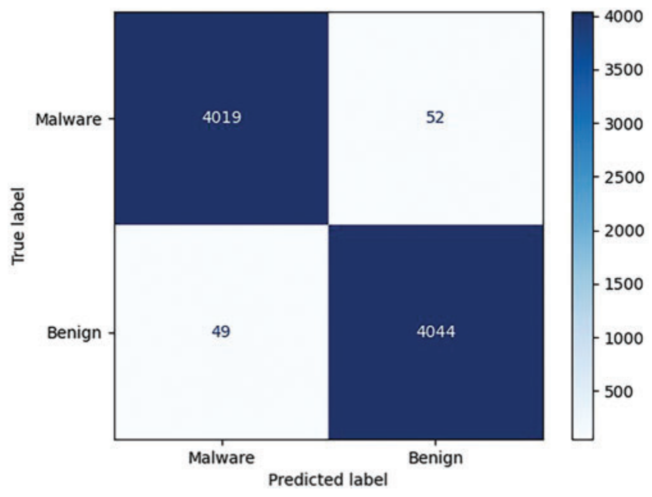

Figure 6: Confusion matrix of the GCN graph embedding method with 2 layers and using the features proposed

4.3.3 Experimental Results According to Scenario 3

Table 4 below shows the experimental results of Scenario 3. The results in Tables 2 and 4, considering the same approach using the features proposed in the paper, show that the results of file classification using the RF algorithm in the proposed model obtained better results than DT and SVM algorithms in all metrics. Specifically, the classification method using RF results in higher accuracy from 0.29% to 1.17%, higher precision from 0.32% to 0.86%, higher recall from 0.27% to 1.49%, and higher F1 from 0.3% to 1.18%. Between algorithms DT and SVM, the SVM algorithm gives the best file classification results with accuracy = 98.74%, precision = 98.55%, recall = 98.92%, and F1 = 98.73%. Furthermore, when comparing this result with other malware detection methods shown in Table 5, we see that using the features proposed still gives better results than other methods. This proves that the proposed features enrich FCG information and demonstrate the effectiveness of the GraphSAGE graph information extraction method.

Table 4: Experimental results evaluate the role of RF algorithm

Approach	Number of GraphSAGE layers	Classification algorithms	Accuracy	Precision	Recall	F1
File classification with proposed features	1	DT	97.98	98.13	97.82	97.98
		SVM	**98.74**	**98.55**	98.92	**98.73**
		Classification using GraphSAGE	98.64	98.21	99.06	98.63
	2	DT	97.99	98.06	97.92	97.99

(Continued)

Table 4 (continued)

Approach	Number of GraphSAGE layers	Classification algorithms	Accuracy	Precision	Recall	F1
		SVM	98.57	98.33	98.79	98.56
		Classification using GraphSAGE	98.70	98.30	**99.10**	98.65
	3	DT	97.86	98.01	97.70	97.85
		SVM	98.16	98.01	98.30	98.15
		Classification using GraphSAGE	98.13	98.26	97.99	98.12
File classification without proposed features	1	DT	85.11	85.19	84.98	85.08
		SVM	77.89	63.89	88.59	74.24
	2	DT	85.08	85.11	84.99	85.05
		SVM	77.92	65.41	87.08	74.71
	3	DT	**85.15**	**85.48**	**84.86**	**85.17**
		SVM	77.98	65.34	87.30	74.74

Table 5: Results of scenario 4 for methods comparison

Methods	Accuracy	Precision	Recall	F1
DexRay [53]	96.82	96.85	97.56	96.15
MalScan [54]	97.58	97.45	97.71	97.58
NATICUSdroid [55]	97.42	96.66	98.18	97.41
Permission [56]	97.28	96.33	97.34	98.38
DeepMalDet [57]	96.39	98.56	94.15	96.30
MalDetGCN [18]	96.82	97.21	96.37	96.79
GCN [5,6]	98.76	98.80	98.72	98.76
GAT [7,8]	98.74	98.72	98.75	98.73
Ours	**99.03**	**98.87**	**99.19**	**99.03**

From the results in Table 4, comparing the approach using the features proposed in the paper and the approach not using these features, we once again see that for both cases using the algorithm SVM and DT classification, the approach using the proposed features gives completely better results than the approach not using these features in all measures, specifically accuracy is 13.59% higher, precision is 13.07% higher, recall is 14.06% higher and F1 is 13.56% higher.

Besides, from the results in Table 4, we see that the classification method using the GraphSAGE model with 2 layers gives the best results with accuracy = 98.70%, precision = 98.30%, recall = 99.10%, and F1 = 98.65%. However, when comparing with the classification results using the RF algorithm in Table 2, it is seen that classification using the GraphSAGE model gives lower results; specifically accuracy is 0.33% lower, precision is 0.57% lower, recall is 0.09% lower and F1 score is 0.38% lower. Furthermore, when comparing the classification results using the GraphSAGE model with using the

SVM machine learning algorithm, we also see that using SVM is better in most measures. This once again proves that when the model uses classification methods using some machine learning algorithms such as RF, SVM will be more effective than classification techniques using the GraphSAGE model.

4.3.4 Experimental Results According to Scenario 4

In this scenario, we compared our method with several methods proposed in other studies on the same dataset. Specifically, we chose 6 Android malware detection methods, including DexRay [53], MalScan [54], NATICUSdroid [55], Permission [56], DeepMalDet [57], MalDetGCN [18]. These methods were chosen because they are representative and open source. DexRay converts APK files into grayscale images and classifies them using a CNN model. MalScan is a malware detection method that analyzes the centrality of sensitive API calls. NATICUSdroid uses native permissions and custom permissions as features to detect malware. Permission is a method of detecting malware based on the features of the application's required permissions, which are selected using a genetic algorithm. DeepMalDet uses opcode strings as features to detect malware. MalDetGCN is a method based on FCGs, where the nodes are enriched with features from opcode sets and API packages. For each comparison method, we experiment with many sets of parameters and select the best result as the final result. Additionally, in this experiment, we compare our proposed approach using GraphSAGE with other approaches using GCN [5,6] and GAT [7,8] to embed the FCG graphs. Table 5 shows the experimental results of scenario 4; the best results are highlighted in bold.

From Table 5, it can be seen that our proposed method gives the best results in all metrics when compared with the remaining 6 methods: DexRay, MalScan, NATICUSdroid, Permission, DeepMalDet, MalDetGCN. Specifically, our method achieves higher accuracy from 1.45% to 2.64%, higher precision from 0.31% to 2.54%, higher recall from 1.01% to 5.04%, and higher F1 value from 0.65% to 2.88%.

In addition, Table 5 shows that our approach using GraphSAGE is more effective than the GCN and GAT methods used in other studies in all measures. This once again proves that the approach of using GraphSAGE to detect Android malware is effective.

4.4 Discussion

4.4.1 Advantages of the Proposed Model

The paper's experimental results show that the proposed model is more effective than other studies and approaches. We think there are three reasons why the model works well.

a) The superiority of the graph information enrichment method

Based on the experimental results, our proposed method for enriching the FCG information has shown superior effectiveness. This is due to two key factors:

Firstly, we enriched the FCG nodes with important graph-structured features, including the in-degree measure, out-degree measure, closeness measure, Katz measure, and clustering coefficient measure of the nodes. These features are critical for representing the FCG graph structure, but are often challenging for graph neural network models, such as GCN, GAT, TAG, GraphSAGE, and DGCNN, to learn directly. By incorporating these structural features into the nodes, we enable the graph neural networks to better differentiate between FCGs with various structures and evaluate the significance of each node more effectively.

Secondly, package names, class names, and function names can provide initial semantic information about the functionalities of a function and its semantic relationships with packages and classes.

Recognizing this, we proposed a method to represent the semantics of package names, class names, and function names using the Skip-gram model. By combining this semantic feature vector with the graph-structured feature vector to form a unique feature vector representing the nodes in the FCG. As mentioned above, the features used to enrich the node information in the FCG are novel and have not been used in other studies, which contributed to our model's excellent performance in malware detection.

b) The superiority of the GraphSAGE

The experimental results from Scenario 2 demonstrate that the GraphSAGE model outperforms other graph embedding models such as GCN, GAT, and TAG. Several reasons can explain this. Firstly, GraphSAGE can use inductive representation learning, meaning it can embed graphs with new structures that were not present in the training dataset. This enables GraphSAGE to handle effectively sets of FCGs that often have continuously changing sizes and structures. Secondly, GraphSAGE employs adaptive sampling methods to select the neighbors of each node during the information aggregation process. This means it can automatically adjust the size and depth of a node's neighborhood instead of using a fixed neighborhood size. This flexibility allows the model to gather more important information from a node's neighbors, leading to better performance in embedding and, consequently, in the overall task, such as malware detection.

c) Flexibility of RF algorithm

Experimental results also showed the superior effectiveness of the RF classification algorithm compared to other traditional classification algorithms such as SVM and DT. In particular, the experimental results also showed that although the hyperparameters of RF changed, the file classification results were stable and highly effective. This shows that using the RF algorithm in our model is a suitable choice, and it is not necessary to use other complex deep learning classification algorithms such as MLP, CNN, or LSTM.

4.4.2 Threats to Validity and Future Development Directions

a) Threats to validity

It can be seen that the proposed method in this paper has brought positive results, but there are still certain limitations. First is the process of enriching information for the FCG graphs. We realized that although the method of enriching information for nodes in the FCG graph has brought good results, it unintentionally creates bulkiness and complexity for the FCG graph. Besides, the process of calculating and extracting information to enrich the node also takes a lot of time and computing resources. The second issue is the extraction of malware features based on the FCG. According to our observations, the GraphSAGE model is suitable and effective for this type of graph. However, it should be noted that each FCG of malware will have a different structure and complexity, particularly in terms of depth. If this extraction process is uniform, it may lead to missing important and meaningful information. Moreover, when comparing and theoretically evaluating GraphSAGE with GAT or GCN, this study finds that GCN can perform well on graphs with structures that are not complex. Therefore, if the dataset consists of small and simple graphs, GCN can serve as a straightforward and effective model for feature extraction. Finally, we found that the experimental data set in our study was relatively uniform. In reality, it is different because benign APK files are often much larger in number than malware APK files. Therefore, detection models must be proposed to solve such data imbalance problems.

b) Future development directions

Based on the limitations pointed out above, we believe that there needs to be research and proposals following the following approaches in the future. First, the information enrichment process for FCG needs to be combined with some graph pruning techniques to obtain a graph that is both full of information but still, the least complex so that the extraction process brings better performance. Second, it is about the graph feature extraction process. Obviously graph neural networks are one of the indispensable approaches at this step. Putting all information about nodes, edges, and features representing nodes into a graph neural network may result in a loss in extracting some important features from the graph. Therefore, we think we can use some additional analysis approaches and extract node features, edge features, and node features separately, using different techniques to extract features of each component in the FCG. Finally, addressing the issue of imbalanced datasets is a major challenge in Android malware detection. Future research will investigate rebalancing techniques such as SMOTE, GAN, and Dropout. These methods can be applied either before or after the graph embedding process to improve dataset balance and model performance. We believe that with such advanced approaches, the model proposed in this paper will certainly bring good results in practice.

5 Conclusion

Detecting Android malware has been an urgent and necessary issue. In this paper, with the goal of improving the efficiency of detecting malware, we provided a new approach based on the combination of some different data mining techniques and algorithms. Experimental results on many different scenarios have proven that this research is not only scientific but also practical. Accordingly, we proposed a new approach based on the information enrichment method for the FCG and graph embedding technique. Experimental results show that the detection and prediction process is superior to other methods when using the proposed method. As for practicality, we believe that the results will open a new approach, a new trend for other detection problems such as detecting Botnets, detecting malware on Windows, detecting URL malware, detecting APT malware, and detecting unusual behavior in the network.

Acknowledgement: This research was supported by the scientific research from the Posts and Telecommunications Institute of Technology, Hanoi, Vietnam.

Funding Statement: The authors received no specific funding for this study.

Author Contributions: Study conception and design: Manh Vu Minh, Cho Do Xuan. Experiment and interpretation of results: Manh Vu Minh. Manuscript preparation: Manh Vu Minh, Cho Do Xuan. All authors reviewed the results and approved the final version of the manuscript.

Availability of Data and Materials: In this study, we used a public dataset, which can be downloaded from the website https://github.com/manhvuminh/Android_Malware_Detection_AC (accessed on 24 October 2024).

Ethics Approval: Not applicable.

Conflicts of Interest: The authors declare no conflicts of interest to report regarding the present study.

References

[1] "Mobile operating system market share worldwide," StatCounter Global Stats. Accessed: May 06, 2024. [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide/

[2] S. Yamaguchi, B. Gupta, S. Yamaguchi, and B. Gupta, "Malware threat in internet of things and its mitigation analysis," in *Research Anthology on Combating Denial-of-Service Attacks*, Hershey, PA, USA: IGI Global, 2021, pp. 371–387.

[3] "Mobile malware statistics, Q3 2023," Accessed: May 06, 2024. [Online]. Available: https://securelist.com/it-threat-evolution-q3-2023-mobile-statistics/111224/

[4] A. Gaurav, B. B. Gupta, and P. K. Panigrahi, "A comprehensive survey on machine learning approaches for malware detection in IoT-based enterprise information system," *Enterp Inf. Syst.*, vol. 17, no. 3, Mar. 2023, Art. no. 2023764. doi: 10.1080/17517575.2021.2023764.

[5] M. Cai, Y. Jiang, C. Gao, H. Li, and W. Yuan, "Learning features from enhanced function call graphs for Android malware detection," *Neurocomputing*, vol. 423, no. 7, pp. 301–307, Jan. 2021. doi: 10.1016/j.neucom.2020.10.054.

[6] H. Gao, S. Cheng, and W. Zhang, "GDroid: Android malware detection and classification with graph convolutional network," *Comput. Secur.*, vol. 106, no. 6, Jul. 2021, Art. no. 102264. doi: 10.1016/j.cose.2021.102264.

[7] S. Chen, B. Lang, H. Liu, Y. Chen, and Y. Song, "Android malware detection method based on graph attention networks and deep fusion of multimodal features," *Expert. Syst. Appl.*, vol. 237, no. 1, Mar. 2024, Art. no. 121617. doi: 10.1016/j.eswa.2023.121617.

[8] C. Catal, H. Gunduz, and A. Ozcan, "Malware detection based on graph attention networks for intelligent transportation systems," *Electronics*, vol. 10, no. 20, Oct. 2021, Art. no. 2534. doi: 10.3390/electronics10202534.

[9] J. Li, L. Sun, Q. Yan, Z. Li, W. Srisa-an and H. Ye, "Significant permission identification for machine-learning-based android malware detection," *IEEE Trans. Ind. Inform.*, vol. 14, no. 7, pp. 3216–3225, Jul. 2018. doi: 10.1109/TII.2017.2789219.

[10] D. Ö. Şahin, O. E. Kural, S. Akleylek, and E. Kılıç, "A novel permission-based Android malware detection system using feature selection based on linear regression," *Neural Comput. Appl.*, vol. 35, no. 7, pp. 4903–4918, Mar. 2023. doi: 10.1007/s00521-021-05875-1.

[11] A. Feizollah, N. B. Anuar, R. Salleh, G. Suarez-Tangil, and S. Furnell, "AndroDialysis: Analysis of android intent effectiveness in malware detection," *Comput. Secur.*, vol. 65, no. 3, pp. 121–134, Mar. 2017. doi: 10.1016/j.cose.2016.11.007.

[12] F. Idrees, M. Rajarajan, M. Conti, T. M. Chen, and Y. Rahulamathavan, "PIndroid: A novel Android malware detection system using ensemble learning methods," *Comput. Secur.*, vol. 68, no. 1, pp. 36–46, Jul. 2017. doi: 10.1016/j.cose.2017.03.011.

[13] Y. Aafer, W. Du, and H. Yin, "DroidAPIMiner: Mining API-level features for robust malware detection in android," presented at the Secur. Priv. Commun. Netw. Conf., Cham, Switzerland, Sep. 2013, pp. 86–103.

[14] L. Onwuzurike, E. Mariconti, P. Andriotis, E. D. Cristofaro, G. Ross and G. Stringhini, "MaMaDroid: Detecting android malware by building markov chains of behavioral models (Extended version)," *ACM Trans. Priv. Secur.*, vol. 22, no. 2, pp. 14:1–14:34, Apr. 2019. doi: 10.1145/3313391.

[15] H. Yang, Y. Wang, L. Zhang, X. Cheng, and Z. Hu, "A novel Android malware detection method with API semantics extraction," *Comput. Secur.*, vol. 137, Feb. 2024, Art. no. 103651. doi: 10.1016/j.cose.2023.103651.

[16] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock android smartphones," presented at the 19th Ann. Symp. Netw. Distrib. Syst. Secur. (NDSS), San Diego, CA, USA, Feb. 5–8, 2012.

[17] X. Chen *et al.*, "Android HIV: A study of repackaging malware for evading machine-learning detection," 2021, *arXiv:1808.04218v4*.

[18] V. K. Vinayaka and C. D. Jaidhara, "Android malware detection using function call graph with graph convolutional networks," presented at the 2nd Int. Conf. Secur. Cyber Comput. Commun. (ICSCCC), Jalandhar, India, May 10–12, 2021, pp. 279–287.

[19] P. Feng, J. Ma, T. Li, X. Ma, N. Xi and D. Lu, "Android malware detection based on call graph via graph neural network," presented at the Int. Conf. Netw. Netw. Appl. (NaNA), Haikou, China, Dec. 8–10, 2020, pp. 368–374.

[20] P. Xu, C. Eckert, and A. Zarras, "Detecting and categorizing Android malware with graph neural networks," presented at the 36th Ann. ACM Symp. Appl. Comput. (SAC), New York, NY, USA, Apr. 22–26, 2021, pp. 22–26.

[21] Z. Liu, R. Wang, N. Japkowicz, H. M. Gomes, B. Peng and W. Zhang, "SeGDroid: An Android malware detection method based on sensitive function call graph learning," *Expert. Syst. Appl.*, vol. 235, no. 4, Jan. 2024, Art. no. 121125. doi: 10.1016/j.eswa.2023.121125.

[22] J. Gu, H. Zhu, Z. Han, X. Li, and J. Zhao, "GSEDroid: GNN-based Android malware detection framework using lightweight semantic embedding," *Comput. Secur.*, vol. 140, May 2024, Art. no. 103807. doi: 10.1016/j.cose.2024.103807.

[23] T. Bhatia and R. Kaushal, "Malware detection in android based on dynamic analysis," presented at the Int. Conf. Cyber Securi. Protect. Dig. Serv. (Cyber Secur.), London, UK, Jun. 19–20, 2017, pp. 1–6.

[24] V. M. Afonso, M. F. de Amorim, A. R. A. Grégio, G. B. Junquera, and P. L. de Geus, "Identifying Android malware using dynamically obtained features," *J. Comput. Virol. Hacking Tech.*, vol. 11, no. 1, pp. 9–17, Feb. 2015. doi: 10.1007/s11416-014-0226-7.

[25] H. Cai, N. Meng, B. Ryder, and D. Yao, "DroidCat: Effective android malware detection and categorization via app-level profiling," *IEEE Trans. Inf. Forens. Secur*, vol. 14, no. 6, pp. 1455–1470, Jun. 2019. doi: 10.1109/TIFS.2018.2879302.

[26] A. Zulkifli, I. R. A. Hamid, W. Md Shah, and Z. Abdullah, "Android malware detection based on network traffic using decision tree algorithm," presented at the Int. Conf. Adv. Comput. Sci. Inf. Technol. (ACSIT), Kuching, Malaysia, Dec. 10–11, 2016, pp. 485–494.

[27] Q. He, Z. Chen, A. Yan, L. Peng, C. Zhao and Y. Shi, "TrafficPSSF: A fast and effective malware detection under online and offline conditions," presented at the Int. Conf. Secur., Pattern Anal., Cybernet. (SPAC), Wuhan, China, Dec. 15–17, 2018, pp. 14–19.

[28] R. Surendran, T. Thomas, and S. Emmanuel, "A TAN based hybrid model for android malware detection," *J. Inf. Secur. Appl.*, vol. 54, no. 3, Oct. 2020, Art. no. 102483. doi: 10.1016/j.jisa.2020.102483.

[29] S. Arshad, M. A. Shah, A. Wahid, A. Mehmood, H. Song and H. Yu, "SAMADroid: A novel 3-level hybrid malware detection model for android operating system," *IEEE Access*, vol. 6, pp. 4321–4339, 2018. doi: 10.1109/ACCESS.2018.2792941.

[30] S. Chen, M. Xue, Z. Tang, L. Xu, and H. Zhu, "StormDroid: A streaminglized machine learning-based system for detecting android malware," presented at the 11th ACM Asia Conf. Comput. Commun. Secur. (ASIA CCS), New York, NY, USA, May 30–Jun. 3, 2016, pp. 377–388.

[31] Z. Li and Y. Zhou, "Ensemble-learning-based android malware detection using hybrid features," presented at the Int. Conf. Innov. Artif. Intell. (ICIAI '24), New York, NY, USA, Aug. 5–7, 2024, pp. 154–158.

[32] P. Thakur, V. Kansal, and V. Rishiwal, "Hybrid deep learning approach based on LSTM and CNN for malware detection," *Wirel Pers. Commun.*, vol. 136, no. 3, pp. 1879–1901, Jun. 2024. doi: 10.1007/s11277-024-11366-y.

[33] Androguard. Accessed: May 06, 2024. [Online]. Available: https://github.com/androguard/androguard

[34] L. C. Freeman, "Centrality in social networks conceptual clarification," *Soc. Netw.*, vol. 1, no. 3, pp. 215–239, Jan. 1978. doi: 10.1016/0378-8733(78)90021-7.

[35] R. Zafarani, M. A. Abbasi, and H. Liu, *Social Media Mining: An Introduction*. Cambridge: Cambridge University Press, 2014. doi: 10.1017/CBO9781139088510.

[36] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, vol. 393, no. 6684, pp. 440–442, Jun. 1998. doi: 10.1038/30918.

[37] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," Sep. 6, 2013, *arXiv:1301.3781*.

[38] X. Rong, "Word2vec parameter learning explained," Jun. 5, 2016. doi: 10.48550/arXiv.1411.2738.

[39] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," Oct. 16, 2013. doi: 10.48550/arXiv.1310.4546.

[40] C. D. Xuan, "A new approach to software vulnerability detection based on CPG analysis," *Cogent Eng.*, vol. 10, no. 1, Dec. 2023, Art. no. 2221962. doi: 10.1080/23311916.2023.2221962.

[41] C. D. Xuan and D. Huong, "A new approach for APT malware detection based on deep graph network for endpoint systems," *Appl. Intell.*, vol. 52, no. 12, pp. 14005–14024, Sep. 2022. doi: 10.1007/s10489-021-03138-z.

[42] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," Sep. 10, 2018. doi: 10.48550/arXiv.1706.02216.

[43] M. Zhang and Y. Chen, "Link prediction based on graph neural networks," presented at the 32nd Conf. Neur. Inf. Process. Syst. (NeurIPS), Montréal, Canada, Dec. 3–8, 2018, pp. 5165–5175.

[44] S. Xiao, S. Wang, Y. Dai, and W. Guo, "Graph neural networks in node classification: Survey and evaluation," *Mach Vis. Appl.*, vol. 33, no. 1, Nov. 2021, Art. no. 4. doi: 10.1007/s00138-021-01251-0.

[45] D. He *et al.*, "Community-centric graph convolutional network for unsupervised community detection," presented at the 29th Int. Joint Conf. Artif. Intell., Yokohama, Japan, Jul. 11–17, 2020, pp. 3515–3521.

[46] M. Balcilar, G. Renton, P. Heroux, B. Gauzere, S. Adam and P. Honeine, "Bridging the gap between spectral and spatial domains in graph neural networks," Mar. 25, 2020. doi: 10.48550/arXiv.2003.11702.

[47] P. Maniriho, A. N. Mahmood, and M. J. M. Chowdhury, "A survey of recent advances in deep learning models for detecting malware in desktop and mobile platforms," *ACM Comput. Surv.*, vol. 56, no. 6, pp. 145:1–145:41, Jan. 2024. doi: 10.1145/3638240.

[48] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "AndroZoo: Collecting millions of Android apps for the research community," presented at the 13th Int. Conf. Min. Softw. Reposit., Austin, TX, USA, May 14–15, 2016, pp. 468–471.

[49] "VirusShare.com," Accessed: Sep. 20, 2024. [Online]. Available: https://virusshare.com/

[50] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," Feb. 22, 2017. doi: 10.48550/arXiv.1609.02907.

[51] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò and Y. Bengio, "Graph attention networks," Feb. 4, 2018. doi: 10.48550/arXiv.1710.10903.

[52] J. Du, S. Zhang, G. Wu, J. M. F. Moura, and S. Kar, "Topology adaptive graph convolutional networks," Feb. 11, 2018. doi: 10.48550/arXiv.1710.10370.

[53] N. Daoudi, J. Samhi, A. K. Kabore, K. Allix, T. F. Bissyandé and J. Klein, "DexRay: A simple, yet effective deep learning approach to android malware detection based on image representation of bytecode," presented at the Deployable Mach. Learn. Secur. Def. Conf., Cham, Switzerland, 2021, pp. 81–106.

[54] Y. Wu, X. Li, D. Zou, W. Yang, X. Zhang and H. Jin, "Fast market-wide mobile malware scanning by social-network centrality analysis," presented at the 34th IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE), San Diego, CA, USA, Nov. 11–15, 2019, pp. 139–150.

[55] A. Mathur, L. M. Podila, K. Kulkarni, Q. Niyaz, and A. Y. Javaid, "NATICUSdroid: A malware detection framework for Android using native and custom permissions," *J. Inf. Secur. Appl.*, vol. 58, no. 1, May 2021, Art. no. 102696. doi: 10.1016/j.jisa.2020.102696.

[56] O. Yildiz and I. A. Doğru, "Permission-based android malware detection system using feature selection with genetic algorithm," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 29, no. 2, pp. 245–262, Feb. 2019. doi: 10.1142/S0218194019500116.

[57] N. McLaughlin *et al.*, "Deep android malware detection," presented at the 7th ACM Conf. Data Appl. Secur. Priv. (CODASPY '17), New York, NY, USA, Mar. 22–24, 2017, pp. 301–308.