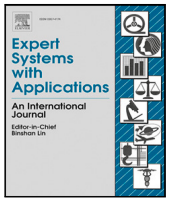
Expert Systems With Applications 237 (2024) 121617

Contents lists available at ScienceDirect

Expert Systems With Applications

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



Android malware detection method based on graph attention networks and deep fusion of multimodal features

Shaojie Chen [a,*], Bo Lang [a,b], Hongyu Liu [a], Yikai Chen [a], Yucai Song [a]

[a] State Key Laboratory of Software Development Environment, Beihang University, Beijing, 100191, China
[b] Zhongguancun Laboratory, Beijing, China

ARTICLE INFO

Keywords:
Android malware detection
Topic model
Class-set call graph
Graph attention networks
Multimodal feature fusion

ABSTRACT

Currently, Android malware detection methods always focus on one kind of app feature, such as structural, semantic, or other statistical features. This paper proposes a novel Android malware detection method that integrates multiple features of Android applications. First, to effectively extract the structural and semantic features, we propose a new type of call graph named the class-set call graph (CSCG) that uses the sets of Java classes as nodes and the call relationships between class sets as edges, and we design a dynamic adaptive CSCG construction method that can automatically determine the node size for applications with different scales. The topic model is used to mine the source code semantics from the class sets as the node features. Then, we use a graph attention network (GAT) with max pooling to extract the CSCG feature that encompass both the semantic and structural features of the Android application. Furthermore, we construct a deep multimodal feature fusion network to fuse the CSCG features with permission features. Experimental results show that our method achieves a detection accuracy of 97.28%–99.54% on the three constructed datasets, which is better than the existing methods.

1. Introduction

The Android system was first launched in 2008, and the number of smartphones with an Android system exceeded 3 billion in 2021. In July 2015, the total number of Android applications in the Google Play Store reached more than 1.6 million, surpassing the number in the iOS App Store. As the number of applications has increased, the amount of Android malwares has also rapidly increased. According to 360's mobile security report (360 Security Center, 2022), the number of new Android malware discovered in 2021 reached 9.4 million. Therefore, studies of Android malware detection methods have become particularly important.

Currently, the main installation file format of Android applications is APK (Android application package). Although Google enabled a new file format AAB (Android APP Bundles) (Beebom Staff, 2021) in 2018, the AAB format is only designed for the uploading of applications by developers, while the installation file downloaded by users is still in the APK format. Therefore, this paper focuses on malware detection for APK format files.

Malware detection methods on different platforms, such as Windows and Android, can be divided into 3 categories, i.e. static methods, dynamic methods, and hybrid methods, based on whether it is necessary to execute applications in feature extraction (Odusami et al., 2018; Qiu et al., 2020). For Android malware detection, static methods extract features from source codes obtained by decompilation. Dynamic methods extract features by running APK files in an actual or simulated environment, and then obtaining the corresponding behavior pattern. Hybrid methods integrate static and dynamic features to describe various aspects of an application. Overall, dynamic methods generally provide high detection accuracy and ensure the detection capability of unknown malware. However, the detection process is complicated and requires extensive computational resources. In addition, dynamic methods usually cannot traverse all possible execution paths of an application, which may lead to missing reports. Static methods do not need to run the applications, resulting in high detection efficiency and effective use in large-scale Android application detection. Therefore, this paper focuses on a new static detection method.

There are three main kinds of static methods: graph methods that represent relations between components of APK files and implement detection based on the graph structure, semantic feature-based methods that extract the semantic features of the APK codes, and feature

* Corresponding author.
E-mail addresses: chenshaojie@buaa.edu.cn (S. Chen), langbo@buaa.edu.cn (B. Lang), liuhongyu@buaa.edu.cn (H. Liu), yk_chen@buaa.edu.cn (Y. Chen), songyucai@buaa.edu.cn (Y. Song).

https://doi.org/10.1016/j.eswa.2023.121617
Received 26 September 2022; Received in revised form 18 March 2023; Accepted 11 September 2023
Available online 20 September 2023

engineering-based methods that use permission, intent and other statistical features. For graph-based methods, Gao, Cheng, and Zhang (2021) and Hei et al. (2021) built a homogeneous information network (HIN) to represent the relationship between applications and their components (such as API and permissions) in one dataset. Then, the relationships between applications are used to detect malware. Most graph methods extract the structural feature of an APK file and use this feature for detection. This kind of method builds a control flow graph (CFG) or a function call graph (FCG) (Allix, Bissyandé, Jérome et al., 2016; Xu, Ren, Qin, & Craciun, 2018; Xu, Ren, & Song, 2019), etc., based on the relationships of internal code regions, such as Java functions or classes, and uses models such as graph convolutional networks to obtain the structural features and implement classification. As malicious behaviors of malware are usually contained in specific code regions, i.e., graph nodes, it is important to fuse the semantic features of codes in the detection. However, in existing graph methods, the semantic features of nodes are not considered. Some graph methods use node features that are statistical features rather than code semantic features (Vinayaka & Jaidhar, 2021; Xu et al., 2017). Milosevic, Dehghantanha, and Choo (2017) and our group in a previous work (Song, Chen, Lang, Liu, & Chen, 2019) proved that semantic features of the code could well reflect the malicious behavior of Android malware. However, these methods only focus on the semantic features without considering the structural information.

To address these problems, this paper proposes a method that combines the structural features, semantic features and permission features of the codes. First, we construct a class-set call graph (CSCG) based on the Java package structure of an APK file. Class-sets represent code regions, and topic vectors extracted from the code regions represent the semantic information and become the features of nodes. Then we use GAT with max-pooling to extract CSCG features, which fuses the semantic features of salient regions and the structural information of the APK file. Finally, CSCG features and permission features are fused by a feature fusion network.

In the construction of CSCG, many Java functions contain few words (perhaps only 1 return statement), resulting in highly random topic vectors. Therefore, we first construct a call graph with the classes from the Java source code instead of FCG, which is called the class call graph. Since each class in Java code is usually associated with an independent file, a class here equals a Java file. Compared with an FCG, the class call graph can effectively increase the information stored in one node. Meanwhile, we note that many applications have a large number of classes that are organized into Java packages. In the Java language, classes in a single package have similar semantics (Arnold, Gosling, & Holmes, 2005). Therefore, to enhance the semantic information of each node, we propose an adaptive class merging algorithm to construct a class-set call graph (CSCG) that uses the sets of Java classes as nodes. CSCG can also greatly decrease the number of nodes, thereby reducing the computational load of the call graphs.

The scale of a CSCG is related to the scale of the APK files and the computational resources. To effectively control the scale of CSCG and ensure that more information in the original call graph is retained, we also set an upper bound and a lower bound of the number of nodes. Codes of third-party libraries generally exist in Android source code; these codes are usually common codes that provide ready-made implementations of specific functionality. These features may constitute noise for malware detection (Aafer, Du, & Yin, 2013; Song et al., 2019; Zhan et al., 2020), thereby affecting the detection effects. However, the code in third-party libraries may also contain malicious behaviors (Li et al., 2017), so we could not directly remove the third-party library. To reduce the influence of the third-party libraries, we prioritize merging the classes in third-party libraries to reduce the proportion of third-party library nodes in the CSCG. In addition, the scale of a call graph is an important piece of information. We adopt proportional reduction in constructing CSCG to retain the scale differences between the class call graphs.

In summary, the contributions of this paper are as follows:

1. The paper proposes a multimodal deep feature fusion method for Android malware detection. We design a new type of call graph (i.e., the class-set call graph) that integrates structural information with the semantic features of the potential topics in the corresponding codes, and builds a deep fusion network to combine graph features and permission features. The proposed method achieves a detection accuracy of 97.28%–99.54% on three open datasets of Android malware, which is higher than that of the state-of-the-art methods.

2. A class-set call graph (CSCG) construction method with an adaptive node size is proposed. The method adopts the concept of proportional reduction to dynamically calculate the number of classes in a single graph node and designs a top-down iterative class merging algorithm based on the tree-like package structure of Java classes. Thus, the constructed call graph can enrich the semantic information of source codes and reduce the scale of the call graph, thereby decreasing the computational loads.

3. A topic model is used to extract semantic features from the source code, and GAT with max pooling is introduced to extract the features of CSCG. In addition, a deep feature fusion network is proposed to achieve the effective fusion of graph features and permission features.

The remainder of the paper is organized as follows. Section 2 introduces the related works focused on Android malware detection. Section 3 introduces the research motivation and the detection framework based on the graph attention network, topic models, and deep fusion network. Section 4 shows the construction of the applied datasets and the experimental results of the model based on these datasets; then, our method is compared with other state-of-the-art methods. Section 5 summarizes the paper.

2. Related work

Android malware detection methods can be divided into static methods, dynamic methods, and hybrid methods from the perspective of feature sources. Static methods extract features without running APK files and perform classification. Many static methods extract features by feature engineering and make classifications using machine learning or deep learning models. Another type of method extracts semantic features to mine semantic information from source code for detection. In addition, some methods use graph methods to extract the structural features of an application or analyze the association between different applications. Dynamic methods extract features by running APK files in an actual or simulated environment. Hybrid methods integrate static and dynamic features to describe various aspects of an application.

The proposed method is a static method, therefore, in this section, we mainly focus on the static detection methods applied in related works. Our method is based on a graph model, so we divide static methods into methods based on feature engineering and sematic features, and methods based on graph models.

2.1. Static methods based on feature engineering and semantic features

Most static methods decompress the APK files to obtain specific files, including “AndroidManifest.xml” and “classes.dex”, and then extract the corresponding features from them. Among these files, the “AndroidManifest.xml” file is used to configure some important information, such as package names, permissions, and program components. Features such as permissions (Milosevic et al., 2017; Şahin, Kural, Akleylek, & Kılıç, 2021; Talha, Alper, & Aydin, 2015; Yuan, Tang, Sun, & Liu, 2020), filter intents (Arp et al., 2014; Cai, Li, & Xiong, 2021; Xu, Li, & Deng, 2016), and hardware components (Arp et al., 2014) can be extracted from this file. The “classes.dex” file is the execution file of the application, and there are usually three methods for processing this kind of file: (1) convert it into Smali language codes with disassembly methods and extract features such as API calls (Aafer et al., 2013; Alazab, Alazab, Shalaginov, Mesleh, & Awajan, 2020;

Seraj, Khodambashi, Pavlidis, & Polatidis, 2022; Shen, Del Vecchio, Mohaisen, Ko, & Ziarek, 2018; Taheri et al., 2020; Yerima, Sezer, & Muttik, 2015; Zhu, Gu, Wang, Xu, & Sheng, 2023), control flow graphs (Allix, Bissyandé, Jérome et al., 2016; Xu et al., 2018, 2019), function call graphs (Xu et al., 2018, 2019), and Android intents (Feizollah, Anuar, Salleh, Suarez-Tangil, & Furnell, 2017; Idrees, Rajarajan, Conti, Chen, & Rahulamathavan, 2017; Kouliaridis, Potha, & Kambourakis, 2020; Taheri et al., 2020); (2) obtain the Java source codes with decompilation methods and then use text analysis or other methods to extract features (Milosevic et al., 2017; Song et al., 2019); and (3) directly regard the binary codes of "classes.dex" as original features (Amin et al., 2020; Hsien-De Huang & Kao, 2018; Ren, Wu, Ning, Hussain, & Chen, 2020; Yadav, Menon, Ravi, Vishvanathan, & Pham, 2022; Yuan, Wang et al., 2020; Zhu, Wei, Wang, Xu, & Sheng, 2023). Some of the methods, such as control flow graphs and function call graphs, are based on graph models, and these methods are discussed in Section 2.2.

Traditional static methods usually extract global feature vectors from APK files and then classify the feature vectors with machine learning models. Common features mainly include permissions (Milosevic et al., 2017; Şahin et al., 2021; Seraj et al., 2022; Talha et al., 2015; Yuan, Tang et al., 2020; Zhu, Gu et al., 2023), intents (Feizollah et al., 2017; Idrees et al., 2017; Kouliaridis et al., 2020; Taheri et al., 2020), intent filters (Arp et al., 2014; Cai et al., 2021; Feizollah et al., 2017; Xu et al., 2016), API calls (Aafer et al., 2013; Alazab et al., 2020; Shen et al., 2018; Taheri et al., 2020; Yerima et al., 2015; Zhu, Gu et al., 2023), and semantic features (Milosevic et al., 2017; Song et al., 2019). Permission features are extracted by analyzing the extracted permission set to select the important permissions as features. Intent and intent filters are also commonly used features for detection. Specifically, intent information is obtained from the "classes.dex" file that encapsulates the calling intents of the application, such as starting an activity or service. Intent filters are defined in the "AndroidManifest.xml" file and are used to specify the type of intents that the application can receive (Feizollah et al., 2017). API call features are usually extracted from the sequences of API calls in the "classes.dex" file. Taheri et al. (2020) extracted permission features, API features, and intent features from APK files to form a binary feature vector. After feature selection with a random forest regressor, detection was performed through similarity calculations.

Semantic features (Milosevic et al., 2017; Song et al., 2019) and n-gram features from the byte stream (Karbab, Debbabi, Derhab, & Mouheb, 2020) are used to detect Android malware. Milosevic et al. (2017) first obtained Java source code, and the bag-of-words method was used to extract features from the source code. Then, an SVM and other models were used to perform binary classification. Song et al. (2019) segmented the decompiled Java source codes after removing the classes from third-party libraries and constructed a topic model to extract the topic vector as features for each sample. Then, SVM and XGBoost were used to perform binary classification. Karbab et al. (2020) extracted permissions, API features, and the n-gram features of bytecodes from the "classes.dex" files or APK files and classified them using a clustering method after dimensionality reduction.

Several other features are used for Android malware detection, namely, hardware components (Arp et al., 2014; Zhu, Gu et al., 2023), app components (Arp et al., 2014; Xu et al., 2016), network addresses (Arp et al., 2014), and Dalvik instructions (Chen, Mao, Yang, Lv, & Zhu, 2018). The Drebin method proposed by Arp et al. (2014), which is based on eight types of basic features, including hardware components and permissions, uses feature embedding to form a feature vector; then, classification is performed based on an SVM model. Chen et al. (2018) extracted Dalvik instructions from Smali code and represented the APK files as sequences of ten Dalvik instruction types. After extracting n-gram sequences as features, random forest and other models were used to perform binary classification and malware family classification. Wang, Zhao, and Wang (2019) extracted seven types of static features, including permissions, intent filters, and API calls, and then combined a deep autoencoder and a CNN (convolutional neural network) to build a hybrid detection model to classify the features. Kim, Kang, Rho, Sezer, and Im (2018) extracted five kinds of features, including API calls, permissions, and components, and used a multimodal feature fusion network for binary classification. Zhu, Gu et al. (2023) proposed MSerNetDroid method; they extracted permission, hardware and API features, transformed the features into an image, and perform classification with MSerNet they proposed.

The byte codes from an APK file (or "classes.dex" file) are malware base data and can be directly classified by deep learning methods (Amin et al., 2020; Daoudi et al., 2021; Hsien-De Huang & Kao, 2018; Ren et al., 2020; Yadav et al., 2022; Yuan, Wang et al., 2020; Zhu, Wei et al., 2023). Daoudi et al. (2021) proposed the DexRay method; they converted APK files into 2-dimensional images according to the value of byte-code and used a CNN model for classification. Yuan, Wang et al. (2020) first read the byte sequence from the APK file and built a Markov transfer probability matrix with a size of 256*256 based on the neighboring relationships among the bytes; then, a CNN model was used for malware family classification. Zhu, Wei et al. (2023) removed headers and data portions, transformed index portion into an image, and perform classification with the MADRF-CNN model they proposed.

In addition to images, sequence (McLaughlin et al., 2017) or text data (Zhang, Tan, Yang, & Li, 2021) can be used to perform classifications by deep learning models. McLaughlin et al. (2017) extracted opcode sequences from the APK files and used a CNN model for classification. Zhang et al. (2021) proposed the TC-Droid method. They built text information by extracting four types of features, including permissions, services, intent and receiver, and then used the TextCNN model for classification.

These methods usually only extract global features, which lack local detailed features. However, the malicious behaviors of malware are usually limited to specific regions, it is difficult to extract salient region features of applications. Some kinds of features are important for malware detection, such as permission features and semantic features; therefore, we reserved these two kinds of features in our detection method.

2.2. Static methods based on the graph model

Compared with methods based on feature engineering or semantic features, graph methods can mine the relationship between components or different applications. There are two kinds of graphs that can be used, in which a homogeneous information network (HIN) can mine relationships between different applications and their components, and a graph of a single application can represent the relationships between components in an application (Qiu et al., 2020).

For methods based on HIN (Gao et al., 2021; Hei et al., 2021), Gao et al. (2021) constructed one complete graph with all available data. In the graph, applications and their APIs were the nodes, and the neighboring relationships between the APIs and call relationships between the applications and APIs were edges. A GCN was used to detect malware by detecting abnormal nodes in the graph. Hei et al. (2021) extracted permission, API, class, interface, etc., from applications and use the components and application as nodes in the HIN. They extract features of applications by node embedding models from the HIN and then perform classification.

For methods based on graphs of internal components, control flow graphs (Allix, Bissyandé, Jérome et al., 2016; Wu, Wang, Li, & Zhang, 2016; Xu et al., 2018, 2019), data flow graphs (Xu et al., 2018, 2019), API call graphs (Pektaş & Acarman, 2020), and function call graphs (Vinayaka & Jaidhar, 2021) are widely used by researchers. The CSBD method proposed by Allix, Bissyandé, Jérome et al. (2016) extracted the control flow graphs of APK files, extracted basic blocks from the graphs, and then represented each APK file as a set of basic blocks. A binary feature vector was constructed according to whether there was a certain basic block in each sample. After feature selection,

models such as C4.5 and a random forest were used to perform binary classification. Xu et al. (2018) extracted the control flow graph and data flow graph from an APK file, combined them with weights, and classified the adjacency matrix of the graphs using a CNN model. Vinayaka and Jaidhar (2021) used methods in an APK file as nodes to construct a function call graph and used degree features, method attributes and method summary information as node features. Then, they extracted node features with graph convolutional networks, such as GCNs and GATs, used the mean value of each node feature as the graph feature, and performed binary classification. In addition to the graphs above, there are some other types of graph construction. Pei et al. proposed the AMalNet method (Pei, Yu, & Tian, 2020) and extracted the m-dimensional features of words, characters, and lexical features from the structure, component and API information in the APK file. Then, a graph was built based on m-dimensional features, with each dimensional feature acting as a node. Finally, they used a GCN to classify the malware types.

In many detection methods based on call graphs, the graphs only contain the relationships among the nodes, thus excluding node features (Pektaş & Acarman, 2020; Xu et al., 2018, 2019). However, the malicious behaviors of malware are usually limited to specific regions, and these methods cannot accurately represent salient region features. Vinayaka and Jaidhar (2021) introduced three types of node features, but the features were obtained based on simple statistics with limited semantic information. In this paper, the topic model method that we previously proposed (Song et al., 2019) is used to construct a class-set call graph, and topic vectors are used as node features. Then, salient region features are obtained from the graph by GAT networks with the max pooling method. This approach effectively combines semantic features and structural information.

2.3. Dynamic methods and hybrid methods

Dynamic methods extract features by running APK files in an actual or simulated environment, and the most common dynamic features include API calls (Jerbi, Dagdia, Bechikh, & Said, 2020), system calls (Burguera, Zurutuza, & Nadjm-Tehrani, 2011; John, Thomas, & Emmanuel, 2020; Surendran, Thomas, & Emmanuel, 2020; Xiao, Zhang, Mercaldo, Hu, & Sangaiah, 2019; Zhou et al., 2019), kernel calls (Wang & Li, 2021), and network behaviors (Wang, Chen et al., 2019; Wang et al., 2017). Hybrid methods (Alzaylaee, Yerima, & Sezer, 2020; Arshad et al., 2018; Han, Subrahmanian, & Xiong, 2020; Zhou, Wang, Zhou, & Jiang, 2012) integrate static and dynamic features to describe various aspects of an application.

Most dynamic methods extract features such as API calls, system calls, kernel calls, and network behaviors by executing applications and use machine learning models for classification. Dynamic methods can be divided into methods based on traditional machine learning and methods based on deep learning.

In traditional machine learning-based dynamic methods, the features mainly include system call features (Burguera et al., 2011; Surendran et al., 2020), kernel call features (Wang & Li, 2021), and network traffic features (Wang, Chen et al., 2019; Wang et al., 2017). For system calls and kernel calls, features are commonly extracted using the following methods: by counting the number of occurrences of each call type (Burguera et al., 2011), by building a call graph based on the method call sequences for detection (Surendran et al., 2020), and by extracting kernel parameters while the machine learning model runs (Wang & Li, 2021). For network traffic, a data stream can be segmented, and methods such as n-gram can be used for feature extraction (Wang et al., 2017).

Among the dynamic methods based on deep learning, most of the methods extract feature vectors from applications and perform classifications based on deep learning models. The general approaches include classifying system call features using deep learning models (Zhou et al., 2019), directly using system call sequences as natural language sentences for classification (Xiao et al., 2019), and using a graph convolutional network to obtain the corresponding system call graph (John et al., 2020).

Hybrid methods separately extract static and dynamic features from applications to obtain comprehensive features from various perspectives (Alzaylaee et al., 2020; Han et al., 2020; Zhou et al., 2012). Han et al. (2020) obtained 120-dimensional static features, 171-dimensional API package call features, and 767-dimensional dynamic features; then, they proposed three different feature transformation methods and classified features using a random forest model.

3. Detection method with multimodal feature fusing

In this section, the motivation of our method and the overall structure of the model are first presented. Then, the construction method of the class-set call graph (CSCG) is described in detail. Finally, we introduce graph feature extraction of CSCG based on GAT, and the fusion of the CSCG features with permission features using a deep network.

3.1. Motivation

In static Android malware detection, the common features include permission features from the “AndroidManifest.xml” file and semantic and structural features from the “classes.dex” file. Different feature types can support different analysis perspectives. The fusion of multimodal features can effectively represent the characteristics of malware and may yield better detection results than using a single type of feature. Therefore, this paper focuses on the deep fusion of multimodal features, as shown in Fig. 1.

Our previous work (Song et al., 2019) verified that topic models can effectively extract code semantic features. However, this method extracts a global topic vector from an application. Since the malicious behavior of malware usually occurs in specific code regions, global topic features cannot accurately represent salient region features. Therefore, we construct a call graph with nodes that correspond to the specific units of the application codes, and the topic vectors of the nodes are extracted as the node features. Then, we use a GAT model with max pooling to extract the salient region features as the features of the whole graph, thus effectively fusing the node features and structural information.

Each node in the traditional call graph usually represents a function. However, a Java function may contain too few words to extract effective topic features. Therefore, we first choose each class as a node, further merged some class nodes into class-set nodes and built a class-set call graph (CSCG). Thus, the semantic information of each node is enhanced, and the scale of the call graph is decreased. As shown in Fig. 1, we assume that with an APK file of 17 Java files, after node merging, we can archive a CSCG of eight nodes. The details of the example are shown in Section 3.3.5.

In addition, permission features are widely used and are important for Android malware detection. The graph features from the CSCG and permission features differ greatly and may have low relevance separately. Therefore, we fuse these two kinds of features through a deep feature fusion network and then directly classify the fused features using a deep neural network. For a feature fusion network, Kim et al. (2018) applied a single fusion point network (direct concatenation of different kinds of features) for Android malware detection. In addition, Gu, Lang, Yue, and Huang (2017) experimentally verified the use of multiple fusion points, which takes multiple level concatenation of different types of features and yields better results than using a single fusion point in image pattern recognition. Therefore, we build a multiple-fusion-point network for our detection method.

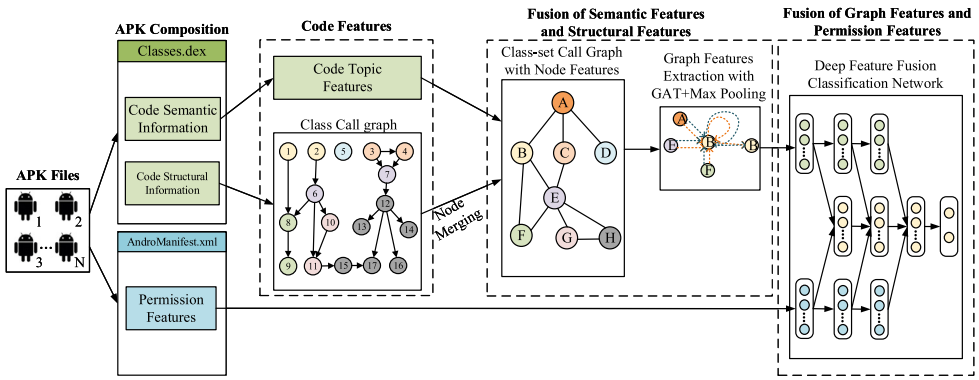


Fig. 1. Overall concept of the detection method.

3.2. Model structure

The structure of our detection method is shown in Fig. 2. The model is divided into four parts: data preprocessing, code topic model training, CSCG construction, graph feature extraction, and multiple feature fusion.

Data preprocessing. We first decompiled the APK file using APKTool (Winsniewski, 2012) to obtain "AndroidManifest.xml" and "classes.dex" files, used LibRadar (Ma, Wang, Guo, & Chen, 2016) to detect the third-party libraries contained in the APK file, and then used JADX (Skylot, 2014) to decompile the "classes.dex" file to obtain the Java source code. Then, according to the 59 high-risk permissions given by StormDroid (Chen, Xue, Tang, Xu, & Zhu, 2016), 59-dimensional permission features were extracted from the "AndroidManifest.xml" file, including the location permission "ACCESS_FINE_LOCATION" and the network permission "ACCESS_WIFI_STATE". There may be multiple permissions with the same name, e.g., the "READ_SETTINGS" permission, and its detailed permissions, including "android.permission.READ_SETTINGS" and "com.android.launcher.permission.READ_SETTINGS". These permissions with the same name may appear in the same "AndroidManifest.xml", so we use the number of occurrences of each permission as the corresponding permission feature.

Code topic model training. This part of the process is consistent with that used in our previous work (Song et al., 2019). First, we removed the codes in third-party libraries from the Java source code and used the Java lexical analyzer to segment the remaining source code. Only identifiers, strings, and numbers were retained in the segmented result. Each sample retained a maximum of 100,000 words. Next, we trained the TF–IDF model based on the training dataset, retained 100,000 words in the dictionary of the TF–IDF model, and obtained the TF–IDF feature vector. Finally, we trained the LSI (latent semantic indexing) model as the topic model, and 500 topics were retained. Then, we calculated the global LSI feature vector. The LSI model and global LSI vector were subsequently used to construct the CSCG.

CSCG construction. For each APK file, we built a class call graph constructed using the public classes of the Java source code. Then, according to the Java package structure and the third-party library detection results, the nodes, i.e., classes, were merged; therefore, each new node represented a class-set. After merging, the CSCG, which decreased the number of nodes to a certain extent, was constructed. Finally, the LSI vector of each class set was calculated to establish the node features. The construction of the CSCG is discussed in detail in Section 3.3.

Graph feature extraction and multiple feature fusion. We used the GAT model with max pooling to extract salient region features from the CSCG as the features of the graph model; this process involves the fusion of the semantic modal and structural modal features. Then, the permission features were used as the permission modal features, and a multiple-fusion-point network was constructed to fuse features and perform classification. One fusion point represents a concatenation of different kinds of features, and multiple fusion points indicate the concatenation of different types of features at multiple levels. Overall, an end-to-end detection method was created that included feature extraction with CSCG, the fusion of multimodal features, and classification. The detection model is introduced in detail in Section 3.4.

Our source code is available at: https://github.com/chenshaojie-happy/Android_Malware_Detection_Method_Based_on_Graph_Attention_Networks.

3.3. Construction of CSCG

In this section, we introduce the CSCG construction method in detail. We first present the overview of our construction method. Then, we introduce the node merging method, and we separately introduce the adaptive node size calculation in the node merging method. After that, we show the node feature extraction and edge construction method. Finally, we give an example of our CSCG construction method.

3.3.1. Overview

We first build the class call graph based on the call relationships among classes. Each class, which is equivalent to a Java file, is regarded as a node in the graph. Then, we merge certain nodes into one node and construct the class-set call graph (CSCG). To enhance the semantic information of each node in CSCG and control the scale of the graph, we set the upper bound ts_max and the lower bound ts_min for the number of nodes when constructing CSCG. ts_max and ts_min can be set based on the computational resources and the APK file size in the dataset. In the Java language, packages create groupings for related interfaces and classes (Arnold et al., 2005). Therefore, in the decompiled Java source code, the semantics of the files in the same package would be more similar than those in different packages. Therefore, based on the Java package structure, we merge the Java files in the same package to ensure semantic consistency in a single class set.

Third-party libraries may interfere with the representativeness of topic vectors; however, codes in third-party libraries may also contain malicious behavior, and removing them may result in incomplete call relationships. Therefore, to reduce the proportion of third-party library nodes, when merging nodes, we try to merge each third-party library into one class-set node and separate the files from non-third-party libraries into multiple class-set nodes as much as possible. That is, when the number of non-third-party files reaches ts_min, each third-party library is directly merged into one class-set node; when the number of non-third-party files is insufficient, the third-party libraries should be separated properly so that the number of class-set nodes reaches ts_min. Thus, the proportion of the class-set nodes corresponding to third-party libraries is decreased, reducing the impact of the third-party libraries.

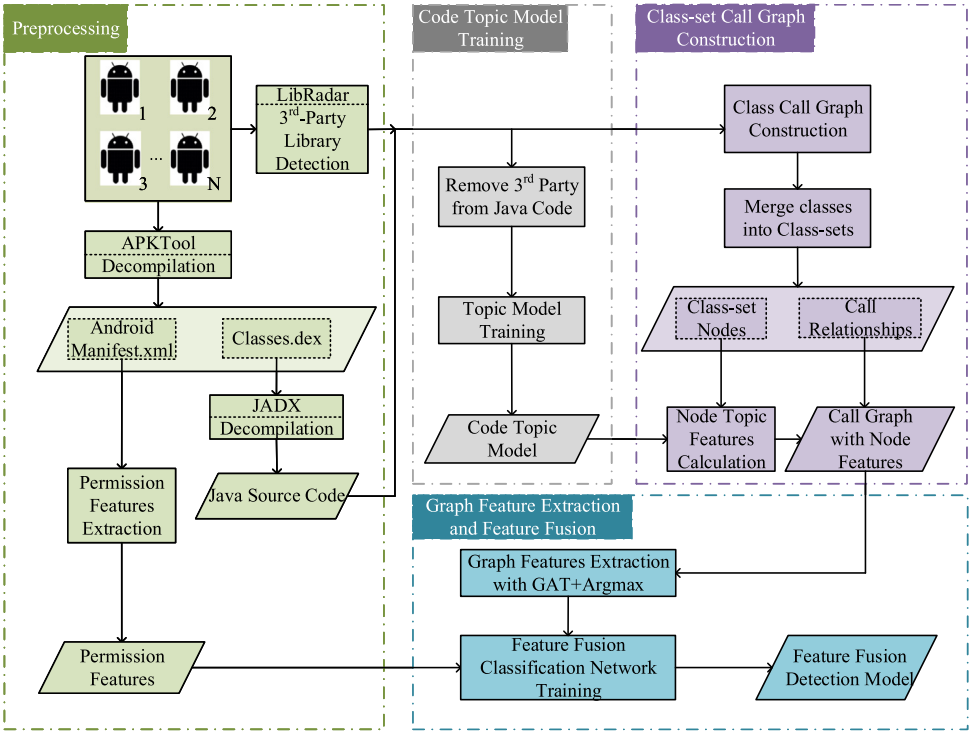


Fig. 2. Structure of the detection method.

The node size k (the number of files/classes contained in a single class-set node) is calculated for each APK file based on ts_max, ts_min and the third-party library detection results to ensure that all non-third-party class-set nodes contain roughly equal amounts of information. Since the scales of the APK files are quite different, to avoid obtaining many large APK files with ts_max nodes in their CSCGs, which may lead to the loss of scale and structural information, we design an adaptive algorithm to automatically determine the value of k for a specific APK file. This algorithm performs proportional reduction for the class call graphs of the APK files, hence effectively preserving the structural and size information associated with each graph.

After obtaining the merged class-set nodes, we calculate the topic vector of the node features. Then, the call relationships among the basic class call graph are mapped to the CSCG as edges.

Next, we introduce the key steps in the construction of the CSCG.

3.3.2. Node merging to construct class-set nodes

To reduce the number of nodes in the code call graph to an acceptable range, we design a class-set construction method that appropriately merges k Java files under the same package into a class-set. Here, k represents the size of a class-set node and is dynamically calculated for each APK file based on the number of Java files and the corresponding thresholds. The calculation process for k will be introduced in Section 3.3.3. Another problem in constructing a CSCG that meets the ts_max and ts_min constraints is controlling the number of third-party library nodes for different sizes of APK files so that the CSCG can represent as much information as possible from non-third-party files.

Our strategy is to ensure that the number of class-set nodes is within the interval [ts_min, ts_max] for each APK file (except for the APK files with fewer Java classes than ts_min). We treat all of the Java files of an APK file as one initial node and gradually split the node until the number of nodes meets the relevant requirement. Based on the structure of Java files, we prioritize splitting the packages of non-third parties. For each non-third-party package, every k files in the package are regarded as a class-set node, and each third-party library package is regarded as a class-set node. Then, if all non-third-party packages are split but the number of class-set nodes is less than ts_min, the third-party packages are appropriately split until the number of class-set nodes reaches ts_min or there is no package left to split.

For this splitting task, one straightforward approach is to perform a depth-first traversal of the Java package structure. However, because the number of files contained in each package is generally not an integer multiple of k, which would result in a large number of class-set nodes with sizes smaller than k, the splitting process may stop too early because of the ts_max limitation. If such early stopping occurs, some large nodes will remain unsplit, resulting in uneven node sizes. To avoid this problem, we iteratively split the largest node that contains the most files without recursive splitting. For each iteration, the selected package is only split into the next-level packages to ensure that the largest node is split in each iteration.

The splitting process is shown in Fig. 3. We first set the first-level packages as the initial class-set nodes, such as “com/” and “android/”, and save the class-set nodes in a list of named $nodes$. In each iteration, the non-third-party nodes are split prior to the third-party nodes; then, the node containing the most files is split, and the splitting result is then updated as $nodes$. When splitting, if a node does not contain next-level packages, then every k files in the node are divided into separate nodes; otherwise, each package will form a node. This iteration is repeated until one of the following conditions is met: (1) there is no node to split, (2) the number of nodes is greater than ts_max, or (3) there is no non-third-party node to split and the number of nodes reaches ts_min.

The number of nodes for the optimal CSCG that we constructed is in the interval [ts_min, ts_max]; additionally, the amount of information contained in all of the nodes is as equal as possible. In practice, we set ts_min to 100 to ensure that there are enough nodes to retain sufficient structural information. We first set ts_max to 1500 according to the computational capability of our server and then adjust it to find the best value for each dataset.

3.3.3. Adaptive class-set node size

The number of public classes contained in different APK files varies greatly, ranging from less than 10 to tens of thousands. Hence, it is not reasonable to use a fixed class-set node size for all APK files.

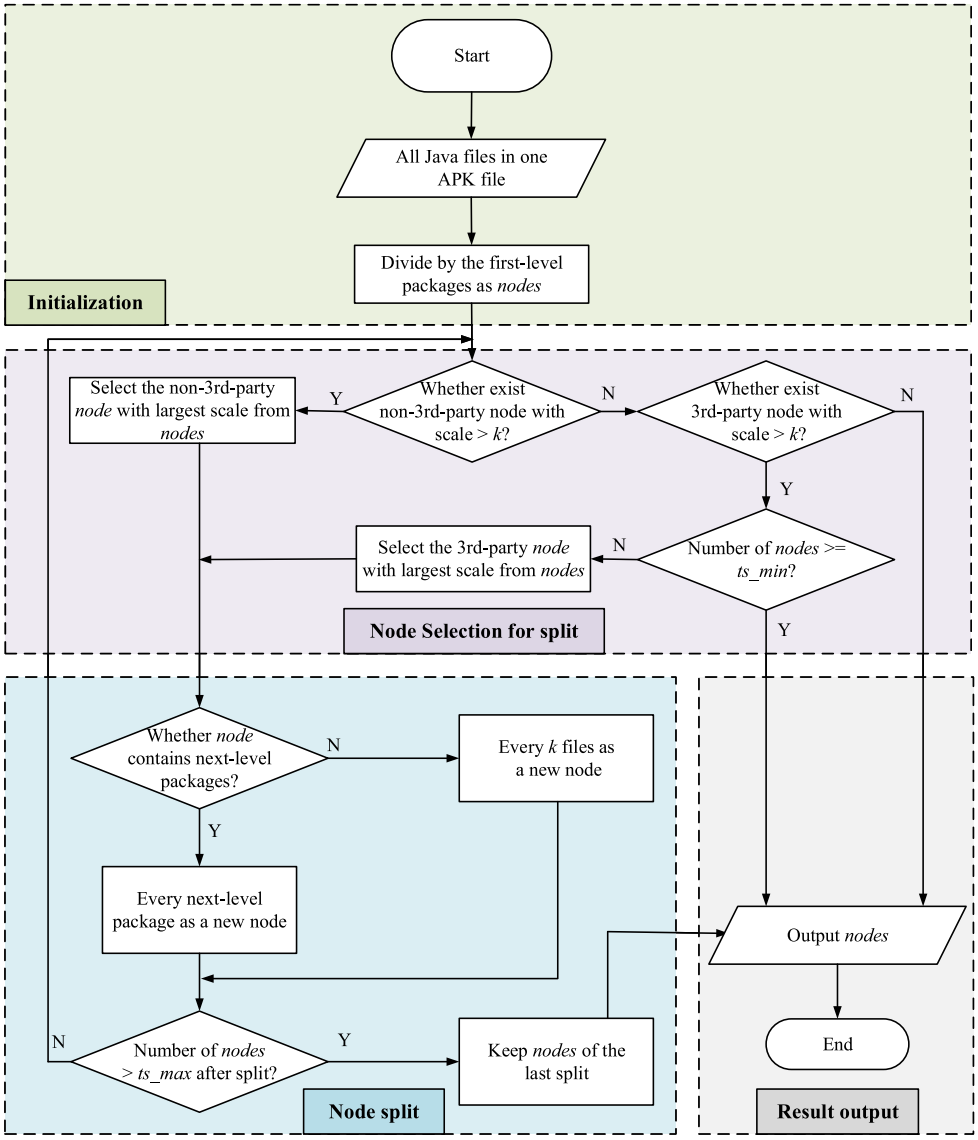


Fig. 3. Flowchart of class-set construction of a single APK file.

The node size k can be related to the reduction ratio between the class call graph and the CSCG. Prior to dividing the Java files into class sets, we first calculate the number of non-third-party Java files $file\_num\_not\_3rd$ and the number of third-party packages pk_num_3rd, and the corresponding sum is named the *sum*. The number of nodes can be in the interval $[ts\_min, ts\_max]$ only if k takes a value in the interval $[\lceil sum/ts\_max \rceil, \lfloor sum/ts\_min \rfloor]$ ($\lceil\ \rceil$ and $\lfloor\ \rfloor$ represent rounding up and rounding down, respectively). Many values of k meet this condition, and for different values of k, the resulting CSCG may differ strongly.

Many APK files contain large numbers of Java files. If the node size is directly calculated as $\lceil sum/ts\_max \rceil$, there will be many CSCGs with ts_max nodes, resulting in the loss of the APK size information. However, through statistical analyses of the relevant data, we found that APK size information is important. Generally, the average size of the malware samples is smaller than that of the benign samples. For instance, the benign samples from Apkpure (Apkpure Team, 2020) included an average of 6589 Java files and a median of 5724 files, and the malware samples from VirusShare (Virusshare, 2020) at a similar time exhibited an average of 1644 files and a median of 388 files. Therefore, we propose a method to achieve the proportional reduction of the call graph at different scales.

When calculating the node size k, min_k is first set according to the distribution of the number of Java files in a single APK file for an entire dataset. min_k is the minimum value of k in a dataset, that is, the minimum reduction ratio of the call graph. Thus, the reduction ratio for each call graph is relatively similar to preserve the scale difference information as much as possible. Then, for each APK file, we dynamically and adaptively calculate k according to the number of Java files, min_k, ts_min and ts_max.

The calculation of min_k is shown in formula (1). We map the number of nodes in the whole dataset to the interval $[0, ts\_max]$. To retain scale differences, we first compute the median of the number of Java files in all APK files, map the median to $ts\_max/2$, divide the median of the whole dataset by $ts\_max/2$, and use the approximate integer as min_k. If min_k is calculated as 0, we set min_k to 1. The $median$ means the median number of Java files in one APK file.

$$min\_k = max(round(median/(ts\_max/2)), 1) \tag{1}$$

For instance, for the VirusShare_Apkpure dataset, the median number of classes is approximately 2812, and ts_max is 1500, and we therefore set $min\_k = 2,812/(1,500/2) \approx 4$ for the dataset.

The calculation of k is shown in formula (2). We should try to map different *sum*s into $[ts\_min, ts\_max]$, and achieve proportional reduction.

For the sum in different intervals, it is necessary to calculate k in different ways. The proportional reduction is mainly performed when $sum \in [\ ts\_min \times min\_k, ts\_max \times min\_k)$. When sum is in other intervals, k is limited by ts_min and ts_max.

$$k = \begin{cases} 1 & if\ \ sum\ \ \in [0, ts\_min) \\ \lfloor sum/ts\_min \rfloor & if\ \ sum\ \ \in [ts\_min, ts\_min \times min\_k) \\ min\_k & if\ \ sum\ \ \in [ts\_min \times min\_k, ts\_max \times min\_k) \\ \lceil sum/ts\_max \rceil & if\ \ sum\ \ \in [ts\_max \times min\_k, +\infty) \end{cases} \tag{2}$$

3.3.4. Node feature extraction and edge construction

After class-set node construction, for each class-set node, a 500-dimensional feature vector is calculated with the topic model.

We then convert the edges in the class call graph to the edges of the corresponding class-set nodes. The call relationships between the nodes in the same class set are ignored, and only one of the repeated relationships is retained.

The CSCG may contain many independent branches or isolated nodes, and the features in different branches cannot be combined through a graph convolutional network, which may result in incomplete features. Therefore, we make the graph a fully connected graph by adding a virtual root node. We use the global LSI features as the root node features and then add the edges from the root node to the topmost nodes of each branch.

In addition, we convert the CSCG to an undirected graph to obtain better graph features by using a graph convolutional network. In graph convolutional networks, each node feature is updated according to its neighboring nodes. If a directed graph is used as the input to the network, each node feature is updated only according to the nodes it calls and is not related to the nodes that call it. However, we believe that both types of call relationships influence the updating of node features; therefore, we convert the CSCG into an undirected graph, and our experimental results verify this approach.

3.3.5. Example of CSCG construction

Here, we show an example of CSCG construction, as illustrated in Fig. 4, in which each class-set node in different steps is marked using the same specific color. In this example, the APK file contains 17 Java files, as shown in Fig. 4. The set of third-party libraries is [“com/_3rd/”], ts_min is 5, ts_max is 20, and min_k for the whole dataset is 2. The construction process is divided into the following steps:

Step 1: Calculate the call relationships among the Java files to obtain the basic class call graph.

Step 2: Calculate the merging relationships among the Java files. First, the k value of the APK file is determined according to Algorithm 1. The sum of the non-third-party files and third-party packages is 12, and the k value is calculated as 2. Then, the class-set construction method is used to merge the nodes, and 7 class-set nodes (B - H) are obtained. Among the class-set nodes, H contains all 6 files in the third-party library, and other nodes contain at most k files.

Step 3: According to the merged class-set nodes, calculate the call relationships between class-set nodes.

Step 4: Word segmentations are performed with the Java lexical analyzer for the 17 classes, and the words corresponding to each class-set node are obtained. Then, calculate the LSI vectors for the 7 class-set nodes.

Step 5: Add the global LSI features as the root nodes A. Since the 3 class-set nodes B, C, and D are not called by any other class-set nodes, we add the edges from A to B, C, and D. Then, the call graph is converted to an undirected graph as the final CSCG.

3.4. Graph feature extraction and multimodal feature fusion

GCNs (Kipf & Welling, 2016) and GATs (Veličković et al., 2017) are commonly used for graph classification. The main objective is to aggregate the features of neighboring nodes and update the features of the current node. The core idea of GCNs is feature decomposition based on the Laplace matrix. GATs use the attention method to calculate the weight of each neighbor node. GCNs can only be used for the convolution of undirected graphs due to the limitation of the Laplace matrix, while GATs can be used for the convolution of directed graphs. Compared with the GCN, by introducing an attention mechanism, the GAT model can better integrate the correlations among the nodes to improve modeling performance.

Therefore, we extract the features of CSCG using a GAT model. The GAT model was proposed by Veličković et al. (2017) and has been used in node feature calculation and node classification applications. To apply a GAT for graph classification, it is necessary to obtain whole graph features based on node features. Knyazev, Lin, Amer, and Taylor (2018) used max pooling to extract whole graph features. Since the malicious behavior of malware usually occurs in specific code regions, using the mean of all node features may mask the features in salient regions. Therefore, max pooling is used to extract features from each dimension in the whole graph. We combine the GAT model and max pooling to propose an adjusted network named Max-GAT. As shown in Fig. 5, Max-GAT extracts the maximum number of features from each dimension through max pooling based on all node features obtained with the GAT, and the results are used as the CSCG features.

In the design of Max-GAT, we experimentally verified that a single-layer GAT performs better than a multilayer GAT. Thus, first-order neighbors have a greater influence on the results than the second-order neighbors for a class-set node, and second-order neighbors may introduce additional noise; therefore, we selected the single-layer GAT model for feature extraction.

In Fig. 5, each CSCG is represented as a n*n adjacency matrix and an n*500 node feature matrix, where 500 is the feature dimension and n is the number of nodes. First, according to the call relationships, the first-order neighbors of each node are identified, and the GAT is used to construct a multi-head attention mechanism, increasing the capacity of the model by applying multiple independent attention mechanisms. In Fig. 5, the number of attention heads is denoted by the number of arrows from one neighboring node to each center node and is adjusted as an important parameter. For each attention head, all of the node features are transformed first, and then for each node, the weighted average of the features of the neighboring nodes is used to obtain updated features. The average of the features obtained by multi-head graph convolution is taken as the final feature. Then, a feature matrix of n*128 is obtained, and through max pooling, the max feature vector (128 dimensions) is selected as the feature vector for the whole graph.

To fuse graph modal and permission modal features, we designed a multilayer deep network, as shown in Fig. 6. The size (output feature dimension) of each layer is indicated in parentheses. For an APK file, the CSCG is first obtained, and then the initial 128-dimensional graph modal feature vector is calculated. The 59-dimensional permission feature vector is selected as the initial permission modal feature vector. A prominent dimension difference between two modal features is not beneficial for feature fusion. Therefore, fully connected layers are introduced for each modal feature to reduce these differences. Then, we build a three-layer neural network with each layer acting as a fusion point and obtain the 64-dimensional fused features from $FC3\_Fus$. The final binary classification result is obtained through the final fully connected layer FC_out.

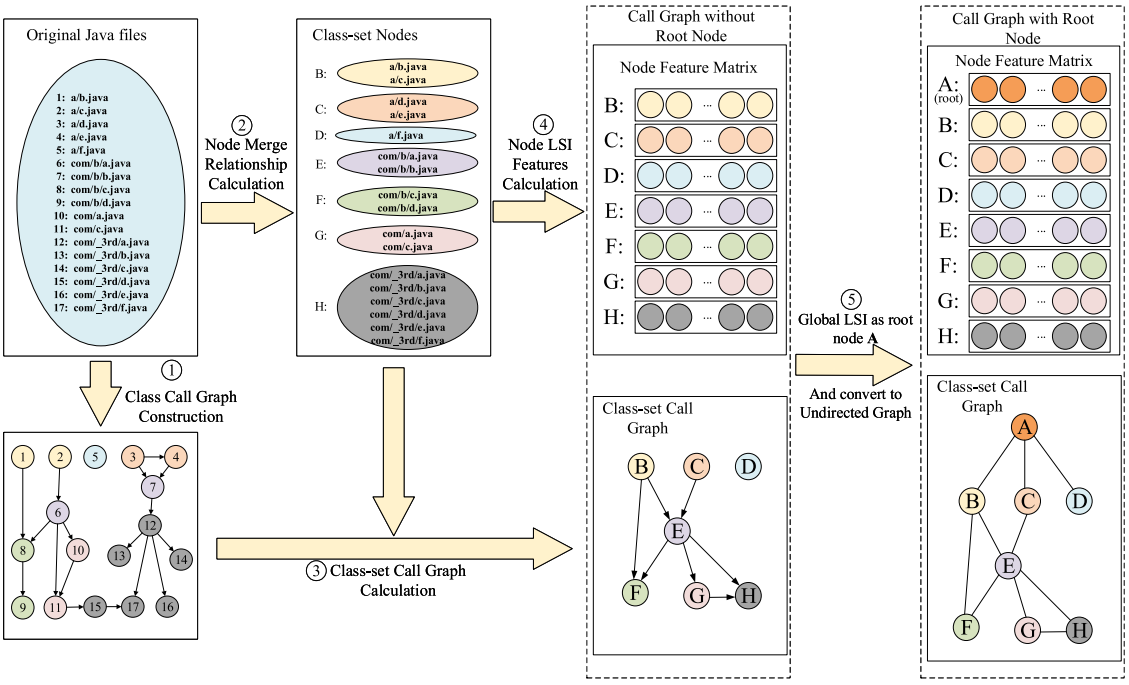


Fig. 4. Example of CSCG construction.

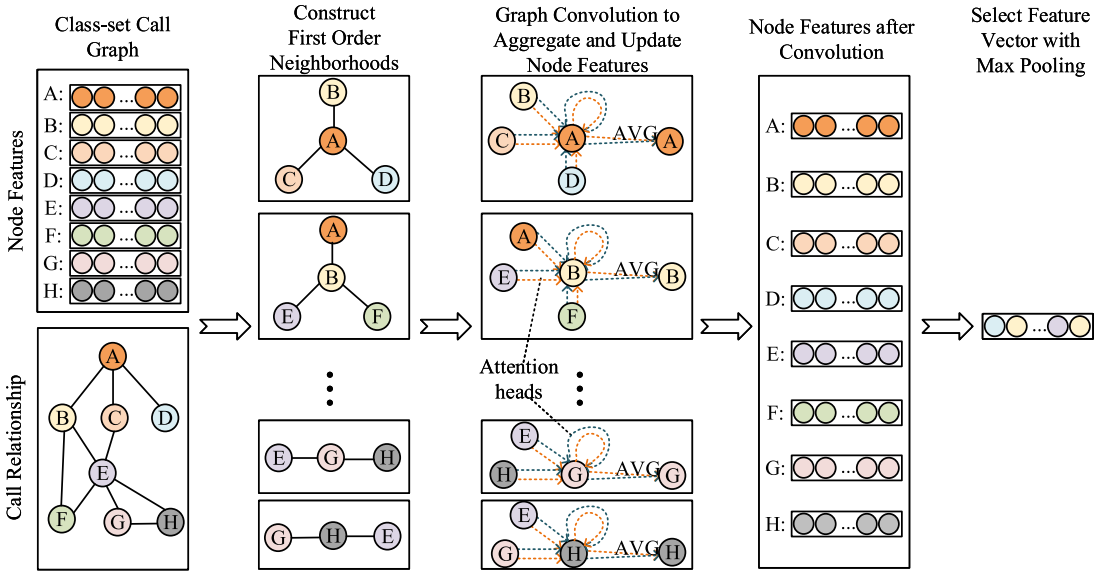


Fig. 5. Feature extraction of CSCG using Max-GAT.

3.5. Time complexity analysis

Our method mainly includes three parts: (1) node merging for CSCG; (2) node feature extraction and edge construction for CSCG; and (3) graph feature extraction, permission feature fusion and classification. The time complexity of these three parts will be calculated separately below. Assume that an APK file contains m Java files, and the total number of nodes in CSCG is n ($n \leq ts\_max$), and the average size of one Java file is s.

The first part is node merging for CSCG. When implementing the node merging method, all Java files in the APK file are sorted alphabetically to ensure the order of internal Java files of each node to make it easier to split nodes. After that, in each iteration we select a node from existing nodes and split it until the end condition is reached. Therefore, its time complexity can be divided into four steps: (1) sorting of Java files, (2) total times of split, (3) selecting a node for split, and (4) splitting the node.

The time complexity of the four steps is as follows: (1) The time complexity of file sorting is nearly $O(m \times log(m))$. (2) According to our construction method, the number of node splits is up to n, corresponding to a time complexity of $O(n)$. (3) For each process of node splitting, we need to traverse the existing nodes to find the node for splitting. The number of nodes increases with the iteration and is less than n, so the time complexity does not exceed $O(n)$. (4) When splitting a node, all files in this node need to be divided into subgroups one by one, the number of files in a node is less than m, and the time complexity does not exceed $O(m)$. Therefore, the overall time complexity of node merging is less than $O(m \times log(m)) + O(n) \times (O(n) + O(m))$; since $n \leq m$, the overall time complexity of class-set construction is less than $O(m \times log(m)) + O(m \times n)$.

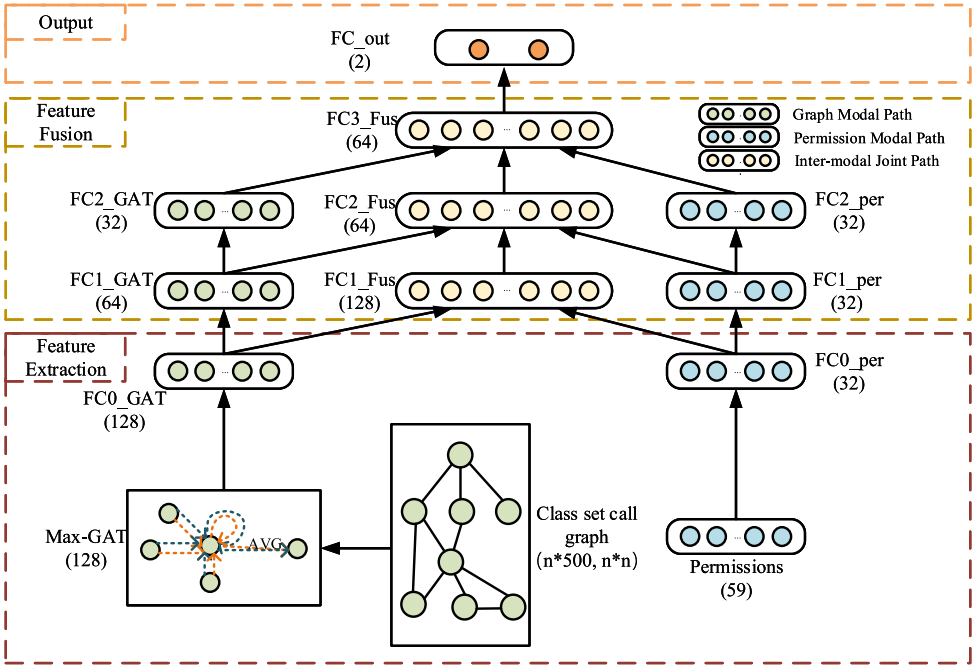


Fig. 6. Structure of the detection model with multimodal feature fusion.

The next part is node feature calculation and edge construction for CSCG. The time complexity of this part can be divided into four steps: (1) performing lexical analysis and calculate calling relationships; (2) merging call relationships; (3) summarizing the words of each node; and (4) calculating the LSI vector.

The time complexity of the four steps is as follows: (1) When performing lexical analysis and calculating the call relationship between classes, we need to traverse and read each Java file. The time complexity is $O(m \times s)$. (2) When merging the call relationships of class, the number of relationships is less than m^2, and we need to traverse all relationships, so the time complexity of this stage is less than $O(m^2)$. (3) When merging words of one node, we need to summarize the words of m files in total, and words in a Java file are roughly proportional to s, so the time complexity is approximately $O(m \times s)$. (4) When calculating the LSI vector of the node, the dimensions of the TF–IDF and the LSI model are fixed, so the time complexity is nearly $O(n)$. The total time complexity is less than $O(m \times s)+O(m^2)+O(m \times s)+O(n) = O(m \times s)+O(m^2)$.

The last part is graph feature extraction, permission feature fusion and classification. In the testing stage of the model, we need to input the CSCG and permission feature to obtain the classification result. Except for Max-GAT, the time consumption of the remaining parts can be considered as a constant, of which the time complexity is roughly O(1). Since the computational complexity if Max-GAT is only related to the number of nodes in CSCG, which is represented by n. The relationships are represented as a $n * n$ adjacency matrix, the time complexity of graph feature extraction is $O(n^2)$.

Summarizing the time complexity of the above three parts, the overall complexity of our method is less than $O(m \times log(m)) + O(m \times n) + O(m \times s) + O(m^2) + O(n^2)$, since $n \leq m$, and the overall time complexity is less than $O(m \times s) + O(m^2)$. Assume that the value of s is roughly constant, the overall time complexity is $O(m^2)$.

4. Experiments

We conduct extensive experiments to show the effectiveness of the detection method we proposed. We constructed three datasets and conducted nine groups of experiments, including comparison experiments and ablation studies, etc. We also analyze the results.

4.1. Datasets and indicators

Currently, there is no public full dataset for the Android malware dataset; therefore, we build our datasets based on the existing benign and malware datasets. We use three malware datasets and two benign datasets to construct three final datasets. The three malware datasets are as follows:

(1) **VirusShare** (Virusshare, 2020). This dataset includes malware information for major system platforms and has been produced since 2012. We downloaded the two most recent compressed packages in January 2021.

(2) **Drebin** (Arp et al., 2014). This dataset is based on Android applications involving 179 different malware families and was collected from August 2010 to October 2012.

(3) **AMD** (Wei, Li, Roy, Ou, & Zhou, 2017). The AMD (Android Malware Dataset) dataset contains 24,553 malware samples belonging to 135 variants of 71 different malware families. The data were collected from 2010 to 2016.

The two benign datasets are as follows:

(1) **Apkpure** (Apkpure Team, 2020). We downloaded the samples in the ".apk" format (excluding the ".xapk" format) for the top 25 pages from each of the 49 major categories from "apkpure.com" in January 2021, removed the samples that were larger than 100 MB and were verified by VirusTotal (Virus Total, 2012).

(2) **AndroZoo** (Allix, Bissyandé, Klein, & Le Traon, 2016). We downloaded the first 10,000 samples sorted by SHA256 from the AndroZoo platform in 2018 and verified them by VirusTotal (Virus Total, 2012).

We calculated the SHA256 value for all of the samples and confirmed that each sample in the two benign datasets was not found in any of the three malware datasets.

The three final datasets were constructed by combining the aforementioned five datasets based on the collection time, as shown in Table 1. Since Drebin and AMD were collected earlier, AndroZoo, which is closer with respect to time, was selected for combination with them. VirusShare was collected later so that Apkpure was selected to be combined with it. Obfuscation techniques have been widely used when

Table 1
Descriptions of the three final datasets.

Dataset	Number of samples	Avg num of classes	Avg ratio of 3rd party in original classes	Ratio of obfuscation samples
VirusShare_ Apkpure	Total: 19,266 Malware (from VirusShare): 10,000 Benign (from Apkpure): 9,266	4023	54%	78%
Drebin_ AndroZoo	Total: 15,106 Malware (from Drebin): 5,546 Benign (from AndroZoo): 9,560	933	82%	62%
AMD_ AndroZoo	Total: 19,560 Malware (from AMD): 10,000 Benign (from AndroZoo): 9,560	876	82%	65%

building APK files. To roughly evaluate the effects of our method on the obfuscated APK files, we calculated the proportion of the obfuscated APK files in the three datasets. According to He, Yang, Hu, and Wang (2019) and Derr, Bugiel, Fahl, Acar, and Backes (2017), we regarded classes named with one or two characters, or several of the same characters as obfuscated. Then, we set a threshold; when an APK file contains more than ten obfuscated classes, we marked it as obfuscated.

We split the datasets into two subdatasets, with 80% of the data used for training and 20% used for testing. We used accuracy (A) and the F-Measure (F1) as the indicators for the evaluation of the model.

4.2. Experimental settings

Our experimental environment was as follows: (1) CPU: AMD Ryzen ThreadRipper 3990X, (2) memory: 128 GB DDR4 with a frequency of 2400 MHz, (3) storage: INTEL 665P NVME 1 TB, (4) GPU: NVIDIA RTX 2080Ti, and (5) operating system: Ubuntu 18.04.

In the experiments, eight methods were selected for comparison, including three methods based on traditional machine learning models and five methods based on deep learning. Since the method developed in this paper is a static method, only static methods are considered for comparison. We selected three traditional machine learning methods: the Drebin (Arp et al., 2014) method is based on traditional feature engineering, the CSDB (Allix, Bissyandé, Jérome et al., 2016) method is based on call graphs, and the LSI+Permission method that we proposed in previous work (Song et al., 2019) is based on semantic features. For static methods based on deep learning, the following five methods that use different modality inputs were selected for comparison: DexRay (Daoudi et al., 2021) transforms APK files into images and uses these images as input, Deep Android Malware Detection (McLaughlin et al., 2017) (marked as DAMD) takes an opcode sequence as input, MSerNetDroid (Zhu, Gu et al., 2023) extracted three types of features to construct image as input, TC-Droid (Zhang et al., 2021) regards the features as texts and is a text classification model, and SAGEConv (Vinayaka & Jaidhar, 2021) uses call graph as input. Here, we considered MSerNetDroid, TC-Droid and SAGEConv to be state-of-the-art methods. Among these methods, Drebin, CSBD, DexRay, DAMD, and SAGEConv are tested using open-source codes. For DexRay, DAMD, and SAGEConv, we used original implementations published by the authors for experiments; for Drebin and CSBD, we used the reimplementations by other researchers (https://github.com/MLDroid/). For TC-Droid, we extracted four types of features to generate text data according to its description in the paper and classified them using the TextCNN model; for MSerNetDroid, we used source code provided by the authors.

Our experiments could be divided into five categories: the comparison experiments with existing methods, the ablation experiments, the generalization test experiment, the comparison experiment with antivirus scanners, and the efficiency comparison experiment.

(1) Comparison experiment with existing methods

We compared our method with the Drebin, CSBD, LSI+Permission, DexRay, DAMD, SAGEConv, TC-Droid, and MSerNetDroid methods.

(2) Ablation experiments

- **Effectiveness experiment for graph construction.** We compared the performance of the models based on different methods of CSCG construction to verify the effectiveness of the proposed approach.
- **Effectiveness experiment for graph feature extraction.** We compared the performance of the models using different graph feature extraction methods to verify the effectiveness of the proposed single-layer GAT.
- **Effectiveness experiment for graph scales.** We compared the performance of the model with different ts_max values.
- **Effectiveness experiment for feature fusion.** To assess the construction of the multimodal feature fusion network, we compared the results when permission features were and were not introduced and compared the performance of different fusion models. The results were used to verify the effectiveness of the proposed multipoint fusion network.
- **Effectiveness experiment for each single feature.** We test the performance of each single feature, which are the LSI feature, GAT network and permission, to show how the different parts affect the overall performance.

(3) Generalization testing experiment

We additionally downloaded 240 malware samples from VirusShare and 240 benign samples Apkpure on June 1, 2022, which have a time span of 17 months from the samples in our training set. We directly test the final model of the VirusShare_Apkpure dataset on the new small dataset to evaluate the generalization abilities.

(4) Comparison experiment with antivirus scanners

We additionally acquired 600 malware and 600 benign samples from CICMalDroid (Mahdavifar, Kadir, Fatemi, Alhadidi, & Ghorbani, 2020), and compared the detection performance of our model with four antivirus scanners on the new dataset.

(5) Efficiency comparison experiment

We compared the time cost of our method with eight comparison methods based on the test set of AMD_AndroZoo, to show the detection efficiency of different methods.

4.3. Results

4.3.1. Comparison experiment with existing methods

This experiment was conducted based on three datasets. The final parameters of our model are shown in Table 2. ts_min and the dropout rate are set to default values based on experience and are not adjusted. The optimal ts_max is selected for each dataset according to the results of experiment (3) in Section 4.3.2, and min_k is directly calculated based on ts_max and Formula (1). The parameters considered for adjusting are the learning rate and number of GAT attention heads. In the experiment, we consider two common values of 0.001 and 0.002 for the learning rate and try 2–8 for the number of attention heads, to select the optimal parameters. When training the model, we choose to use Adam as the optimizer, and set batch size as 10; we selected the model after 50 epoch iterations as the final model. To prevent our model from overfitting, we further split the training set into two parts, of which 90% of the samples continue to be used as the training set, and the other 10% are used as the validation set to observe the fitting status of the model. We ensured that after 50 epochs, the accuracy on the validation set did not decrease significantly, and we used the result on

Table 2
Best parameters of our model for each dataset.

Dataset	*ts_min*	*ts_max*	*min_k*	learning rate	dropout	Attention heads of GAT layer
VirusShare_Apkpure	100	1,500	4	0.001	0.2	2
Drebin_AndroZoo	100	1,200	1	0.002	0.2	4
AMD_AndroZoo	100	1,200	1	0.002	0.2	7

Table 3
Performance of comparison experiments.

Dataset	**VirusShare_Apkpure**		**Drebin_AndroZoo**		**AMD_AndroZoo**	
Indicator	**Accuracy**	**F1**	**Accuracy**	**F1**	**Accuracy**	**F1**
GAT + Permission (ours)	97.28%	97.29%	99.27%	99.01%	99.54%	99.55%
LSI + Permission (2019)	96.76%	96.76%	97.98%	97.23%	98.16%	98.22%
Drebin (2014)	97.15%	97.16%	97.95%	97.25%	97.37%	97.50%
CSBD (2016)	95.85%	95.91%	97.98%	97.20%	98.24%	98.26%
DexRay (2021)	92.22%	92.29%	94.21%	92.02%	88.19%	88.51%
DAMD (2017)	92.62%	92.72%	96.28%	94.93%	89.33%	89.22%
TC-Droid (2021)	95.43%	95.40%	97.94%	97.21%	97.24%	97.31%
SAGEConv (2021)	95.05%	95.09%	96.66%	95.40%	95.82%	95.89%
MSerNetDroid (2023)	96.16%	96.16%	98.00%	97.26%	98.44%	98.49%

the test set as the final result. However, according to our experimental results, when using different parameters, the difference in accuracy of each dataset is basically all within 0.5%. For each comparison method, we also tested multiple sets of parameters and selected the best result as the final result. The indicators on the three test sets for all methods are shown in Table 3.

In all tables of the experimental results, the best result is shown in red, and the second-best result is shown in blue. Table 3 shows that our method yields the best accuracy and F1 value for the three datasets. The accuracy of the Drebin method for the VirusShare_Apkpure dataset is only slightly lower than that of our method, but the Drebin method is significantly less accurate than our method for the other two datasets. Compared with the four deep learning methods, our method achieves the best detection effects; compared with MSerNetDroid, which achieves the best performance among the five methods, our method reaches a higher accuracy by 1.1–1.3% and a higher F1 value by 1.0–1.8%. In addition, the results show that the obfuscation techniques would not have a significant influence on our detection effects.

After analyzing the differences between the methods, we believe that the DexRay method directly convert the APK file into an image and perform classification; it only requires simple preprocessing but cannot achieve good detection effects. The DAMD, TC-Droid and MSerNetDroid methods extract sequence, text or binary features and then make classifications with deep learning models; thus, the detection effects are greatly limited by the original features they extracted. The Drebin method used eight types of features, the selected features are comprehensive and achieve good detection results on three datasets. The LSI + permission method we previously proposed used topic vectors to represent the semantic features from Android malware, which shows the effectiveness of the topic model. However, both methods extract global features of an Android application, which lacks a description of the local features and they also did not consider structural features. Our methods made up for this shortcoming by introducing the class-set call graph. The CSBD and SAGEConv methods are all based on graph classification. The CSBD method only used the structural features of the malware; although the SAGEConv methods introduced the node features of the graph; its features are basic statistical features. Compared with these two methods, we extracted more effective malware features by introducing LSI vectors as node semantic features in addition to the structural information.

4.3.2. Ablation experiments

(1) Experiment of graph construction

This experiment compared the performance of four graph construction methods based on three datasets. Here, *ts_max* was set to 1500 based on the computational capability. These four methods all use a single-layer GAT network for feature extraction and directly perform classification without the fusion of permission features. Additionally, the first three models are undirected graphs. The four methods are as follows.

(1) *min_k* is not set for all datasets; i.e., *min_k* is set to the default value of 1;

(2) *min_k* is set according to the scale of the APK files in each dataset. Due to the difference in the number of Java files in these datasets, *min_k* for each dataset cannot be set to the same value. The median numbers of Java files in the three datasets are as follows: approximately 383 for AMD_AndroZoo, approximately 371 for Drebin_AndroZoo, and approximately 2812 for VirusShare_Apkpure. The median numbers for AMD_AndroZoo and Drebin_AndroZoo are much lower than *ts_max* / 2, so *min_k* is set to 1, which is the same as the default value. The *min_k* value for VirusShare_Apkpure is set to 4 according to formula (1).

(3) Add the root node into the CSCG after setting *min_k*.

(4) Set *min_k*, and add the root node, but use a directed graph.

The experimental results are shown in Table 4.

Table 4 shows that the model that uses an undirected graph with the setting of *min_k* and addition of the root node achieved the best result. After setting *min_k* for VirusShare_Apkpure, the indicators improved slightly. Additionally, the average number of class-set nodes was reduced from 500 to 470, reducing resource consumption. We compared the results with and without the added root node. After adding the root node, the model performed better for all three datasets. This finding suggests that introducing the root node enhances model performance; notably, a relationship between the root node and the topmost node is formed, and a fully connected graph is generated. The results of the undirected graph and directed graph methods show that the undirected graph performs better than the directed graphs. Thus, for a node, both the nodes it calls and the nodes that call it should be considered in feature updating. Therefore, we used undirected graphs as the inputs of the GAT in the subsequent experiments.

(2) Experiment for graph feature extraction

This experiment compared the performance of the models using three graph convolutional networks that were all based on the best preprocessing method identified in experiment (1) in Section 4.3.2, which included an undirected graph, a dynamically set *min_k* value and the addition of a root node, and *ts_max* was set to 1500. The three models use the following different graph convolutional networks to extract graph features and directly perform classification without the

Table 4
Performance of different graph construction methods.

Dataset	VirusShare_Apkpure		Drebin_AndroZoo		AMD_AndroZoo	
Indicator	**Accuracy**	**F1**	**Accuracy**	**F1**	**Accuracy**	**F1**
Undirected	96.60%	96.63%	98.91%	98.47%	99.05%	99.05%
Undirected + *min_k*	96.63%	96.66%	98.91%	98.47%	99.05%	99.05%
Undirected + *min_k* + root node	96.81%	96.84%	98.97%	98.60%	99.16%	99.18%
Directed + *min_k* + root node	96.47%	96.49%	98.71%	98.24%	99.13%	99.15%

Table 5
Performance of different graph convolutional networks.

Dataset	VirusShare_Apkpure		Drebin_AndroZoo		AMD_AndroZoo	
Indicator	**Accuracy**	**F1**	**Accuracy**	**F1**	**Accuracy**	**F1**
Single-layer GAT	96.81%	96.84%	98.97%	98.60%	99.16%	99.18%
Two-layer GAT	96.65%	96.68%	98.84%	98.42%	99.06%	99.08%
GCN	96.39%	96.43%	98.51%	97.98%	98.80%	98.83%

Table 6
Performance of different *ts_max*.

Dataset	VirusShare_Apkpure		Drebin_AndroZoo		AMD_AndroZoo	
Indicator	**Accuracy**	**F1**	**Accuracy**	**F1**	**Accuracy**	**F1**
ts_max=600	96.68%	96.71%	99.01%	98.64%	99.29%	99.30%
ts_max=800	96.63%	96.65%	98.97%	98.60%	99.31%	99.33%
ts_max=1,000	96.68%	96.70%	98.91%	98.51%	99.24%	99.25%
ts_max=1,200	96.70%	96.73%	99.04%	98.69%	99.41%	99.43%
ts_max=1,500	96.81%	96.84%	98.97%	98.60%	99.26%	99.28%

fusion of permission features: (1) a single-layer GAT model, (2) a two-layer GAT model, and (3) a GCN model. The experimental results are shown in Table 5.

Table 5 shows that the single-layer GAT performed best. We compared the results of the single-layer GAT and the multilayer GAT. Notably, the results obtained with a single layer are better than those obtained after multilayer graph convolution operations, demonstrating that the first-order neighbors have a greater influence on the results than second-order neighbors. For the results of the single-layer GAT and GCN, the GAT model performs better than the GCN, mainly due to the introduction of attention mechanisms and the weights of neighboring nodes.

(3) Experiment for graph scales

This experiment compared the performance of five different values of *ts_max* to find the influence of different graph scales. We obtain the features of the CSCGs with different scales by using the single-layer GAT model and test the performance on the three datasets. The experimental results are shown in Table 6.

The experimental results show that the best *ts_max* varies with different datasets. The median numbers of Java files in the three datasets differ greatly, and are as follows: approximately 383 for AMD_AndroZoo, approximately 371 for Drebin_AndroZoo, and approximately 2812 for VirusShare_Apkpure. Therefore, the best *ts_max* for AMD_AndroZoo and Drebin_AndroZoo is smaller, while the best *ts_max* for VirusShare_Apkpure is larger. Due to the limitation of our server, the largest *ts_max* can be set to 1500. In the case where greater computational resources are available, we can try a larger *ts_max* for VirusShare_Apkpure. The experimental results also show that *ts_max* does not strongly influence the detection performance.

(4) Effectiveness experiment of feature fusion

This experiment compared the performance of three feature fusion networks. The *ts_max* for VirusShare_Apkpure is set to 1500, while the *ts_max* for Drebin_AndroZoo and AMD_AndroZoo is set to 1200. We use a single-layer GAT to extract features. The three methods are as follows:

(1) Only graph features are used for classification;

(2) A single-fusion-point network is used to fuse graphs and permission features, which is the same as a concatenation of the graph features and permission features. In Fig. 6, this means directly using the output of $FC1\_Fus$ as the input of FC_out;

(3) A multiple-fusion-point network is used to fuse graphs and permission features for classification.

The experimental results are shown in Table 7.

Table 7 shows that the multiple-fusion-point network achieves the best performance. The feature fusion method can effectively enhance model performance by introducing permission features. Additionally, the multiple-fusion-point network performs better than the single-fusion-point network for all three datasets, possibly implying that the use of multiple fusion points can achieve better fusion effects than the use of a single fusion point.

(5) Effectiveness experiment for a single feature

We test the performance of each single feature of our models, which are the LSI feature, GAT network, and permission, to show how the different parts affect the overall performance. Since node features are necessary in the GAT network, we test LSI+GAT. Therefore, we compared the performance of four types of features: only LSI features, only permission features, LSI + GAT features, and LSI + GAT + Permission features. The LSI features and permission features are directly classified with fully connected layers. Here, LSI + GAT features, and LSI + GAT + Permission features are equal to the settings of Experiment (4) in Section 4.3.2, which are only graph features and multiple fusion points. The experimental results are shown in Table 8.

From the experimental results, all three parts of our method have impacts on the performance of the model. Comparing the methods with only the LSI and LSI+GAT methods, after introducing the GAT model, the overall accuracy of the model is improved by 2.3%–3.8%, which proves that our GAT method has a significant impact on the model effect. In addition, the method of permission feature only has a low detection effect, but the introduction of permission feature can further improve the effect of the model.

Table 7
Performance of different feature fusion networks.

Dataset	VirusShare_Apkpure		Drebin_AndroZoo		AMD_AndroZoo	
Indicator	**Accuracy**	**F1**	**Accuracy**	**F1**	**Accuracy**	**F1**
Only graph features	96.81%	96.84%	99.04%	98.69%	99.41%	99.43%
Single fusion point	96.89%	96.91%	99.20%	98.92%	99.44%	99.43%
multiple fusion points	97.28%	97.29%	99.27%	99.01%	99.54%	99.55%

Table 8
Performance of each single feature and fusion features.

Dataset	VirusShare_Apkpure		Drebin_AndroZoo		AMD_AndroZoo	
Indicator	**Accuracy**	**F1**	**Accuracy**	**F1**	**Accuracy**	**F1**
Only LSI	94.47%	94.49%	96.89%	95.71%	95.56%	95.55%
Only Permission	91.80%	91.77%	93.64%	91.13%	92.02%	92.25%
LSI + GAT	96.81%	96.84%	99.04%	98.69%	99.41%	99.43%
LSI+GAT+permission	97.28%	97.29%	99.27%	99.01%	99.54%	99.55%

Table 9
Generalization performance of our model.

VirusShare_Apkpure Test Set		VirusShare_Apkpure Generalization Set	
Accuracy	**F1**	**Accuracy**	**F1**
97.28%	97.29%	96.67%	96.65%

4.3.3. Generalization testing experiment

Since different malware families vary greatly, it is difficult to achieve good performance for all malware families. Malware families evolve from time to time, and various varieties update rapidly. Therefore, similar to the researches in Pei et al. (2020), Yerima and Khan (2019), we use a new generalization of the malware families in our test datasets after some time period for this experiment. For the three malware datasets we used, only VirusShare is still updated; we choose VirusShare_Apkpure to conduct the test. We additionally downloaded the 240 most recent malware samples from VirusShare and 240 benign samples from Apkpure on June 1, 2022, which have a time span of 17 months with samples from the training set. The experimental results of our model are shown in Table 9.

Although the result is slightly decreased on the dataset from the same sources with a time span of 17 months, we can still ensure that our model has high accuracy and F1 score. It is shown that our model has certain generalization abilities.

4.3.4. Comparison experiment with antivirus scanners

We acquired android application samples from CICMalDroid (Mahdavifar et al., 2020), and compare our method with the four antivirus scanners. The samples of CICMalDroid were collected from December 2017 to December 2018, and contained benign samples and four malware categories: Adware, Banking, SMS, and Riskware. We randomly selected 600 benign samples, and 150 malware samples from each category to construct a testing dataset. We choose to use the final model trained on the AMD_AndroZoo dataset to perform detection, and we ensure that the samples we selected does not contain malwares in the AMD dataset which we used for model training. We choose four antivirus scanners for comparison: Kaspersky (AO Kaspersky Lab, 2023), Avast (Avast Software, 2023), 360 Antivirus (360 CN, 2023), and Huorong Internet Security (Beijing Huorong Network Technology, 2023). The final results of the comparison are shown in Table 10.

From the results, we found that our method achieved a recall rate of more than 83% for the four types of Android malwares with no false positives on the dataset. As to the antivirus scanners, the performance of Kaspersky is closest to our method. The results prove that, the model we trained with approximately 15,000 samples could achieve better detection effects than antivirus scanners for the four types of Android malwares. Since the samples were acquired approximately five years ago, and antivirus scanners may have generated proper signatures for such samples, they may achieve a gradually increased detection effect over time.

Table 10
Performance of our method and antivirus scanners.

Indicators	Accuracy	Precision	Recall	F1
Ours	91.58%	100.00%	83.17%	90.81%
Kaspersky	91.41%	100.00%	82.83%	90.61%
Avast	85.08%	100.00%	70.17%	82.47%
360	78.75%	100.00%	72.50%	84.06%
Huorong	77.67%	99.11%	55.80%	71.40%

Table 11
Time cost comparison results.

Method	Avg time/s (Average time per sample)
GAT + Permission (ours)	1.1305
LSI+Permission (2019)	0.7993
Drebin (2014)	0.3216
CSBD (2016)	0.3536
DexRay (2021)	0.0276
DAMD (2017)	0.2475
TC-Droid (2021)	0.4367
SAGEConv (2021)	0.2575
MSerNetDroid (2023)	0.1785

4.3.5. Efficiency comparison experiment

We compare the time cost of our method with eight comparison methods based on the test set of AMD_AndroZoo. Each method runs on the server described in Section 4.2, and each method runs in parallel as much as possible to improve the detection efficiency. Table 11 shows the time cost of each method, and Table 12 shows the time cost of each stage of our method.

In Table 11, it is shown that our method requires longer time to some extent, since we conduct more operations than other methods to achieve better detection effects. Among the comparison methods, the DexRay method only needs to read the byte stream and convert it into images for detection; it achieved best detection efficiency, but could not achieve good detection effects. The other seven methods need to perform preprocessing, which results in more time costs.

Table 12 shows that, in our method, the preprocessing stage consumes most of the detection time, and JADX consumes the most. Since our method contains three preprocessing steps, and the comparison methods usually use only one tool for preprocessing, the time cost is relatively shorter than ours. We think that our method could still meet

Table 12
Time cost of each stage in our method.

Stages	Processing	Avg time/s	Total time/s
Preprocessing	Literadar	0.2319	0.7686
	Apktool	0.1540	
	JADX	0.3827	
Global LSI and Permission feature extraction	Lexical analyzer	0.0156	0.0305
	Global LSI extraction	0.0148	
	Permission extraction	0.0001	
CSCG construction	Topology construction	0.2366	0.2659
	Node LSI feature calculation	0.0293	
GAT testing	Loading data	0.0403	0.0654
	Testing	0.0252	
Total	Total	1.1305	1.1305

practical real-time performance demands, and the prepossessing time overhead may be reduced by more parallelization when running on advanced servers with rich computing resources.

5. Conclusions

In this paper, an Android malware detection method based on the deep fusion of multimodal features is proposed. This method constructs a new type of call graph named the class-set call graph. The LSI features of the class-set codes are used as node features, and the salient region features are extracted with a graph attention network. Therefore, this method effectively integrates the semantic features and structural features of Android applications. To enhance the semantic information of each node, a dynamic adaptive node merging method is proposed. This approach can also reduce the influence of the third-party libraries and reduce the scale of the call graph while preserving the original information as much as possible. Finally, a multiple feature fusion point network is established; it fuses graph features and permission features to perform binary classification for Android applications. We conducted experiments based on multiple datasets, and the results indicate that our method achieves an accuracy of 97.28–99.54%, which is higher than that of the state-of-the-art methods.

In our future work, we will further study the preprocessing parallelization approach to improve the detection efficiency of our method, and consider some detection schemes to deal with obfuscation techniques. Additionally, we will pay more attention to specific types of malwares, such as hot loading module detection and phishing detection.

CRediT authorship contribution statement

Shaojie Chen: Methodology, Software, Validation, Writing – original draft, Editing. **Bo Lang:** Conceptualization, Supervision, Writing – review & editing. **Hongyu Liu:** Investigation, Writing – review & editing, Data curation. **Yikai Chen:** Investigation, Writing – review & editing. **Yucai Song:** Data curation, Software, Validation.

Declaration of competing interest

The authors declare that no conflict of interest exists in the submission of this manuscript, and manuscript is approved by all authors for publication. And the work described was original research that has not been published previously, and not under consideration for publication elsewhere, in whole or in part. All the authors listed have approved the manuscript that is enclosed.

Data availability

Data will be made available on request.

Acknowledgments

This work was funded by State Key Laboratory of Software Development Environment [grant number SKLSDE-2020ZX-02]. All authors read and approved the final manuscript.

References

360 CN (2023). 360 Antivirus. Retrieved from https://sd.360.cn/index.html, Accessed March 4, 2023.

360 Security Center (2022). 2021 Mobile security report. Retrieved from https://pop.shouji.360.cn/safe_report/Mobile-Security-Report-202112.pdf, Accessed June 20, 2022.

Aafer, Y., Du, W., & Yin, H. (2013). Droidapiminer: Mining api-level features for robust malware detection in android. In *International conference on security and privacy in communication systems* (pp. 86–103). Springer, http://dx.doi.org/10.1007/978-3-319-04283-1_6.

Alazab, M., Alazab, M., Shalaginov, A., Mesleh, A., & Awajan, A. (2020). Intelligent mobile malware detection using permission requests and api calls. *Future Generation Computer Systems*, *107*, 509–521. http://dx.doi.org/10.1016/j.future.2020.02.002.

Allix, K., Bissyandé, T. F., Jérome, Q., Klein, J., Le Traon, Y., et al. (2016). Empirical assessment of machine learning-based malware detectors for android. *Empirical Software Engineering*, *21*(1), 183–211. http://dx.doi.org/10.1007/s10664-014-9352-6.

Allix, K., Bissyandé, T. F., Klein, J., & Le Traon, Y. (2016). Androzoo: Collecting millions of android apps for the research community. In *2016 IEEE/ACM 13th working conference on mining software repositories* (pp. 468–471). IEEE, http://dx.doi.org/10.1145/2901739.2903508.

Alzaylaee, M. K., Yerima, S. Y., & Sezer, S. (2020). DL-droid: Deep learning based android malware detection using real devices. *Computers & Security*, *89*, Article 101663. http://dx.doi.org/10.1016/j.cose.2019.101663.

Amin, M., Tanveer, T. A., Tehseen, M., Khan, M., Khan, F. A., & Anwar, S. (2020). Static malware detection and attribution in android byte-code through an end-to-end deep system. *Future Generation Computer Systems*, *102*, 112–126. http://dx.doi.org/10.1016/j.future.2019.07.070.

AO Kaspersky Lab (2023). Kaspersky cyber security solutions. Retrieved from https://www.kaspersky.com/, Accessed March 4, 2023.

Apkpure Team (2020). APKPure. Retrieved from https://apkpure.com/, Accessed January 1, 2021.

Arnold, K., Gosling, J., & Holmes, D. (2005). *The java programming language*. Addison Wesley Professional.

Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., & Siemens, C. (2014). Drebin: Effective and explainable detection of android malware in your pocket.. In *Ndss, Vol. 14* (pp. 23–26).

Arshad, S., Shah, M. A., Wahid, A., Mehmood, A., Song, H., & Yu, H. (2018). Samadroid: A novel 3-level hybrid malware detection model for android operating system. *IEEE Access*, *6*, 4321–4339. http://dx.doi.org/10.1109/ACCESS.2018.2792941.

Avast Software (2023). Avast antivirus software. Retrieved from https://www.avast.com/, Accessed March 4, 2023.

Beebom Staff (2021). Google to replace APK with android app bundles (AAB) starting august 2021. Retrieved from https://beebom.com/google-replaces-apk-with-android-app-bundles-aab/, Accessed July 21, 2021.

Beijing Huorong Network Technology (2023). Huorong internet security. Retrieved from https://www.huorong.cn/, Accessed March 4, 2023.

Burguera, I., Zurutuza, U., & Nadjm-Tehrani, S. (2011). Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on security and privacy in smartphones and mobile devices* (pp. 15–26). http://dx.doi.org/10.1145/2046614.2046619.

Cai, L., Li, Y., & Xiong, Z. (2021). Jowmdroid: Android malware detection based on feature weighting with joint optimization of weight-mapping and classifier parameters. *Computers & Security*, *100*, Article 102086. http://dx.doi.org/10.1016/j.cose.2020.102086.

Chen, T., Mao, Q., Yang, Y., Lv, M., & Zhu, J. (2018). TinyDroid: A lightweight and efficient model for android malware detection and classification. *Mobile Information Systems*, *2018*, http://dx.doi.org/10.1155/2018/4157156.

Chen, S., Xue, M., Tang, Z., Xu, L., & Zhu, H. (2016). Stormdroid: A streaminglized machine learning-based system for detecting android malware. In *Proceedings of the 11th ACM on Asia conference on computer and communications security* (pp. 377–388). http://dx.doi.org/10.1145/2897845.2897860.

Daoudi, N., Samhi, J., Kabore, A. K., Allix, K., Bissyandé, T. F., & Klein, J. (2021). Dexray: A simple, yet effective deep learning approach to android malware detection based on image representation of bytecode. In *International workshop on deployable machine learning for security defense* (pp. 81–106). Springer, http://dx.doi.org/10.1007/978-3-030-87839-9_4.

Derr, E., Bugiel, S., Fahl, S., Acar, Y., & Backes, M. (2017). Keep me updated: An empirical study of third-party library updatability on android. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security* (pp. 2187–2200). http://dx.doi.org/10.1145/3133956.3134059.

Feizollah, A., Anuar, N. B., Salleh, R., Suarez-Tangil, G., & Furnell, S. (2017). Androdialysis: Analysis of android intent effectiveness in malware detection. *Computers & Security*, *65*, 121–134. http://dx.doi.org/10.1016/j.cose.2016.11.007.

Gao, H., Cheng, S., & Zhang, W. (2021). Gdroid: Android malware detection and classification with graph convolutional network. *Computers & Security*, *106*, Article 102264. http://dx.doi.org/10.1016/j.cose.2021.102264.

Gu, Z., Lang, B., Yue, T., & Huang, L. (2017). Learning joint multimodal representation based on multi-fusion deep neural networks. In *International conference on neural information processing* (pp. 276–285). Springer, http://dx.doi.org/10.1007/978-3-319-70096-0_29.

Han, Q., Subrahmanian, V., & Xiong, Y. (2020). Android malware detection via (somewhat) robust irreversible feature transformations. *IEEE Transactions on Information Forensics and Security*, *15*, 3511–3525. http://dx.doi.org/10.1109/TIFS.2020.2975932.

He, Y., Yang, X., Hu, B., & Wang, W. (2019). Dynamic privacy leakage analysis of android third-party libraries. *Journal of Information Security and Applications*, *46*, 259–270. http://dx.doi.org/10.1016/j.jisa.2019.03.014.

Hei, Y., Yang, R., Peng, H., Wang, L., Xu, X., Liu, J., et al. (2021). Hawk: Rapid android malware detection through heterogeneous graph attention networks. *IEEE Transactions on Neural Networks and Learning Systems*, http://dx.doi.org/10.1109/TNNLS.2021.3105617.

Hsien-De Huang, T., & Kao, H.-Y. (2018). R2-d2: Color-inspired convolutional neural network (cnn)-based android malware detections. In *2018 IEEE international conference on big data* (pp. 2633–2642). IEEE, http://dx.doi.org/10.1109/BigData.2018.8622324.

Idrees, F., Rajarajan, M., Conti, M., Chen, T. M., & Rahulamathavan, Y. (2017). Pindroid: A novel android malware detection system using ensemble learning methods. *Computers & Security*, *68*, 36–46. http://dx.doi.org/10.1016/j.cose.2017.03.011.

Jerbi, M., Dagdia, Z. C., Bechikh, S., & Said, L. B. (2020). On the use of artificial malicious patterns for android malware detection. *Computers & Security*, *92*, Article 101743. http://dx.doi.org/10.1016/j.cose.2020.101743.

John, T. S., Thomas, T., & Emmanuel, S. (2020). Graph convolutional networks for android malware detection with system call graphs. In *2020 Third ISEA conference on security and privacy* (pp. 162–170). IEEE, http://dx.doi.org/10.1109/ISEA-ISAP49340.2020.235015.

Karbab, E. B., Debbabi, M., Derhab, A., & Mouheb, D. (2020). Scalable and robust unsupervised android malware fingerprinting using community-based network partitioning. *Computers & Security*, *97*, Article 101965. http://dx.doi.org/10.1016/j.cose.2020.101965.

Kim, T., Kang, B., Rho, M., Sezer, S., & Im, E. G. (2018). A multimodal deep learning method for android malware detection using various features. *IEEE Transactions on Information Forensics and Security*, *14*(3), 773–788. http://dx.doi.org/10.1109/TIFS.2018.2866319.

Kipf, T. N., & Welling, M. (2016). Semi-supervised classification with graph convolutional networks. http://dx.doi.org/10.48550/arXiv.1609.02907, arXiv preprint arXiv:1609.02907.

Knyazev, B., Lin, X., Amer, M. R., & Taylor, G. W. (2018). Spectral multigraph networks for discovering and fusing relationships in molecules. http://dx.doi.org/10.48550/arXiv.1811.09595, arXiv preprint arXiv:1811.09595.

Kouliaridis, V., Potha, N., & Kambourakis, G. (2020). Improving android malware detection through dimensionality reduction techniques. In *MLN* (pp. 57–72). http://dx.doi.org/10.1007/978-3-030-70866-5_4.

Li, M., Wang, W., Wang, P., Wang, S., Wu, D., Liu, J., et al. (2017). Libd: Scalable and precise third-party library detection in android markets. In *2017 IEEE/ACM 39th international conference on software engineering* (pp. 335–346). IEEE, http://dx.doi.org/10.1109/ICSE.2017.38.

Ma, Z., Wang, H., Guo, Y., & Chen, X. (2016). Libradar: fast and accurate detection of third-party libraries in android apps. In *Proceedings of the 38th international conference on software engineering companion* (pp. 653–656). http://dx.doi.org/10.1145/2889160.2889178.

Mahdavifar, S., Kadir, A. F. A., Fatemi, R., Alhadidi, D., & Ghorbani, A. A. (2020). Dynamic android malware category classification using semi-supervised deep learning. In *2020 IEEE intl conf on dependable, autonomic and secure computing, intl conf on pervasive intelligence and computing, intl conf on cloud and big data computing, intl conf on cyber science and technology congress* (pp. 515–522). IEEE, http://dx.doi.org/10.1109/DASC-PICom-CBDCom-CyberSciTech49142.2020.00094.

McLaughlin, N., Martinez del Rincon, J., Kang, B., Yerima, S., Miller, P., Sezer, S., et al. (2017). Deep android malware detection. In *Proceedings of the seventh ACM on conference on data and application security and privacy* (pp. 301–308). http://dx.doi.org/10.1145/3029806.3029823.

Milosevic, N., Dehghantanha, A., & Choo, K.-K. R. (2017). Machine learning aided android malware classification. *Computers & Electrical Engineering*, *61*, 266–274. http://dx.doi.org/10.1016/j.compeleceng.2017.02.013.

Odusami, M., Abayomi-Alli, O., Misra, S., Shobayo, O., Damasevicius, R., & Maskeliunas, R. (2018). Android malware detection: A survey. In *International conference on applied informatics* (pp. 255–266). Springer, http://dx.doi.org/10.1007/978-3-030-01535-0_19.

Pei, X., Yu, L., & Tian, S. (2020). AMalNet: A deep learning framework based on graph convolutional networks for malware detection. *Computers & Security*, *93*, Article 101792. http://dx.doi.org/10.1016/j.cose.2020.101792.

Pektaş, A., & Acarman, T. (2020). Deep learning for effective android malware detection using API call graph embeddings. *Soft Computing*, *24*(2), 1027–1043. http://dx.doi.org/10.1007/s00500-019-03940-5.

Qiu, J., Zhang, J., Luo, W., Pan, L., Nepal, S., & Xiang, Y. (2020). A survey of android malware detection with deep neural models. *ACM Computing Surveys*, *53*(6), 1–36. http://dx.doi.org/10.1145/3417978.

Ren, Z., Wu, H., Ning, Q., Hussain, I., & Chen, B. (2020). End-to-end malware detection for android IoT devices using deep learning. *Ad Hoc Networks*, *101*, Article 102098. http://dx.doi.org/10.1016/j.adhoc.2020.102098.

Şahin, D. O., Kural, O. E., Akleylek, S., & Kılıç, E. (2021). A novel permission-based android malware detection system using feature selection based on linear regression. *Neural Computing and Applications*, 1–16. http://dx.doi.org/10.1007/s00521-021-05875-1.

Seraj, S., Khodambashi, S., Pavlidis, M., & Polatidis, N. (2022). HamDroid: permission-based harmful android anti-malware detection using neural networks. *Neural Computing and Applications*, *34*(18), 15165–15174. http://dx.doi.org/10.1007/s00521-021-06755-4.

Shen, F., Del Vecchio, J., Mohaisen, A., Ko, S. Y., & Ziarek, L. (2018). Android malware detection using complex-flows. *IEEE Transactions on Mobile Computing*, *18*(6), 1231–1245. http://dx.doi.org/10.1109/TMC.2018.2861405.

Skylot (2014). JADX. Retrieved from https://github.com/skylot/jadx, Accessed August 1, 2020.

Song, Y., Chen, Y., Lang, B., Liu, H., & Chen, S. (2019). Topic model based android malware detection. In *International conference on security, privacy and anonymity in computation, communication and storage* (pp. 384–396). Springer, http://dx.doi.org/10.1007/978-3-030-24907-6_29.

Surendran, R., Thomas, T., & Emmanuel, S. (2020). Gsdroid: Graph signal based compact feature representation for android malware detection. *Expert Systems with Applications*, *159*, Article 113581. http://dx.doi.org/10.1016/j.eswa.2020.113581.

Taheri, R., Ghahramani, M., Javidan, R., Shojafar, M., Pooranian, Z., & Conti, M. (2020). Similarity-based android malware detection using hamming distance of static binary features. *Future Generation Computer Systems*, *105*, 230–247. http://dx.doi.org/10.1016/j.future.2019.11.034.

Talha, K. A., Alper, D. I., & Aydin, C. (2015). APK auditor: Permission-based android malware detection system. *Digital Investigation*, *13*, 1–14. http://dx.doi.org/10.1016/j.diin.2015.01.001.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., & Bengio, Y. (2017). Graph attention networks. http://dx.doi.org/10.48550/arXiv.1710.10903, arXiv preprint arXiv:1710.10903.

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

Virus Total (2012). Virustotal-free online virus, malware and howpublished scanner. Retrieved from https://www.virustotal.com, Accessed January 10, 2021.

Virusshare (2020). VirusShare.com - because sharing is caring. Retrieved from https://virusshare.com/, Accessed January 1, 2021.

Wang, S., Chen, Z., Yan, Q., Yang, B., Peng, L., & Jia, Z. (2019). A mobile malware detection method using behavior features in network traffic. *Journal of Network and Computer Applications*, *133*, 15–25. http://dx.doi.org/10.1016/j.jnca.2018.12.014.

Wang, X., & Li, C. (2021). Android malware detection through machine learning on kernel task structures. *Neurocomputing*, *435*, 126–150. http://dx.doi.org/10.1016/j.neucom.2020.12.088.

Wang, S., Yan, Q., Chen, Z., Yang, B., Zhao, C., & Conti, M. (2017). Detecting android malware leveraging text semantics of network flows. *IEEE Transactions on Information Forensics and Security*, *13*(5), 1096–1109. http://dx.doi.org/10.1109/TIFS.2017.2771228.

Wang, W., Zhao, M., & Wang, J. (2019). Effective android malware detection with a hybrid model based on deep autoencoder and convolutional neural network. *Journal of Ambient Intelligence and Humanized Computing*, *10*(8), 3035–3043. http://dx.doi.org/10.1007/s12652-018-0803-6.

Wei, F., Li, Y., Roy, S., Ou, X., & Zhou, W. (2017). Deep ground truth analysis of current android malware. In *International conference on detection of intrusions and malware, and vulnerability assessment* (pp. 252–276). Springer, http://dx.doi.org/10.1007/978-3-319-60876-1_12.

Winsniewski, R. (2012). Android–apktool: A tool for reverse engineering android apk files. *Retrieved February*, *10*, 2020.

Wu, S., Wang, P., Li, X., & Zhang, Y. (2016). Effective detection of android malware based on the usage of data flow APIs and machine learning. *Information and Software Technology*, *75*, 17–25. http://dx.doi.org/10.1016/j.infsof.2016.03.004.

Xiao, X., Zhang, S., Mercaldo, F., Hu, G., & Sangaiah, A. K. (2019). Android malware detection based on system call sequences and LSTM. *Multimedia Tools and Applications*, *78*(4), 3979–3999. http://dx.doi.org/10.1007/s11042-017-5104-0.

Xu, K., Li, Y., & Deng, R. H. (2016). Iccdetector: Icc-based malware detection on android. *IEEE Transactions on Information Forensics and Security*, *11*(6), 1252–1264. http://dx.doi.org/10.1109/TIFS.2016.2523912.

Xu, X., Liu, C., Feng, Q., Yin, H., Song, L., & Song, D. (2017). Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security* (pp. 363–376). http://dx.doi.org/10.1145/3133956.3134018.

Xu, Z., Ren, K., Qin, S., & Craciun, F. (2018). Cdgdroid: Android malware detection based on deep learning using CFG and DFG. In *International conference on formal engineering methods* (pp. 177–193). Springer, http://dx.doi.org/10.1007/978-3-030-02450-5_11.

Xu, Z., Ren, K., & Song, F. (2019). Android malware family classification and characterization using CFG and DFG. In *2019 International symposium on theoretical aspects of software engineering* (pp. 49–56). IEEE, http://dx.doi.org/10.1109/TASE.2019.00-20.

Yadav, P., Menon, N., Ravi, V., Vishvanathan, S., & Pham, T. D. (2022). EfficientNet convolutional neural networks-based android malware detection. *Computers & Security*, *115*, Article 102622. http://dx.doi.org/10.1016/j.cose.2022.102622.

Yerima, S. Y., & Khan, S. (2019). Longitudinal performance analysis of machine learning based android malware detectors. In *2019 International conference on cyber security and protection of digital services* (pp. 1–8). IEEE, http://dx.doi.org/10.1109/CyberSecPODS.2019.8885384.

Yerima, S. Y., Sezer, S., & Muttik, I. (2015). High accuracy android malware detection using ensemble learning. *IET Information Security*, *9*(6), 313–320. http://dx.doi.org/10.1049/iet-ifs.2014.0099.

Yuan, H., Tang, Y., Sun, W., & Liu, L. (2020). A detection method for android application security based on TF-IDF and machine learning. *Plos one*, *15*(9), Article e0238694. http://dx.doi.org/10.1371/journal.pone.0238694.

Yuan, B., Wang, J., Liu, D., Guo, W., Wu, P., & Bao, X. (2020). Byte-level malware classification based on markov images and deep learning. *Computers & Security*, *92*, Article 101740. http://dx.doi.org/10.1016/j.cose.2020.101740.

Zhan, X., Fan, L., Liu, T., Chen, S., Li, L., Wang, H., et al. (2020). Automated third-party library detection for android applications: Are we there yet? In *2020 35th IEEE/ACM international conference on automated software engineering* (pp. 919–930). IEEE, http://dx.doi.org/10.1145/3324884.3416582.

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

Zhou, Q., Feng, F., Shen, Z., Zhou, R., Hsieh, M.-Y., & Li, K.-C. (2019). A novel approach for mobile malware classification and detection in android systems. *Multimedia Tools and Applications*, *78*(3), 3529–3552. http://dx.doi.org/10.1007/s11042-018-6498-z.

Zhou, Y., Wang, Z., Zhou, W., & Jiang, X. (2012). Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. In *NDSS, Vol. 25* (4), (pp. 50–52).

Zhu, H.-j., Gu, W., Wang, L.-m., Xu, Z.-c., & Sheng, V. S. (2023). Android malware detection based on multi-head squeeze-and-excitation residual network. *Expert Systems with Applications*, *212*, Article 118705. http://dx.doi.org/10.1016/j.eswa.2022.118705.

Zhu, H., Wei, H., Wang, L., Xu, Z., & Sheng, V. S. (2023). An effective end-to-end android malware detection method. *Expert Systems with Applications*, *218*, Article 119593. http://dx.doi.org/10.1016/j.eswa.2023.119593.