Computers & Security 140 (2024) 103807

Contents lists available at ScienceDirect

Computers & Security

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

GSEDroid: GNN-based Android malware detection framework using lightweight semantic embedding

Jintao Gu, Hongliang Zhu *, Zewei Han, Xiangyu Li, Jianjin Zhao

School of Cyberspace Security, Beijing University of Posts and Telecommunications, Beijing, China

ARTICLE INFO

Keywords:
Android malware detection
Graph neural network
API call graph
Dalvik opcode embedding
CodeBERT
TextCNN

ABSTRACT

Currently, the prevalence of Android malware remains substantial. Malicious programs increasingly use advanced obfuscation techniques, posing challenges for security professionals with enhanced disguises, a proliferation of variants, and escalating detection difficulty. Leveraging semantic features presents a promising avenue to address these challenges. Rich semantic information encapsulated within opcodes and API call graphs has been identified as crucial in distinguishing benign from malicious applications. Consequently, various Natural Language Processing (NLP) technologies, such as Word2vec, are employed to encode features of Dalvik opcode sequences, thereby yielding embedded representations. Given that malware developers often opt for semantically similar APIs to achieve comparable functionalities, it is posited that the opcode embeddings for such APIs should exhibit similar characteristics. However, simple NLP models that only extract statistical information are insufficient for understanding obfuscated malware's behavioral patterns, as they do not provide comprehensive semantic insights. To bridge this gap, we propose a novel, lightweight embedding model based on CodeBERT and TextCNN. This model aims for efficient and precise representation of opcode sequences. Consequently, we introduce GSEDroid, an Android malware detection framework that uses an API call graph with permission and opcode semantic features to characterize APKs. This approach converts the detection challenge into a graph classification task executed via a graph neural network algorithm. The efficacy of our method has been validated through comparative analyses with other techniques. Experimental results demonstrate that our GraphSage+SAGPooling model achieved an accuracy of 99.47% and an F1-score of 99.44%, underscoring its effectiveness in Android malware detection.

1. Introduction

With the global proliferation of Android devices, the expansive user base presents a substantial attack surface. According to Statcounter's 2023 report on Operating System Market Share Worldwide (GlobalStats, 2023), as of February 2023, the Android operating system holds a dominant 43.88% share in the global market. Although the Android OS's openness and customizability provide users with more choices and flexibility, they also facilitate the spread and installation of malicious software. Android malware attackers leverage various strategies for monetization, including theft of sensitive information, ad display, and ransom demands. Current cybercrime trends show a shift in focus towards the quality of Android malware rather than its quantity.

The G DATA Mobile Security Report (CyberDefence General Dynamics, 2023), indicates a continued downward trend in the first half of 2022, with attackers releasing two to three malware-infected applications per minute, down from the previous average of five. Although the frequency of attacks has decreased, their quality has noticeably increased. The 2020 Overview of Internet Network Security Situation in China by CNCERT (Internet Emergency Center of China, 2020) reveals that around 3.03 million new mobile internet malicious programs were identified in the year, an 8.5% increase from the previous year. However, from 2014 to 2020, there was a consistent annual decrease in the number of malicious apps removed. This situation underscores the urgent need to control the spread of malicious Android applications at their source, effectively cutting off propagation channels and improving malware detection accuracy. These challenges increase the demands on Android malware detection technologies, necessitating more sophisticated and effective methodologies to combat the evolving cyber threat landscape.

* Corresponding author.
E-mail addresses: 2022010334@bupt.cn (J. Gu), zhuhongliang@bupt.edu.cn (H. Zhu), hanzewei@bupt.edu.cn (Z. Han), xiangyuli@bupt.edu.cn (X. Li), zhaojianjin@bupt.edu.cn (J. Zhao).

https://doi.org/10.1016/j.cose.2024.103807
Received 27 November 2023; Received in revised form 1 February 2024; Accepted 6 March 2024

Available online 12 March 2024

Current research in Android malware detection extensively employs data mining and machine learning methodologies to develop effective detection mechanisms. These approaches can be broadly categorized into two types. The first category utilizes a range of machine learning algorithms, such as Support Vector Machines (SVM), Random Forests, and XGBoost. This approach involves extracting code, resources, configuration, and other essential elements from Android application packages (APKs) to form feature vectors. These vectors are then processed using classifiers constructed with machine learning algorithms to differentiate between benign and malicious software. However, traditional machine learning algorithms are typically more suited for resolving relatively straightforward problems and may not perform as effectively in handling large-scale data and complex patterns (LeCun et al., 2015; Zhao et al., 2021b). The second category marks the application of deep learning algorithms in the realm of Android malware detection. The adoption of deep learning techniques to address specific scenarios has emerged as a predominant trend, with Graph Neural Networks (GNNs) becoming a focal area of research within the domain of deep learning.

An increasing number of researchers are adopting techniques from diverse domains to develop Android malware detection methods. Innovations include methods derived from the analysis of images (Yadav et al., 2022), audio (Tarwireyi et al., 2023), network traffic (Wang et al., 2018), and logs (Meng et al., 2021). Earlier studies predominantly focused on syntactic features, such as requested permissions (Arora et al., 2020), intent behaviors, inter-component communications (ICCs) (Cai et al., 2019), and system calls (Bhat et al., 2023). Recently, the rapid advancement in graph deep learning technologies has garnered significant attention. These technologies are adept at learning high-quality graph representations, thereby efficiently accomplishing various tasks on graphs. Consequently, the semantic features embedded in graph data, such as Function Call Graphs (FCGs), have become increasingly pivotal. Researchers are now constructing graphs to analyze and detect malicious software. For instance, GDroid (Gao et al., 2021) maps APKs and APIs onto a large heterogeneous graph based on call relationships and API usage patterns, transforming the original problem into a node classification task. This approach introduces a novel method for Android malware detection and family classification using Graph Convolutional Networks (GCNs). Additionally, PermPair (Arora et al., 2020) proposes a novel methodology to detect malicious applications by constructing graphs using permission pairs extracted from manifest files.

Previous studies (Khan et al., 2021; Pektaş and Acarman, 2020; Zhang et al., 2018) have indicated that opcodes in executable files harbor potential information. Given their ability to intricately represent behavioral patterns of programs, Dalvik opcodes are considered among the most prevalent static features for distinguishing between benign and malicious Android software. However, there lacks a standardized method for processing these features. Manually crafted feature extraction processes are notably costly in terms of resources and time. To automate this process and effectively identify latent information and extract deep features, it is imperative to construct an appropriate embedding model that efficaciously represents opcode sequences. The similarity of opcode sequences to textual data and their close resemblance to English words make them suitable candidates for processing with NLP techniques. In recent years, several studies have proposed the use of N-gram, Word2vec, and other such methodologies for handling opcode sequences. Nonetheless, these approaches are not without their limitations. There is a lack of comparative evaluation and rational explanation regarding the effectiveness of these NLP models in embedding representations of opcode sequences. This assessment is vital for determining the applicability and effectiveness of these models in accurately capturing the semantic essence of opcode sequences.

To address the challenge of transforming opcode sequences into vector embeddings, we have adopted an innovative approach, synergizing the powerful capabilities of the pre-trained model CodeBERT with the TextCNN model. This amalgamation precisely maps the API's opcode sequences into embedded vectors. CodeBERT provides us with enriched contextual information and the ability to capture long-distance dependencies, while TextCNN excels in autonomously identifying and extracting valuable local features. Moreover, we have further integrated permission features to achieve a more comprehensive description of API node characteristics. This methodology enables us to construct an API call graph infused with intricate semantic layers, where each graph distinctively represents an APK. Building on this foundation, we incorporate two graph neural network technologies: GraphSage and SAGPooling. Utilizing the sampling-based inductive learning framework of GraphSage, we aggregate information from neighboring nodes and capture the local neighborhood structure, facilitating effective learning of node representations from the API call graph and enhancing our understanding of API call patterns. Additionally, we leverage the self-attention mechanism of SAGPooling to preserve the most informative nodes during the pooling process, aiding the model in accurately capturing key structural features of the call graph. This aspect is crucial for malware detection, as malicious behaviors may manifest in specific sections of an APK. The integration of these technologies provides our Android malware detection framework, GSEDroid, with a robust learning infrastructure. This enables us to deeply understand and identify complex patterns hidden within the API call graph, significantly enhancing our ability to detect malicious software.

In summary, the contributions of this paper are threefold:

1. We introduce a novel embedding model for representing Dalvik opcode sequences, employing the combined strengths of the pre-trained CodeBERT and TextCNN models to facilitate efficient semantic embedding of APIs.
2. We amalgamate various features such as permission attributes of APK, API call graph structural characteristics, and semantic features of Dalvik opcode sequences within APIs. This leads to the development of GSEDroid, a framework that leverages advanced graph classification algorithms for the detection of Android malware.
3. The efficacy of GSEDroid is substantiated through extensive comparative experiments. The combined graph model of GraphSage and SAGPooling achieved a detection accuracy of 99.47% on public datasets.

The remainder of this paper is organized as follows: Section 2 provides an overview of the related work. Section 3 delineates the proposed Android malware detection framework, GSEDroid. Section 4 describes our experimental dataset, experimental setup, analysis of the results, and comparisons with other detection methodologies. Section 5 offers insights into future work, and Section 6 concludes the paper.

2. Related work

Current methodologies for analyzing Android malware are categorized into dynamic and static approaches. Dynamic methods (Alzaylaee et al., 2020; Jerbi et al., 2020; Saracino et al., 2018; Wong and Lie, 2018) involve installing and running applications in sandbox environments or on real devices to differentiate between benign and malicious software. Numerous Android malwares employ advanced obfuscation techniques such as bytecode encryption (Rastogi et al., 2013), Java reflection, and dynamic code loading to circumvent detection systems. Additionally, specific issues are targeted for resolution, such as malware evolution (Fang et al., 2023; Jerbi et al., 2022), concept drift (Chen et al., 2023; Guerra-Manzanares and Bahsi, 2022; Guerra-Manzanares et al., 2022), and model aging (Xu et al., 2022; Zhang et al., 2020). Conversely, researchers predominantly employ static analysis techniques to gather information about application behaviors. Static analysis offers a comprehensive snapshot of the application without requiring execution.

2.1. Static method based on feature optimization

JOWMDroid (Cai et al., 2021) innovated with a novel feature weighting static analysis methodology. This approach utilizes Information Gain (IG) for the selection of features contributing substantial information and employs the Differential Evolution (DE) algorithm for the joint optimization of weight mapping functions and classifier parameters. However, this method grapples with challenges related to high feature dimensionality and the complexity of delineating inter-feature correlations.

To address these issues, CENDroid (Badhani and Muttoo, 2019) integrates static features, such as API tags and permissions, with clustering and ensemble classifiers. This technique has demonstrated proficient performance with feature sets composed of APIs and permissions in classification tasks. Despite its exemplary performance across various classifiers, as a static analysis tool, CENDroid encounters limitations due to dynamic loading and reflection.

FARM (Han et al., 2020) introduces a feature transformation-based Android malware detector, incorporating both established detection functionalities and three novel types of feature transformations.

Kim et al. (2019) proposed a multimodal deep learning approach for malware detection, which, by analyzing components like Manifest files, Dex files, and .so files in APKs, maximizes the inclusion of a diverse range of feature types, thereby enriching the information extracted and encapsulating the characteristics of applications more comprehensively.

Furthermore, FCSCNN (Kong et al., 2022) introduces the Feature Centralized Siamese Convolutional Neural Network, a design strategy that effectively selects valuable permission and API call features. This method achieves high-performance Android malware detection by calculating the distance between APKs and the mean centers of benign and malicious samples, thereby optimizing both detection performance and speed.

2.2. Opcode-based detection with CodeBERT and TextCNN

2.2.1. Opcode-based detection

Initially, the frequency distribution of opcodes was confirmed in studies by Bilar (2007); Yewale and Singh (2016) as an effective feature for identifying malware. Later, the processing of opcode features has been roughly divided into two categories, one is processed as a graph, and the other is processed as a sequence.

Anderson et al. (2011) transformed opcode sequences into Markov chain constructs, utilizing Gaussian and spectral kernels to develop similarity matrices. Their findings also indicated diminished performance when instructions with operands were used. Runwal et al. (2012) constructed opcode graphs and implemented malware detection through similarity scoring. Hashemi et al. (2017) adapted Anderson et al. (2011) construction approach, uniquely employing a “power iteration” method to embed these graphs into the feature space. Zhang et al. (2018) developed a weighted probability graph of Dalvik opcodes and employed information entropy to prune the graph, ultimately utilizing global topological similarity for maliciousness detection. The novelty of G3MD (Khalilian et al., 2018) lies in applying graph mining to opcode graphs to extract frequent subgraphs for malware detection.

Moskovitch et al. (2008) utilized TF-IDF to represent opcode sequences and applied Fisher scores for feature reduction. Zhang et al. (2018) represented opcode clusters using an opcode bi-gram matrix, detecting malware through binary probability distribution. Yuxin and Siyi (2019) used N-grams to represent opcode sequences, capturing local dependencies and pattern features between opcodes. Kang et al. (2019); Khan et al. (2021) represented opcodes based on word2vec, employing the LSTM method for dimensionality reduction. Jiang et al. (2019) encoded opcodes into feature vectors using Doc2vec. Jeon and Moon (2020) compressed longer opcode sequences into shorter ones using an Opcode-level Convolutional Autoencoder (OCAE), followed by a Dynamic Recurrent Neural Network (DRNN) classifier for malware detection. DroidRL (Wu et al., 2023) also utilized N-grams for opcode sequence processing, leveraging reinforcement learning for automatic feature selection and employing the Double Deep Q-Network (DDQN) algorithm to select optimal feature subsets, addressing the “dimensionality explosion” issue inherent in N-grams.

2.2.2. CodeBERT and TextCNN

CodeBERT (Feng et al., 2020) is structured on a multi-layer bidirectional transformer network, wherein each Transformer block comprises a Multi-Head Self-Attention mechanism and a Feed-Forward layer. The architecture encompasses 12 layers, with 12 Attention heads per layer, a head size of 64, 768 hidden units, and 125 million parameters, using a model architecture that is basically consistent with CRoBERTa-base (Liu et al., 2019). CodeBERT represents an extension of BERT with a dual-mode capability. It eschews the context matching pre-training objective (Next Sentence Prediction, NSP) employed by BERT (Devlin et al., 2019), retaining the Masked Language Modeling (MLM) pre-training objective and introducing the Replacement Token Detection (RTD) as its additional pre-training target. CodeBERT is distinctively trained using both natural language and source code data. The self-attention mechanism facilitates contextually relevant encoding, while the fully connected layers map the encoded vectors to new vector spaces. This integrative approach enables CodeBERT to capture the bidirectional nuances of opcode sequences and to deeply comprehend the long-distance contextual dependencies within these sequences, thereby providing potent feature representations for subsequent detection tasks.

Utilizing TextCNN (Kim, 2014), local information akin to N-gram concepts can be extracted from the sequences. By integrating features of different-sized N-grams as representations of the entire opcode sequence, local features in the context of opcode sequences are essentially sliding windows composed of several opcodes. The advantage of convolutional neural networks lies in their ability to autonomously combine and filter N-gram features, thereby obtaining semantic information at various levels of abstraction.

2.3. Graph-based detection and graph neural network

2.3.1. Graph-based detection

S^3 Feature (Ou and Xu, 2022) extends the Function-Call Graph (FCG) by marking sensitive nodes, resulting in a Sensitive Function-Call Graph (SFCG). This process involves mining a plethora of sensitive subgraphs with suspicious behaviors and their neighboring subgraphs from the SFCG, which are then encoded into feature vectors to represent the application.

Frenklach et al. (2021) introduced a static analysis approach for Android applications based on an Application Similarity Graph (ASG). Their research posits that the key to classifying application behaviors lies in their common reusable building blocks, such as modules with identical functionalities.

To identify shared features among malicious software samples, Navarro et al. (2018) proposed an ontology-based framework to model the complex network of relationships among components, attributes, and interfaces related to the permission mechanism within the Android ecosystem. This framework elucidates the relationships between applications and system elements, identifying shared features of malware samples within this high-level abstraction. However, a significant limitation of this approach is the exponential increase in time and space requirements as the graph size grows.

MSDROID (He et al., 2023) also focuses on code segments with malicious intent. By concentrating on these malicious code segments and providing an interpretative mechanism, MSDROID enhances the understanding of malware behavior and patterns, offering a more nuanced approach to malware detection.

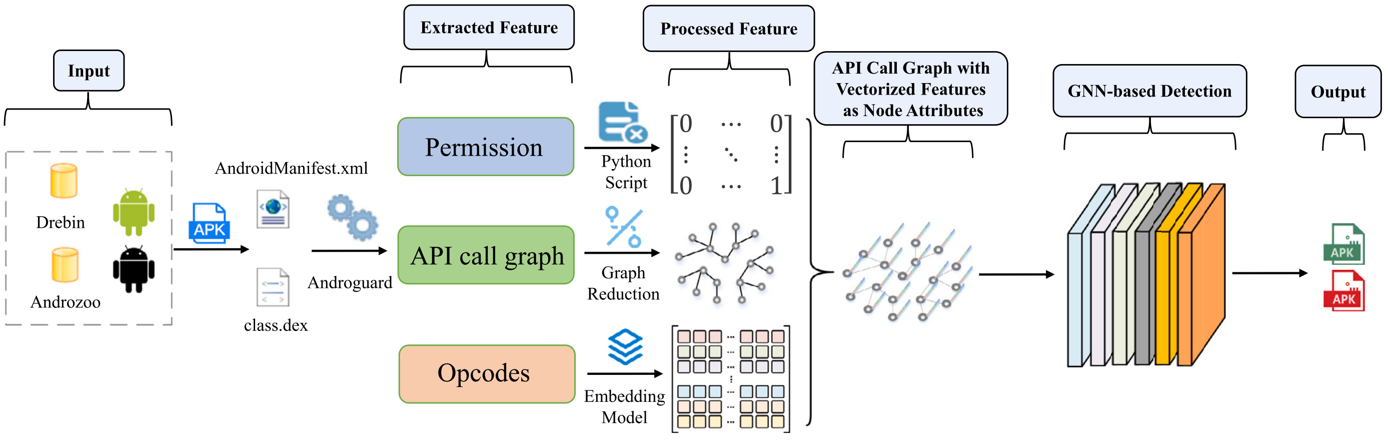


Fig. 1. Framework Overview.

2.3.2. *Graph neural network*

Throughout the evolution of Graph Neural Network (GNN) technology, the initial focus was primarily on how to effectively learn low-dimensional representations for each node within graph-structured data. Initially, GNN research was centered around spectral-method-based models, which perform convolution operations in the Fourier space of graphs. One of the early milestones, ChebNet (Defferrard et al., 2016), employed Chebyshev polynomials to filter node features within the graph's Fourier space. Subsequently, GCN (Kipf and Welling, 2017) simplified the parameterization of ChebNet, effectively reducing overfitting and enhancing the model's performance in practical applications.

However, spectral-method GNNs, due to their reliance on the overall graph structure and the eigenvectors of the graph Laplacian, faced limitations in scalability and generalizability. To overcome these constraints, researchers proposed spatial-method-based GNNs. These GNNs define convolution operations directly in the graph space, aggregating adjacent features of nodes to emulate the way Convolutional Neural Networks process image data. GraphSAGE (Hamilton et al., 2017) marked a key innovation in this phase, generating node embeddings effectively through universal aggregator functions, thereby fostering the model's generalizability to new nodes or graphs. Following this, GAT (Veličković et al., 2018) introduced an attention mechanism, allowing the model to assess the importance of each neighbor's information from the perspective of the target node. Further research brought forth GIN (Xu et al., 2019a), which enhanced the model's expressive power for graph data through meticulous control of the graph structure.

Additionally, various pooling techniques were developed to improve GNN performance, aimed at efficiently reducing dimensions and extracting representative features of graphs. Set2Set (Vinyals et al., 2016) allows GNNs to process sets of nodes, capturing complex relationships within graph-structured data. GlobalAttentionPooling (Li et al., 2017) employs an attention mechanism to weight node features globally. SAGPooling (Lee et al., 2019) selectively aggregates nodes at each layer to form a hierarchical representation of the graph.

3. Proposed approach

As shown in Fig. 1, the overall framework of GSEDroid includes original feature extraction and feature processing, building an API call graph with node attribute characteristics to represent the APK, and transforming Android malware detection problems into graph classification tasks using graph neural networks. Algorithm 1 shows the generation algorithm for building an API call graph with node attribute characteristics.

Table 1
Top 10 Ranked Features.

Top 10 Permissions
'android.permission.INTERNET'
'android.permission.ACCESS_NETWORK_STATE'
'android.permission.WRITE_EXTERNAL_STORAGE'
'android.permission.ACCESS_WIFI_STATE'
'android.permission.READ_PHONE_STATE'
'android.permission.WAKE_LOCK'
'android.permission.VIBRATE'
'android.permission.READ_EXTERNAL_STORAGE'
'android.permission.ACCESS_FINE_LOCATION'
'android.permission.ACCESS_COARSE_LOCATION'

3.1. *Feature processing module*

In Android, there are two primary categories of permissions: AOSP (Android Open Source Project) permissions and third-party permissions. AOSP permissions are those that are inherent to the Android operating system, defined by the developers and maintainers of Android. These permissions regulate the access of applications to device and system resources. On the other hand, third-party permissions are crafted by independent developers or organizations and are requested by third-party applications, becoming active upon user authorization. The dataset contains approximately 12,000 third-party permissions (exactly 12,564). Since most API methods in APKs necessitate AOSP-related permissions and considering the common misuse of Android's fundamental permissions by malware, we concentrated on 334 AOSP permissions extracted from the dataset. From these, we selected the top 64 permissions based on request frequency. This study involved the collection of permission-related information detailed in Table 1 by parsing tags within the Manifest files. The existence of the extracted request permissions was used as the basis for constructing the permission feature vector.

Since some APIs are methods from third-party libraries without accessible opcode sequences, and others are linked with native code via JNI, often written in languages such as C/C++, we aimed to standardize the opcode types input into the model. Furthermore, to match the number of nodes in the API call graph with those having obtainable features, we used and adapted the Androguard tool (Desnos and Gueguen, 2012), an open-source Python library for decompiling and analyzing application files. While constructing the API call graphs, we excluded methods not defined in the DalvikVMObject, including certain third-party library methods. These may consist of diverse code types, such as static Java bytecode, dynamically loaded runtime code, and methods in native languages like C/C++ linked via JNI (.so files) (Qiu et al., 2022). We also automatically filtered out isolated nodes, i.e., nodes that neither call other APIs nor are called by other APIs. This pruning of the call graph

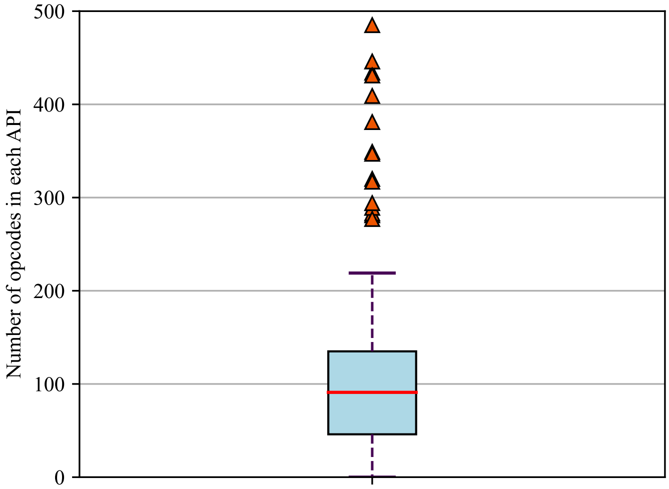


Fig. 2. Distribution of Opcode Counts in APIs.

Table 2
Dalvik Opcode Mapping Token Table.

Dalvik opcode	Mapping token
move;move/16;move/from16	[“move”]
add-double;add-double/2addr	[“add”,”double”]

const;const/16;const/4;
const/high16;
	[“const”]

float-to-int	[“float”,”to”,”int”]
if-eqz	[“if”, “equal”, “zero”]
iput-byte	[“instance”, “put”, “byte”]
return-void	[“return”, “void”]
invoke-virtual; invoke-virtual/range	[“invoke”, “virtual”]
...	...

aims to reduce its complexity while retaining sensitive APIs commonly found in malicious software as much as possible.

Subsequently, we employed Python scripts to analyze the semantic features of API opcode sequences. There are approximately 218 different Dalvik opcodes.

Fig. 2 shows the number of opcodes contained in each API. Most APIs contain less than 250 opcodes. In order to help CodeBERT identify Dalvik opcodes and include them in the vocabulary, we performed the mapping operation on the opcodes as shown in Table 2.

3.2. Module that constructs an API call graph containing node features

Algorithm 1 : API call graph generation with node feature.

```
Input: apk
Output: API call graph with feature vector set: ${\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_n}$
$init\_graph \leftarrow$ androguard($apk$); // API call graph
$node\_vec \leftarrow$ an empty array;
for api in androguard($apk$) do
    $per\_vec \leftarrow$ GetPerVec($api$);
    $op\_seq \leftarrow$ androguard($api$);
    $E \leftarrow$ CodeBERT-Base(op_seq); // $E \in \mathbb{R}^{N\times 768}$
    $E_{\text{padded}} \leftarrow \text{Padding}(E)$;
    $F \leftarrow \{\text{Conv}(E_{\text{padded}}, \text{size}) \text{ for size} \in [2,9]\}$;
    $P \leftarrow \{\text{KMaxPool}(A_{\text{size}}, 1) \text{ for } A_{\text{size}} \text{ in } F\}$;
    $P_{\text{concat}} \leftarrow \text{Concat}(P)$;
    $op\_vec \leftarrow \text{Lin}(P_{\text{concat}})$;
    $api\_vec \leftarrow \text{Combine}(op\_vec, per\_vec)$;
    add $api\_vec$ to $node\_vec$;
end for
$final\_graph \leftarrow \text{Integrate}(init\_graph, node\_vec)$;
return $final\_graph$;
```

Fig. 3 illustrates the embedding model process for handling opcode sequences. As outlined in Algorithm 1, opcode sequences from APIs are fed into the pre-trained CodeBERT-base model. Through a series of 12 stacked Transformer Encoder Blocks, we obtain vector representations for each token in the CodeBERT output, forming a word vector matrix of dimensions (N*768). For opcodes like “add-double” in Table 2, we decompose and map them into two tokens: “add” and “double.” This approach aims to maximize the utilization of the prior knowledge embedded in the pretrained model, CodeBERT. To achieve this, it’s crucial to ensure that all tokens mapped from opcodes are included in CodeBERT’s vocabulary. Therefore, the ‘N’ here does not represent the number of opcode sequences, but rather the number of tokens corresponding to the mapped opcode sequences. The sequences are then processed by a TextCNN layer. To ensure uniform vector lengths, a padding or truncation operation is performed based on the range of opcode counts mentioned earlier. Convolution operations are conducted using kernels of varying sizes, with ReLU functioning as the activation function. Each kernel size is represented with 64 filters, as noted by Zhang and Wallace (2016), who mention that employing multiple filters of the same size aims to learn complementary features from the same window. Subsequent K-Max pooling (with k set to 1) selects the top K features from each feature vector, which are then concatenated to form a composite vector representation. A linear layer then compresses these features into fixed dimensions of 32, 64, 96, 128. The extracted permission feature vectors and opcode feature vectors are concatenated and merged with the API call graph, creating a graph representation of the APK.

Combining the pre-trained CodeBERT model with the TextCNN model for feature vector extraction from Dalvik opcode sequences offers the advantage of integrating multi-layer semantic information. CodeBERT, with its self-attention mechanism, is capable of capturing long-distance dependencies and complex contextual information within sequences. The model, having learned extensive prior knowledge from natural language and programming language data during pre-training, can capture rich semantic representations, especially since Dalvik opcodes closely resemble programming and natural languages. This implies that the model is better equipped to understand the relationships between instructions and data flows. TextCNN (Text Convolutional Neural Network), on the other hand, further extracts local features by combining feature extraction layers with different filter sizes, thus obtaining features from information of various granularities. The opcode sequences may contain specific syntactic structures or patterns, which TextCNN can capture through convolutional operations, aiding in the extraction of key features from the opcode sequences. The combination of these two approaches provides a robust feature extraction mechanism, analyzing input sequences from different perspectives, enhancing the model’s understanding of Dalvik opcode sequences, effectively improving the quality of feature extraction, and capturing semantic information more comprehensively.

Combining these two methods allows the embedding vectors to be compressed into a smaller-dimensional vector, improving the model’s training speed and reducing computational costs. Fine-tuning CodeBERT typically requires relatively more time, as it demands large datasets and multiple training iterations; fine-tuning TextCNN, however, demands less time due to the model’s smaller size, faster training speed, and higher computational cost. In resource-constrained environments, using CodeBERT alone may be inefficient. TextCNN, being a lightweight convolutional neural network, can reduce computational demands while maintaining performance.

3.3. GNN-based detection module

In our final stage, the graphs are fed into a Graph Neural Network (GNN) for graph classification tasks. The GNN framework we utilize comprises three primary components: filtering layers, activation layers, and graph pooling layers. Within this framework, the function of the

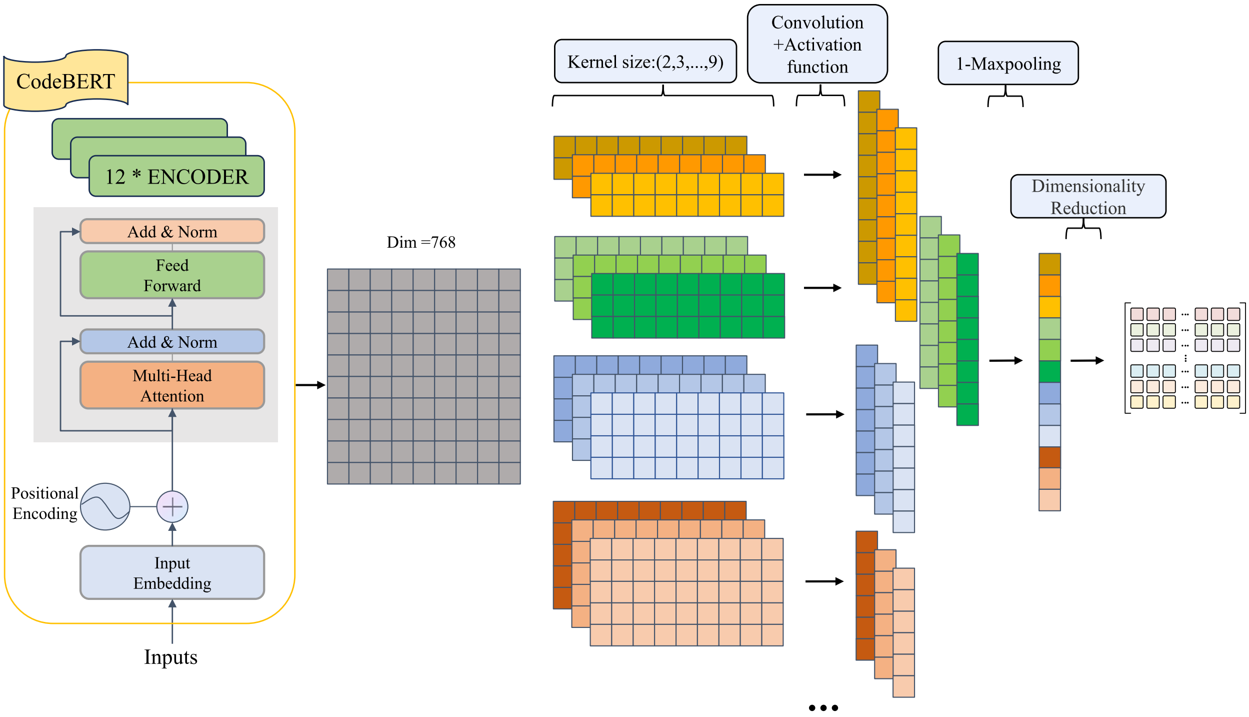


Fig. 3. Embedding Model for Dalvik Opcode Sequences.

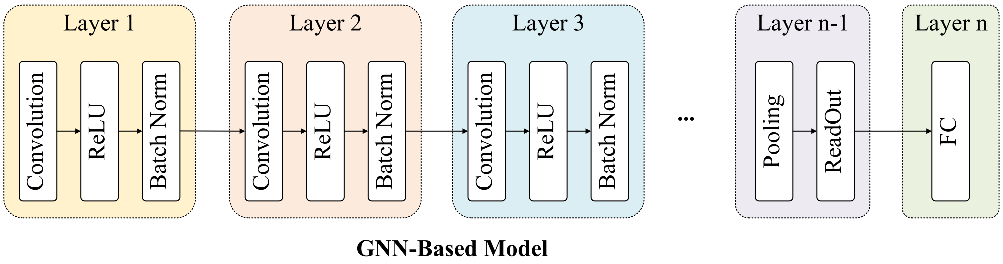


Fig. 4. Detection Model Based on Graph Neural Networks.

filtering and activation layers is to generate enhanced API node features, while the graph pooling layers aggregate these node features to produce high-level characteristics capable of capturing the complete information of the call graph. This process results in coarsened graphs that possess more abstract and higher-level node features. The structure of our GNN in Fig. 4, consists of several blocks composed of filtering layers, activation layers, and regularization, followed by graph pooling and ReadOut layers, culminating in a graph-level representation that is processed through a fully connected layer to produce the output. Selecting appropriate filtering and graph pooling layers is crucial in GNN construction, as their design directly impacts the model's performance and its ability to comprehend graph structures.

3.3.1. Filter layer

The filtering layer we ultimately employed is SAGEConv, which combines the features of a node's neighbors with its own to generate new node representations. The formula for SAGEConv is as follows:

$$H_i^{(l+1)} = \sigma\left(W \cdot \text{AGGREGATE}_{j\in\mathcal{N}(i)}\left(H_j^{(l)}\right)\right)$$

Here, $\mathcal{N}(i)$ represents the neighbors of node i, W is the weight matrix, and AGGREGATE is a differentiable, permutation-invariant function (such as mean, sum, or max pooling). The unique aspect of GraphSage is its inductive capability, meaning it can generalize to unseen nodes, which is particularly crucial for dynamically changing Android applications.

3.3.2. Graph pooling layer

The graph pooling layer we selected is SAGPooling, or Self-Adaptive Graph Pooling. This layer orders and selects nodes based on scores learned through training, thereby extracting key subgraph structures. It computes a score for each node, followed by node downsampling or pooling based on these scores. If X represents the node features and S is the score vector, the process of SAGPooling can include the following steps:

$$S = \text{GNN}_{score}(X)$$

$$X' = \text{Top-K}(S, X)$$

$$H = \text{GNN}_{pool}(X')$$

Here, GNN_{score} is a score computed by a graph neural network, Top-K is a selection operation that chooses the top K nodes in X based on scores S, and GNN_{pool} is another graph neural network that performs a pooling operation on the selected node features X'.

SAGPooling allows the model to sift through the entire graph to filter out key features, thereby achieving higher accuracy in classification tasks. By integrating the strengths of GraphSage and SAGPooling, our model is not only more powerful in feature extraction but also more efficient in processing large-scale and complex API call graphs, a necessity for identifying and classifying thousands of Android applications.

4. Experiments and results

4.1. Dataset

To ensure that our data samples remain contemporary and to enhance the universality and persuasiveness of our detection results, we not only selected 4,145 malicious APKs from the earlier Drebin dataset (Arp et al., 2014) but also downloaded approximately 86,000 APKs (69,115 benign and 24,052 malicious) spanning the years 2018 to 2023 from AndroZoo (Allix et al., 2016). The benign APKs were each verified as non-malicious by VirusTotal, with no anti-virus engines flagging them as malicious. Conversely, each malicious APK was identified as such by at least three anti-virus engines via VirusTotal (Sistemas, 2004). From this collection, we selectively curated 6,754 benign and 4,509 malicious APKs, dating from 2018 to 2021, with API counts ranging between 1,000 to 6,000. Consequently, the total number of APKs utilized for our final experiments amounted to 15,408, including 6,754 benign and 8,654 malicious.

4.2. Baseline

In our study, we benchmarked against several baseline methods:

1. Drebin (Arp et al., 2014) captures eight groups of string sets from APKs, including requested hardware components, permissions, application component names, filtered intents, restricted API calls, actual used permissions, suspicious API calls, and network addresses. These string sets form feature vectors that are then classified using a Support Vector Machine (SVM).

2. Mamadroid (Mariconti et al., 2017) constructs Markov chains by evaluating the probabilities of all transitions between abstract API calls derived from abstract API call sequences. Feature vectors are then derived for each application based on its Markov chain, where each nonzero vector component corresponds to the probability in the Markov chain. Classification is performed using machine learning algorithms. Specifically, we employed the most effective classifier reported in the paper—Random Forest. For family patterns, we used 51 trees with a maximum depth of 8, while for package patterns, we used 101 trees with a maximum depth of 64.

3. DroidEvolver (Xu et al., 2019b) constructs feature vectors based on the presence of API calls and then establishes a model pool with a set of online learning algorithms, rather than any single detection model for malware detection. This model pool helps detect and mitigate biases of any single detection model, yielding more reliable detection outcomes during the detection phase. The five linear online learning algorithms used include Passive Aggressive (PA), Online Gradient Descent (OGD), Adaptive Regularization of Weight Vectors (AROW), Regularized Dual Averaging (RDA), and Adaptive Forward-Backward Splitting (Ada-FOBOS).

4.3. Experiment setup

In experiments, GPUs are used to accelerate the graph deep learning algorithm we use for detection. We implemented the detection module of the framework using Pytorch and DGL; using Dell-Precision-7920-Tower server, configured with two NVIDIA RTX A5000 GPUs with 24GB memory size, 64GB memory size, and 96 CPU cores. The software used is Python = 3.8.16, Androguard = 3.4.0a1, DGL = 1.1.1+cu117.

The following are the evaluation indicators of the experiment:

True positive (TP): the number of positive samples that are correctly predicted as positive.

False negative (FN): the number of positive samples that are incorrectly predicted as negative.

False positive (FP): the number of negative samples that are incorrectly predicted as positive.

True negative (TN): the number of negative samples that are correctly predicted as negative.

Table 3
Final Selection of Hyperparameters.

Hyperparameters	Search Range	Final Chosen
Learning Rate	[0.001,0.01,0.1]	0.001
Dropout	[0.2,0.3,0.4,0.5]	0.3
Batch Size	[32,64,96,128]	64
Number of Layers	[3,4,5,6]	4
Number of Hidden Units	[32,64,96,128]	128
Epochs	[50,100,150,200]	50
Optimizer	[Adam,Sgd,Adagrad]	Adam
Loss Function	[Cross_entropy,mse,Huber]	Binary Cross-Entropy
Activation Functions	[Relu,Elu,Sigmoid,Tanh]	Relu

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

4.4. Results and analysis

In this study, we employed a 10-fold cross-validation method, dividing the dataset into an 8:1:1 split. Initially, eight parts of the data were used for training, with hyperparameter selection determined based on the performance on the validation set. Subsequently, for the final experiments, nine parts of the data were used for training and one part for testing. We utilized grid search for hyperparameter tuning on the validation set and applied a common regularization technique, Early Stopping, to monitor the model's performance on the validation set during training. Training is halted when performance ceases to improve, preventing overfitting and saving training time.

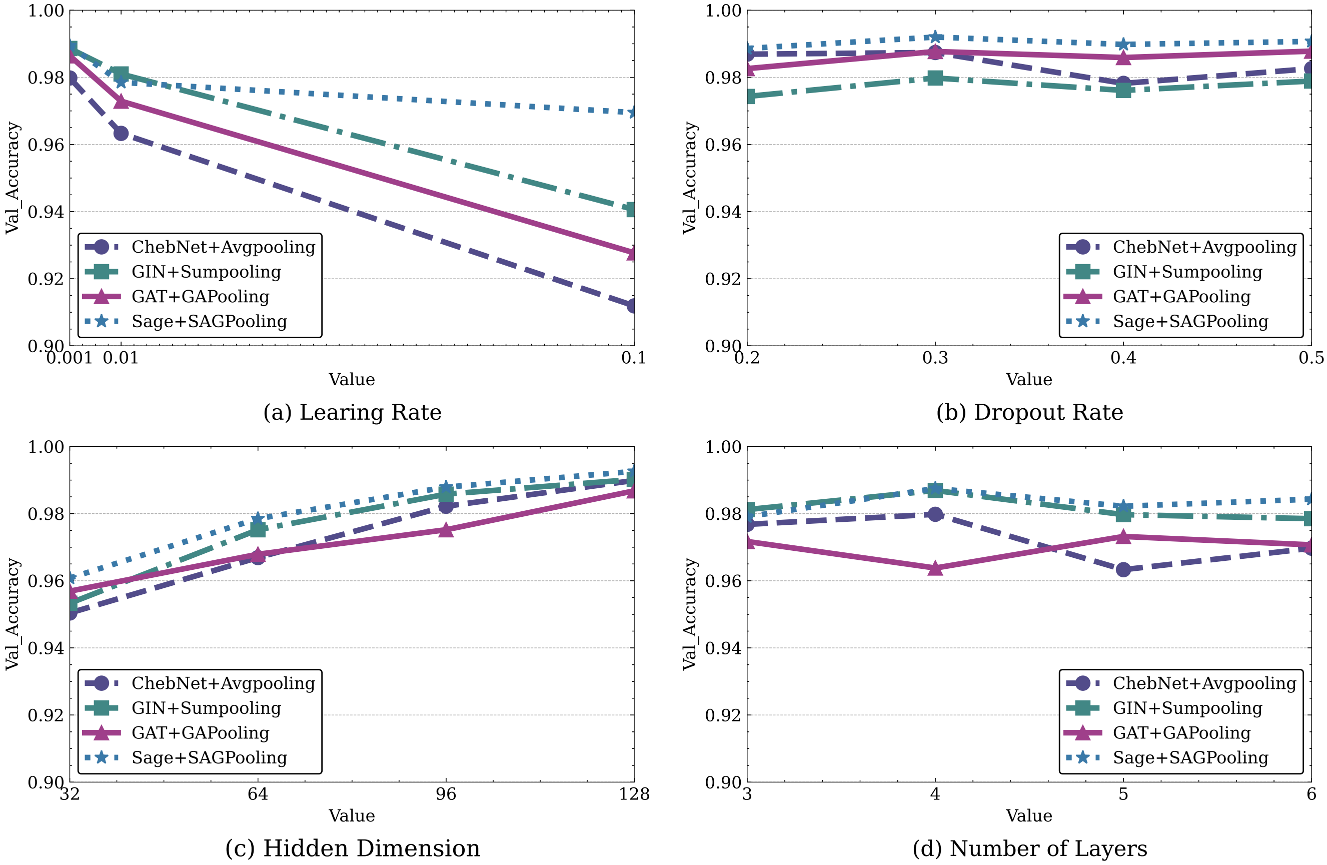
4.4.1. Hyperparameter selection

Fig. 5 illustrates the performance of the selected GraphSage+SAGPooling model on the validation set, with the final hyperparameters detailed in Table 3. Aside from common hyperparameter settings, the following are additional configurations for selected models: in GAT, the number of attention heads (num_heads) is set to 4; in Set2set, the number of iterations (n_iters) is 6 and the number of recurrent layers (n_layers) is 3; the Chebyshev filter size is set to 3; and in SageConv, the aggregator type is set to 'pool'.

During the training process depicted in Fig. 6, the GraphSage+SAGPooling model's accuracy on the training set rapidly climbed to near 100%, demonstrating the model's robust data-fitting capabilities. As the number of iterations increased, the test accuracy consistently remained slightly lower than the training accuracy, likely due to the model learning excessive noise in certain samples of the training set, which limited its generalization capability. The sharp decline in the loss curve further corroborated the model's rapid learning and convergence. However, the loss plateaued at a low level near zero, indicating that the model's performance had essentially reached its peak before 50 epochs. Overall, the model exhibited commendable testing effectiveness and reasonable generalization performance.

4.4.2. Comparison with different graph neural network algorithms

We also compare our model with other graph filtering methods, including ChebConv, GraphConv, GINConv, and GATConv, as well as different graph pooling methods like SumPooling, AvgPooling, GlobalAttentionPooling, and Set2Set. Results presented in Fig. 7 indicate that the GraphSage+SAGPooling model surpasses other models across all metrics, establishing it as the most robust choice for tasks demanding high precision, recall, and accuracy. These experimental findings suggest that the performance of a Graph Neural Network (GNN) is relatively stable across different combinations of filtering layers and pooling



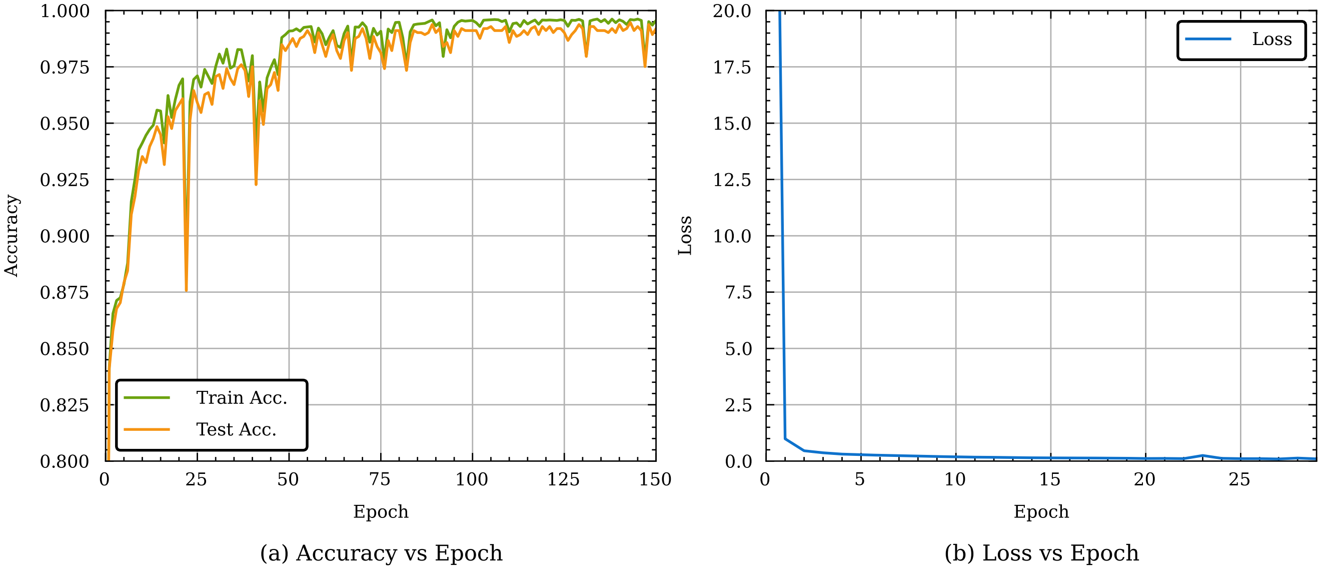
(a) Learing Rate

(b) Dropout Rate

(c) Hidden Dimension

(d) Number of Layers

Fig. 5. Hyperparameter Tuning Result.



(a) Accuracy vs Epoch

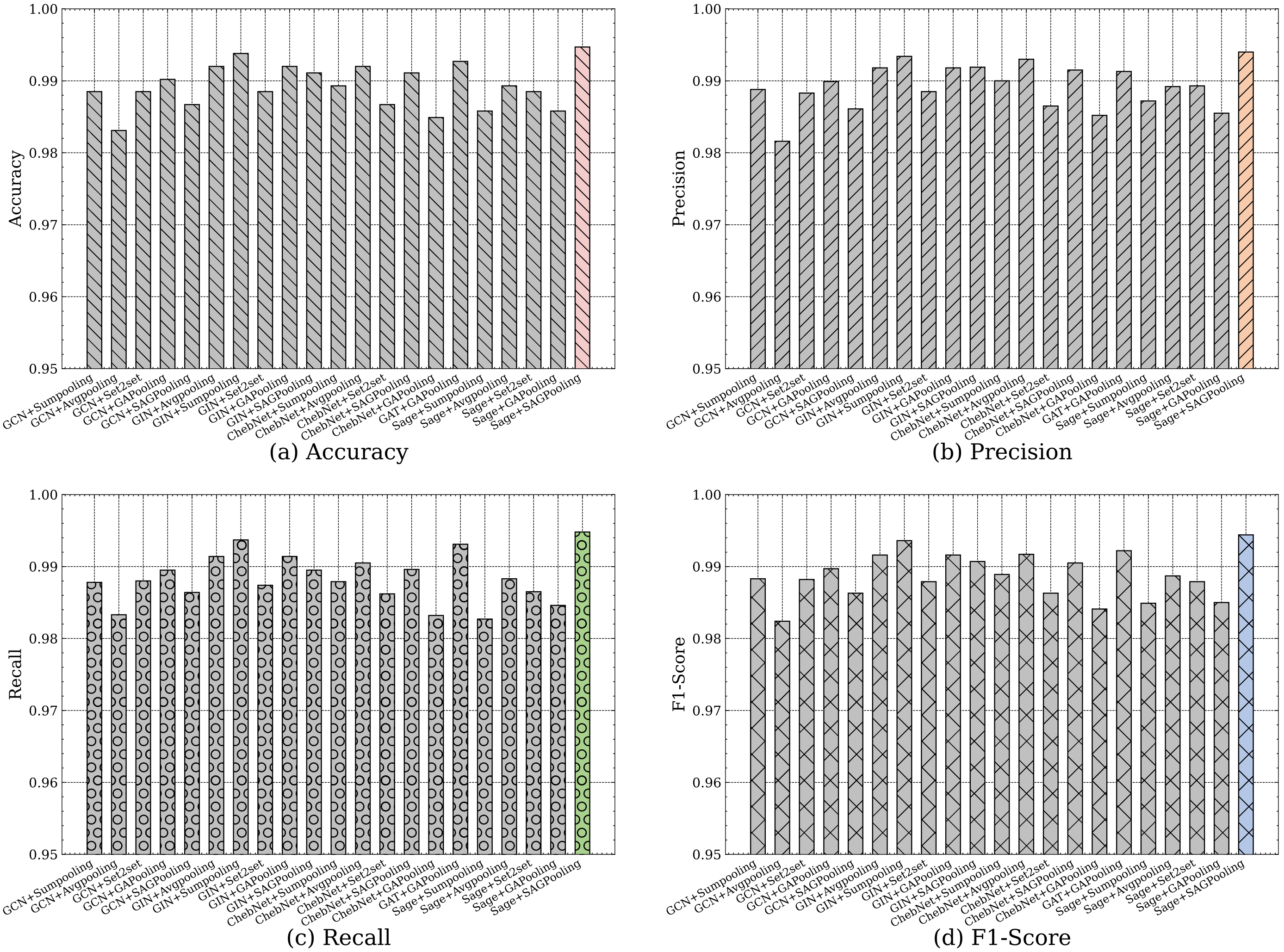
(b) Loss vs Epoch

Fig. 6. Accuracy/Loss vs Epoch. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

methods used for malware classification. This stability underscores the effectiveness of our embedding representation method. The combinations of GATConv+GlobalAttentionPooling and GINConv+SumPooling also demonstrate very high detection performance, making them viable and robust alternatives.

4.4.3. Comparison with different embedded models

In Fig. 8, we further compare our method with other embedding approaches such as TF-IDF, Word2vec (CBOW), Word2vec (SG), Doc2vec (PV-DM), Doc2vec (PV-DBOW), LSTM (Word2vec), GRU (Word2vec), and Bert-base-uncased, each generating node embedding features in dimensions of 32, 64, 96, 128. Since TF-IDF is mainly based on word frequency statistics, it ignores context and lacks semantic information, resulting in its worst representation. Word2vec represents opcode sequences by averaging the embeddings of all words in the sequence. Given that CBOW overlooks the order and semantics of opcodes, Skim-Gram naturally performs slightly better. However, simply averaging sequences does not adequately consider their order, so Doc2vec,



(a) Accuracy

(b) Precision

(c) Recall

(d) F1-Score

Fig. 7. Detection Performance of Different Graph Neural Networks.

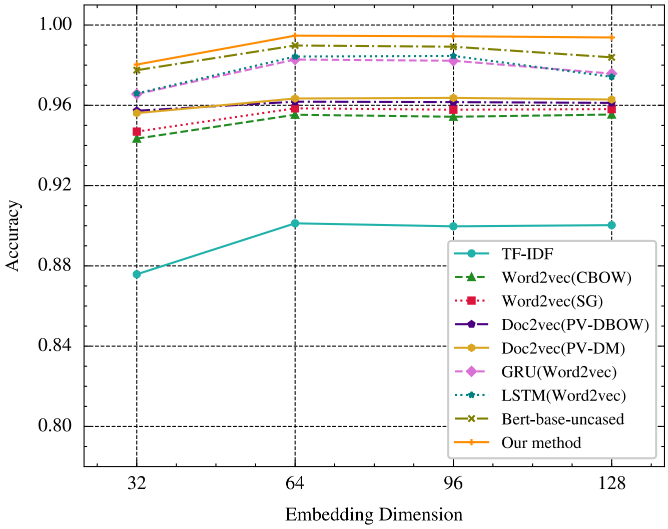


Fig. 8. Comparative Analysis of Embedding Methods Across Various Dimensions.

which adds modeling of sequence information, compensates for some of Word2vec's limitations. RNNs possess the capability to capture interactions between opcodes that are distantly located within a program, an aspect where Doc2Vec might fall short. Bert-base-uncased uses the embedding of the [CLS] token to represent the entire sequence, showing effectiveness second only to our method. Our analysis indicates that Bert-base-uncased, using Word-Piece to find the most probable combinations for its vocabulary, might incorrectly segment certain opcodes (like 'invoke', 'bool'). In contrast, CodeBERT uses BPE for frequent combination-based vocabulary construction and is trained on both programming and natural languages, resulting in a larger vocabulary and, as evidenced, superior performance over Bert-base-uncased. All these methods are evaluated using the GraphSage+SAGPooling model for graph classification. Results show that our embedding model contributes most effectively to classification accuracy, with 64-dimensional features already representing opcode sequences well. This suggests that our embedding model most closely captures the semantic representation of opcode sequences, aiding the graph neural network in better learning the behavioral patterns of malicious software.

4.4.4. *Ablation experiment*

The ablation study in Table 4 reveals the impact of various combinations of feature information on model performance. The data indicate that models relying solely on API call graphs are already quite effective, suggesting that API call graphs inherently contain extensive information about the behavior of malicious software. The incorporation of permission information into the API call graph base enhances model accuracy. Impressively, when API call graphs are merged with semantic information, there is a notable increase in model accuracy, highlighting the vital role of semantic information in deciphering patterns within the API call

Table 4
Ablation Study Results.

Method	Accuracy
Only API call graph	0.9079
API call graph + permission	0.9232
API call graph + semantic information	0.9896
API call graph + permission +semantic information	0.9947

Table 5
Comparative Analysis of Android Malware Detection Methods.

Method	Accuracy	Run time
Drebin	0.9732	4.53 s
MamaDroid(family)	0.9550	22.89 s
MamaDroid(package)	0.9725	35.47 s
DroidEvolver	0.9766	13.18 s
GSEDroid	0.9947	12-14 s

graph. Ultimately, combining all these types of information culminates in a model achieving an astonishing accuracy of 0.9947, underscoring the significance of integrating diverse information types.

In conclusion, while a standalone API call graph provides considerable insights into app behavior, combining it with other features like permissions and semantic information significantly boosts model performance. This holistic approach enables a more comprehensive and in-depth understanding of the behavior and internal structures of malicious software, leading to outstanding detection results.

4.4.5. Comparison with other approaches

Table 5 demonstrates that, compared to existing open-source Android malware detection methods, our approach still achieves the best detection results. Drebin characterizes APKs through feature vectors containing observable strings from eight feature sets, achieving commendable detection effectiveness for a lightweight static method. In MamaDroid, the abstraction in family mode is more lightweight, hence faster in feature extraction, while the package mode offers a finer granularity and more information, resulting in better detection performance but at the cost of increased time and space resources. Among the five linear online learning algorithms used in DroidEvolver, Adaptive Regularization of Weight Vectors (AROW) (Crammer et al., 2009) performs best, whereas Adaptive Forward-Backward Splitting (AdaFOBOS) (Duchi et al., 2011) shows the least effectiveness.

During the experiments, we tracked and analyzed the time spent at each stage of the method. The analysis results are shown in Table 5. The running time column in Table 5 is the average time for processing and detecting an apk. The feature extraction stage is the most time-consuming aspect of our method, particularly the embedding model for opcode sequences, totaling approximately 34 hours. It takes around 8.24 seconds per benign apk and 7.84 seconds per malicious apk. In the dataset, extracting the API call graphs of all apks (excluding node features) took a total of 13.5 hours, with an average of 3.16 seconds for benign apks and 3.18 seconds for malicious apks. In comparison, the time required to calculate the permission vector for each API is negligible, and integrating the node features into the API call graph took a total of 2 hours and 44 minutes. The graph model takes approximately 17 minutes and 13 seconds to train, while the inference process only requires 1.15 seconds, which is extremely brief. Therefore, the average total processing and detection time of our method for each apk is approximately 12-14 seconds, and the primary time consumption occurs in the feature extraction phase, a commonality shared with other baseline methods. Regardless of family mode or package mode, the MamaDroid method has the longest runtime. Drebin takes 4.53 seconds in single-process mode, and in multi-process mode, the runtime can be reduced to less than 1 second. Our method is quite comparable to DroidEvolver in terms of time consumption. Overall, our method can be considered lightweight.

5. Discussion

In this section, we address the limitations of our current approach and outline potential strategies for mitigation:

5.1. Extensibility of GSEDroid

GSEDroid offers scalability for future enhancements. We plan to augment the call graph with additional API-related features such as API call frequency, functional classification of APIs, security ratings or risk labels from the security community, and dynamic behavioral features during runtime. These additions can enrich the contextual understanding of the APIs.

5.2. Model and graph size optimization

Future work includes employing knowledge distillation techniques to reduce the size of the pre-trained model and graph compression methods to minimize the size of the API call graph. This approach aims to optimize our time and space overhead, making the system more efficient.

5.3. Code packing in malware

Some malicious software employs code packing techniques to prevent analysts from accessing the application bytecode. Our method currently requires complete DEX code to build semantic embedding features of APIs. Therefore, an efficient code unpacking system will be necessary for preprocessing DEX file extraction and heterogeneous code analysis. Solutions like DexHunter (Zhang et al., 2015) and AppSpear (Li et al., 2018) offer promising directions in addressing this challenge.

5.4. Graph-based adversarial techniques

Enriching our method with specific graph-based adversarial techniques (Demontis et al., 2019; Li et al., 2023; Zhao et al., 2021a) is crucial. Future considerations include adopting adversarial training, injecting adversarial samples into the model, and marking them as threats, to enhance robustness against adversarial attacks.

5.5. Realistic ratio of benign to malicious software

Pendlebury et al. (2019) highlight that unrealistic assumptions about the ratio of benign to malicious software can lead to biased performance evaluations. In reality, the number of benign applications significantly surpasses that of malicious ones, with the latter ideally representing less than 23% of the total. Future research will consider this practical scenario for a more realistic ratio setting in our dataset, ensuring more accurate performance assessments.

6. Conclusion

In this paper, the combination of CodeBERT and TextCNN constructs a more powerful, flexible, and generalizable lightweight embedding model, significantly enhancing the feature extraction effectiveness of Dalvik opcode sequences. This amalgamation effectively harnesses the strengths of both models: the rich contextual prior knowledge and long-distance semantic dependencies from CodeBERT, along with the local feature extraction capability of TextCNN, provide a more comprehensive and accurate feature representation. Building upon this, we introduce GSEDroid, an Android malware detection framework that constructs API call graphs incorporating both permission features and opcode semantic features, utilizing the GraphSage+SAGPooling model. Through a series of evaluative analyses on factors influencing model performance and extensive comparative experiments on model effectiveness, our results validate the efficacy of our approach and lay a solid foundation for future research in the domain.

CRediT authorship contribution statement

Jintao Gu: Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Software, Writing – original draft, Writing – review & editing. **Hongliang Zhu:** Conceptualization, Funding acquisition, Methodology, Project administration, Resources, Supervision, Validation. **Zewei Han:** Data curation, Methodology, Software, Validation. **Xiangyu Li:** Software, Validation, Visualization. **Jianjin Zhao:** Conceptualization, Methodology, Validation, Writing – review & editing.

Declaration of competing interest

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

Data availability

The data that has been used is confidential.

Acknowledgements

This work was supported in part by the National Natural Science Foundation of China under Grants 62172055 and U20B2045; and in part by the BUPT Scientific and Technological Innovation Capacity Improvement Project under Grant 2021XD-A09.

References

Allix, K., Bissyandé, T.F., Klein, J., Le Traon, Y., 2016. AndroZoo: collecting millions of Android apps for the research community. In: Proceedings of the 13th International Conference on Mining Software Repositories, pp. 468–471.

Alzaylaee, M.K., Yerima, S.Y., Sezer, S., 2020. DL-Droid: deep learning based Android malware detection using real devices. Comput. Secur. 89, 101663. https://doi.org/10.1016/j.cose.2019.101663.

Anderson, B., Quist, D., Neil, J., Storlie, C., Lane, T., 2011. Graph-based malware detection using dynamic analysis. J. Comput. Virol. 7, 247–258. https://doi.org/10.1007/s11416-011-0152-x.

Arora, A., Peddoju, S.K., Conti, M., 2020. PermPair: Android malware detection using permission pairs. IEEE Trans. Inf. Forensics Secur. 15, 1968–1982. https://doi.org/10.1109/TIFS.2019.2950134.

Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., 2014, Siemens, CERT. Drebin: effective and explainable detection of Android malware in your pocket. In: NDSS, pp. 23–26.

Badhani, S., Muttoo, S.K., 2019. CENDroid—a cluster-ensemble classifier for detecting malicious Android applications. Comput. Secur. 85, 25–40. https://doi.org/10.1016/j.cose.2019.04.004.

Bhat, P., Behal, S., Dutta, K., 2023. A system call-based Android malware detection approach with homogeneous & heterogeneous ensemble machine learning. Comput. Secur. 130, 103277. https://doi.org/10.1016/j.cose.2023.103277.

Bilar, D., 2007. Opcodes as predictor for malware. Int. J. Electron. Secur. Digit. Forensics 1, 156–168. https://doi.org/10.1504/IJESDF.2007.016865.

Cai, H., Meng, N., Ryder, B., Yao, D., 2019. DroidCat: effective Android malware detection and categorization via app-level profiling. IEEE Trans. Inf. Forensics Secur. 14, 1455–1470. https://doi.org/10.1109/TIFS.2018.2879302.

Cai, L., Li, Y., Xiong, Z., 2021. JOWMDroid: Android malware detection based on feature weighting with joint optimization of weight-mapping and classifier parameters. Comput. Secur. 100, 102086. https://doi.org/10.1016/j.cose.2020.102086.

Chen, Y., Ding, Z., Wagner, D., 2023. Continuous learning for Android malware detection. In: 32nd USENIX Security Symposium (USENIX Security), vol. 23, pp. 1127–1144.

Crammer, K., Kulesza, A., Dredze, M., 2009. Adaptive regularization of weight vectors. In: Advances in Neural Information Processing Systems, vol. 22. https://proceedings.neurips.cc/paper_files/paper/2009/hash/8ebda540cbcc4d7336496819a46a1b68-Abstract.html.

CyberDefence General Dynamics, 2023. G DATA Mobile Security Report: Attacks on Smartphones Every Minute. Technical Report URL https://presse.gdata.de/News_Detail.aspx?id = 174612&menueid = 28982&l = english.

Defferrard, M., Bresson, X., Vandergheynst, P., 2016. Convolutional neural networks on graphs with fast localized spectral filtering. In: Advances in Neural Information Processing Systems, vol. 29. URL https://proceedings.neurips.cc/paper_files/paper/2016/hash/04df4d434d481c5bb723be1b6df1ee65-Abstract.html.

Demontis, A., Melis, M., Biggio, B., Maiorca, D., Arp, D., Rieck, K., Corona, I., Giacinto, G., Roli, F., 2019. Yes, machine learning can be more secure! A case study on Android malware detection. IEEE Trans. Dependable Secure Comput. 16, 711–724. https://doi.org/10.1109/TDSC.2017.2700270.

Desnos, A., Gueguen, G., 2012. Androguard. URL https://androguard.readthedocs.io/en/latest/tools/androdd.html.

Devlin, J., Chang, M.W., Lee, K., Toutanova, K., 2019. BERT: pre-training of deep bidirectional transformers for language understanding. https://doi.org/10.48550/arXiv.1810.04805. arXiv:1810.04805, 2019.

Duchi, J., Hazan, E., Singer, Y., 2011. Adaptive subgradient methods for online learning and stochastic optimization. J. Mach. Learn. Res. 12. https://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf.

Fang, W., He, J., Li, W., Lan, X., Chen, Y., Li, T., Huang, J., Zhang, L., 2023. Comprehensive Android malware detection based on federated learning architecture. IEEE Trans. Inf. Forensics Secur. 18, 3977–3990. https://doi.org/10.1109/TIFS.2023.3287395.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., Zhou, M., 2020. CodeBERT: a pre-trained model for programming and natural languages. https://doi.org/10.48550/arXiv.2002.08155. arXiv:2002.08155, 2020.

Frenklach, T., Cohen, D., Shabtai, A., Puzis, R., 2021. Android malware detection via an app similarity graph. Comput. Secur. 109, 102386. https://doi.org/10.1016/j.cose.2021.102386.

Gao, H., Cheng, S., Zhang, W., 2021. GDroid: Android malware detection and classification with graph convolutional network. Comput. Secur. 106, 102264. https://doi.org/10.1016/j.cose.2021.102264.

GlobalStats, S., 2023. Operating System Market Share Worldwide. Technical Report. URL https://gs.statcounter.com/os-market-share.

Guerra-Manzanares, A., Bahsi, H., 2022. On the relativity of time: implications and challenges of data drift on long-term effective Android malware detection. Comput. Secur. 122, 102835. https://doi.org/10.1016/j.cose.2022.102835.

Guerra-Manzanares, A., Luckner, M., Bahsi, H., 2022. Concept drift and cross-device behavior: challenges and implications for effective Android malware detection. Comput. Secur. 120, 102757. https://doi.org/10.1016/j.cose.2022.102757.

Hamilton, W., Ying, Z., Leskovec, J., 2017. Inductive representation learning on large graphs. Adv. Neural Inf. Process. Syst. 30. https://proceedings.neurips.cc/paper/6703-inductive-representation-learning-on-large-graphs.

Han, Q., Subrahmanian, V.S., Xiong, Y., 2020. Android malware detection via (somewhat) robust irreversible feature transformations. IEEE Trans. Inf. Forensics Secur. 15, 3511–3525. https://doi.org/10.1109/TIFS.2020.2975932.

Hashemi, H., Azmoodeh, A., Hamzeh, A., Hashemi, S., 2017. Graph embedding as a new approach for unknown malware detection. J. Comput. Virol. Hacking Tech. 13, 153–166. https://doi.org/10.1007/s11416-016-0278-y.

He, Y., Li, Y., Wu, L., Yang, Z., Ren, K., Qin, Z., 2023. MsDroid: identifying malicious snippets for Android malware detection. IEEE Trans. Dependable Secure Comput. 20, 2025–2039. https://doi.org/10.1109/TDSC.2022.3168285.

Jeon, S., Moon, J., 2020. Malware-detection method with a convolutional recurrent neural network using opcode sequences. Inf. Sci. 535, 1–15. https://doi.org/10.1016/j.ins.2020.05.026.

Jerbi, M., Chelly Dagdia, Z., Bechikh, S., Ben Said, L., 2022. Android malware detection as a Bi-level problem. Comput. Secur. 121, 102825. https://doi.org/10.1016/j.cose.2022.102825.

Jerbi, M., Dagdia, Z.C., Bechikh, S., Said, L.B., 2020. On the use of artificial malicious patterns for Android malware detection. Comput. Secur. 92, 101743. https://doi.org/10.1016/j.cose.2020.101743.

Jiang, J., Li, S., Yu, M., Li, G., Liu, C., Chen, K., Liu, H., Huang, W., 2019. Android malware family classification based on sensitive opcode sequence. In: 2019 IEEE Symposium on Computers and Communications (ISCC), pp. 1–7.

Kang, J., Jang, S., Li, S., Jeong, Y.S., Sung, Y., 2019. Long short-term memory-based malware classification method for information security. Comput. Electr. Eng. 77, 366–375. https://doi.org/10.1016/j.compeleceng.2019.06.014.

Khalilian, A., Nourazar, A., Vahidi-Asl, M., Haghighi, H., 2018. G3MD: mining frequent opcode sub-graphs for metamorphic malware detection of existing families. Expert Syst. Appl. 112, 15–33. https://doi.org/10.1016/j.eswa.2018.06.012.

Khan, K.N., Ullah, N., Ali, S., Khan, M.S., Nauman, M., Ghani, A., 2021. OP2VEC: an opcode embedding technique and dataset design for end-to-end detection of Android malware. Secur. Commun. Netw. 2022. https://doi.org/10.1155/2022/3710968.

Kim, T., Kang, B., Rho, M., Sezer, S., Im, E.G., 2019. A multimodal deep learning method for Android malware detection using various features. IEEE Trans. Inf. Forensics Secur. 14, 773–788. https://doi.org/10.1109/TIFS.2018.2866319.

Kim, Y., 2014. Convolutional neural networks for sentence classification. https://doi.org/10.48550/arXiv.1408.5882. arXiv:1408.5882, 2014.

Kipf, T.N., Welling, M., 2017. Semi-supervised classification with graph convolutional networks. https://doi.org/10.48550/arXiv.1609.02907. arXiv:1609.02907, 2017.

Kong, K., Zhang, Z., Yang, Z.Y., Zhang, Z., 2022. FCSCNN: feature centralized Siamese CNN-based Android malware identification. Comput. Secur. 112, 102514. https://doi.org/10.1016/j.cose.2021.102514.

LeCun, Y., Bengio, Y., Hinton, G., 2015. Deep learning. Nature 521, 436–444. https://doi.org/10.1038/nature14539.

Lee, J., Lee, I., Kang, J., 2019. Self-attention graph pooling. In: International Conference on Machine Learning, pp. 3734–3743. https://proceedings.mlr.press/v97/lee19c.html.

Li, B., Zhang, Y., Li, J., Yang, W., Gu, D., 2018. AppSpear: automating the hidden-code extraction and reassembling of packed Android malware. J. Syst. Softw. 140, 3–16. https://doi.org/10.1016/j.jss.2018.02.040.

Li, H., Cheng, Z., Wu, B., Yuan, L., Gao, C., Yuan, W., Luo, X., 2023. Black-box adversarial example attack towards FCG based Android malware detection under incomplete feature information. In: 32nd USENIX Security Symposium (USENIX Security), vol. 23, pp. 1181–1198.

Li, Y., Tarlow, D., Brockschmidt, M., Zemel, R., 2017. Gated graph sequence neural networks. http://arxiv.org/abs/1511.05493. arXiv:1511.05493, 2017.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., Stoyanov, V., 2019. RoBERTa: a robustly optimized BERT pretraining approach. URL http://arxiv.org/abs/1907.11692. arXiv:1907.11692, 2019.

Mariconti, E., Onwuzurike, L., Andriotis, P., De Cristofaro, E., Ross, G., Stringhini, G., 2017. MaMaDroid: detecting Android malware by building Markov chains of behavioral models. https://doi.org/10.48550/arXiv.1612.04433. arXiv:1612.04433, 2017.

Meng, Z., Xiong, Y., Huang, W., Miao, F., Huang, J., 2021. AppAngio: revealing contextual information of Android app behaviors by API-level audit logs. IEEE Trans. Inf. Forensics Secur. 16, 1912–1927. https://doi.org/10.1109/TIFS.2020.3044867.

Moskovitch, R., Feher, C., Tzachar, N., Berger, E., Gitelman, M., Dolev, S., Elovici, Y., 2008. Unknown malcode detection using OPCODE representation. Intell. Secur. Inform. 5376, 204–215. https://doi.org/10.1007/978-3-540-89900-6_21.

National Internet Emergency Center of China, 2020. Overview of Internet Network Security Situation in China in 2020. Technical Report. https://www.cert.org.cn/publish/main/46/2021/20210526121148344277777/20210526121148344277777_.html.

Navarro, L.C., Navarro, A.K., Grégio, A., Rocha, A., Dahab, R., 2018. Leveraging ontologies and machine-learning techniques for malware analysis into Android permissions ecosystems. Comput. Secur. 78, 429–453. https://doi.org/10.1016/j.cose.2018.07.013.

Ou, F., Xu, J., 2022. S3Feature: a static sensitive subgraph-based feature for Android malware detection. Comput. Secur. 112, 102513. https://doi.org/10.1016/j.cose.2021.102513.

Pektaş, A., Acarman, T., 2020. Learning to detect Android malware via opcode sequences. Neurocomputing 396, 599–608. https://doi.org/10.1016/j.neucom.2018.09.102.

Pendlebury, F., Pierazzi, F., Jordaney, R., Kinder, J., Cavallaro, L., 2019. TESSERACT: eliminating experimental bias in malware classification across space and time. In: 28th USENIX Security Symposium (USENIX Security), vol. 19, pp. 729–746. https://www.usenix.org/conference/usenixsecurity19/presentation/pendlebury.

Qiu, J., Yang, X., Wu, H., Zhou, Y., Li, J., Ma, J., 2022. LibCapsule: complete confinement of third-party libraries in Android applications. IEEE Trans. Dependable Secure Comput. 19, 2873–2889. https://doi.org/10.1109/TDSC.2021.3075817.

Rastogi, V., Chen, Y., Jiang, X., 2013. DroidChameleon: evaluating Android anti-malware against transformation attacks. In: Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security, pp. 329–334.

Runwal, N., Low, R.M., Stamp, M., 2012. Opcode graph similarity and metamorphic detection. J. Comput. Virol. 8, 37–52. https://doi.org/10.1007/s11416-012-0160-5.

Saracino, A., Sgandurra, D., Dini, G., Martinelli, F., 2018. MADAM: effective and efficient behavior-based Android malware detection and prevention. IEEE Trans. Dependable Secure Comput. 15, 83–97. https://doi.org/10.1109/TDSC.2016.2536605.

Sistemas, 2004. VirusTotal. URL. https://www.virustotal.com/.

Tarwireyi, P., Terzoli, A., Adigun, M.O., 2023. Using multi-audio feature fusion for Android malware detection. Comput. Secur. 131, 103282. https://doi.org/10.1016/j.cose.2023.103282.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., Bengio, Y., 2018. Graph attention networks. https://doi.org/10.48550/arXiv.1710.10903. arXiv:1710.10903, 2018.

Vinyals, O., Bengio, S., Kudlur, M., 2016. Order matters: sequence to sequence for sets. https://doi.org/10.48550/arXiv.1511.06391. arXiv:1511.06391, 2016.

Wang, S., Yan, Q., Chen, Z., Yang, B., Zhao, C., Conti, M., 2018. Detecting Android malware leveraging text semantics of network flows. IEEE Trans. Inf. Forensics Secur. 13, 1096–1109. https://doi.org/10.1109/TIFS.2017.2771228.

Wong, M.Y., Lie, D., 2018. Tackling runtime-based obfuscation in Android with TIRO. In: 27th USENIX Security Symposium (USENIX Security), vol. 18, pp. 1247–1262. https://www.usenix.org/conference/usenixsecurity18/presentation/wong.

Wu, Y., Li, M., Zeng, Q., Yang, T., Wang, J., Fang, Z., Cheng, L., 2023. DroidRL: feature selection for Android malware detection with reinforcement learning. Comput. Secur. 128, 103126. https://doi.org/10.1016/j.cose.2023.103126.

Xu, J., Li, Y., Deng, R.H., Xu, K., 2022. SDAC: a slow-aging solution for Android malware detection using semantic distance based API clustering. IEEE Trans. Dependable Secure Comput. 19, 1149–1163. https://doi.org/10.1109/TDSC.2020.3005088.

Xu, K., Hu, W., Leskovec, J., Jegelka, S., 2019a. How powerful are graph neural networks?. https://doi.org/10.48550/arXiv.1810.00826. arXiv:1810.00826, 2019.

Xu, K., Li, Y., Deng, R., Chen, K., Xu, J., 2019b. DroidEvolver: self-evolving Android malware detection system. In: 2019 IEEE European Symposium on Security and Privacy (EuroS&P), pp. 47–62.

Yadav, P., Menon, N., Ravi, V., Vishvanathan, S., Pham, T.D., 2022. EfficientNet convolutional neural networks-based Android malware detection. Comput. Secur. 115, 102622. https://doi.org/10.1016/j.cose.2022.102622.

Yewale, A., Singh, M., 2016. Malware detection based on opcode frequency. In: 2016 International Conference on Advanced Communication Control and Computing Technologies (ICACCCT), pp. 646–649.

Yuxin, D., Siyi, Z., 2019. Malware detection based on deep learning algorithm. Neural Comput. Appl. 31, 461–472. https://doi.org/10.1007/s00521-017-3077-6.

Zhang, J., Qin, Z., Zhang, K., Yin, H., Zou, J., 2018. Dalvik opcode graph based Android malware variants detection using global topology features. IEEE Access 6, 51964–51974. https://doi.org/10.1109/ACCESS.2018.2870534.

Zhang, X., Zhang, Y., Zhong, M., Ding, D., Cao, Y., Zhang, Y., Zhang, M., Yang, M., 2020. Enhancing state-of-the-art classifiers with API semantics to detect evolved Android malware. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pp. 757–770.

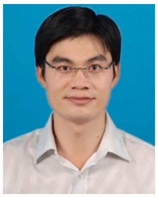
Zhang, Y., Luo, X., Yin, H., 2015. DexHunter: toward extracting hidden code from packed Android applications. In: Computer Security – ESORICS 2015, vol. 9327, pp. 293–311.

Zhang, Y., Wallace, B., 2016. A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification. https://doi.org/10.48550/arXiv.1510.03820. arXiv:1510.03820, 2016.

Zhao, K., Zhou, H., Zhu, Y., Zhan, X., Zhou, K., Li, J., Yu, L., Yuan, W., Luo, X., 2021a. Structural attack against graph based Android malware detection. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, pp. 3218–3235.

Zhao, Y., Li, L., Wang, H., Cai, H., Bissyandé, T.F., Klein, J., Grundy, J., 2021b. On the impact of sample duplication in machine-learning-based Android malware detection. ACM Trans. Softw. Eng. Methodol. 30, 1–38. https://doi.org/10.1145/3446905.

Jintao Gu graduated from Beijing University of Posts and Telecommunications in 2022 with a bachelor's degree in information security. He is currently studying for a PhD at the School of Cyberspace Security, Beijing University of Posts and Telecommunications. His research interests include software security, Android malware analysis, deep learning, and artificial intelligence.



Hongliang Zhu received the Ph. D. degree in information security from Beijing University of Posts and Telecommunications. He is an associate professor in School of Cyberspace Security, Beijing University of Posts and Telecommunications, meanwhile, serves as vice director for Beijing Engineering Lab for Cloud Security. His current research interests include network security and software security.

Zewei Han graduated from Beijing University of Posts and Telecommunications in 2022 with a bachelor's degree in engineering. He is currently studying for a master's degree at Beijing University of Posts and Telecommunications. His research interests include malicious code analysis and encrypted traffic analysis.

Xiangyu Li graduated from Chongqing University in 2021 with a bachelor's degree in information security. He is currently studying for a master's degree at the School of Cyberspace Security, Beijing University of Posts and Telecommunications. His research interests include software security and Android malware analysis.

Jianjin Zhao received B.Eng. degree from Beijing University of Posts and Telecommunications, China, in 2019. He is currently pursuing the Ph.D. degree in Cyberspace Security at Beijing University of Posts and Telecommunications, China. His current research interests include encrypted traffic analysis an audit log analysis.