Sensitive Behavioral Chain-Focused Android Malware Detection Fused With AST Semantics

Jiacheng Gong, *Graduate Student Member, IEEE*, Weina Niu, *Senior Member, IEEE*, Song Li, *Member, IEEE*, Mingxue Zhang, and Xiaosong Zhang

***Abstract*— The proliferation of Android malware poses a substantial security threat to mobile devices. Thus, achieving efficient and accurate malware detection and malware family identification is crucial for safeguarding users' individual property and privacy. Graph-based approaches have demonstrated remarkable detection performance in the realm of intelligent Android malware detection methods. This is attributed to the robust representation capabilities of graphs and the rich semantic information. The function call graph (FCG) is the most widely used graph in intelligent Android malware detection. However, existing FCG-based malware detection methods face challenges, such as the enormous computational and storage costs of modeling large graphs. Additionally, the ignorance of code semantics also makes them susceptible to structured attacks. In this paper, we proposed AndroAnalyzer, which embeds abstract syntax tree (AST) code semantics while focusing on sensitive behavior chains. It leverages FCGs to represent the macroscopic behavior of the application, and employs structured code semantics to represent the microscopic behavior of functions. Furthermore, we proposed the sensitive function call graph (SFCG) generation algorithm to narrow down the analysis scope to sensitive function calls, and the AST vectorization algorithm (AST2Vec) to capture structured code semantics. Experimental results demonstrate that the proposed SFCG generation algorithm noticeably reduces graph size while ensuring robust detection performance. AndroAnalyzer outperforms the baseline methods in binary and multiclass classification tasks, achieving F1-scores of 99.21% and 98.45% respectively. Moreover, AndroAnalyzer (trained with samples of 2010-2018) exhibits good generalization capabilities in detecting samples of 2019-2022.**

***Index Terms*— Android malware detection, function call graph, abstract syntax tree, code semantic embedding, graph neural networks.**

Received 23 October 2023; revised 24 April 2024; accepted 20 September 2024. Date of publication 26 September 2024; date of current version 7 October 2024. This work was supported in part by the National Science Foundation of China under Grant 62372086, in part by Chongqing Natural Science Foundation Innovation and Development Joint Foundation under Grant CSTB2023NSCQ-LZX0003, and in part by Sichuan Natural Science Foundation under Grant 24ZNSFSC0038. The associate editor coordinating the review of this article and approving it for publication was Dr. Aaron Visaggio. *(Corresponding author: Weina Niu.)*

Jiacheng Gong is with the School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu 611731, China (e-mail: gongjc.uestc@gmail.com).

Weina Niu and Xiaosong Zhang are with the Institute for Advanced Study, University of Electronic Science and Technology of China, Shenzhen 518110, China, and also with the School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu 611731, China (e-mail: vinusniu@uestc.edu.cn; johnsonzxs@uestc.edu.cn).

Song Li and Mingxue Zhang are with the State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou 310058, China (e-mail: songl@zju.edu.cn; mxzhang97@zju.edu.cn).

Digital Object Identifier 10.1109/TIFS.2024.3468891

I. Introduction

THE development of Internet of Things (IoT) has led to continuous implementation of the digital living concept, and widespread adoption of mobile devices. Concurrently, Android OS, the dominant operating system for mobile devices, is facing remarkable challenges posed by massive Android malware. According to reports by Kaspersky [1], in the first quarter of 2023, 307,529 malicious installation packages were detected. These malware strains often infiltrate user devices covertly, aiming to steal sensitive information or gain control over the device, posing a severe threat to users' financial assets and privacy.

In response to the security threats posed by Android malware, and considering the substantial costs associated with manual software analysis, numerous intelligent malware detection methods have been proposed [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20]. These methods are designed to identify a vast number of malicious software instances efficiently.

In recent years, graph-based Android malware detection methods have garnered notable attention [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15]. This is because the graphs can capture complex relationships between different components of malware, providing a multi-level information representation of malware. It enables the analysis of associations between different malware instances. Among these methods, function call graphs (FCGs), as a structural representation of software, can capture the call relationships between functions, and allows the discovery of potential malicious behavior patterns. Therefore, FCGs are most widely used in intelligent Android malware detection. Aiming to model application behavior and optimize modeling costs, we focus on FCGs over graphs about feature relations or more detailed graphs, such as control flow graphs. However, in existing FCG-based methods, the following issues have been identified:

(1) Fine-grained modeling and large-scale analysis. Existing FCG-based malware detection methods can be broadly categorized into three types.

- The first category of methods generate feature vectors based on the usage or frequency of the API calls in the FCGs. These vectors are subsequently utilized to perform malware detection [5]. These methods heavily rely on the expertise of the designer and can be influenced by subjective factors.

- The second category of methods utilize FCGs as input to the graph neural networks (GNNs) and employ structural or frequency features of nodes for classification [9], [10], [11]. However, the FCGs can be quite large, leading to considerable computational and storage overhead for subsequent AI model training. For example, consider a sample with an MD5 hash of 88ddf2594600f4b570478fa92a7050a0; its APK file size is 30.18MB, and the extracted Dex files have a size of 52.42MB. The FCG generated by Androguard [21] contains 356,687 nodes and 1,700,696 edges. In mainstream app markets, many popular apps have sizes exceeding 50MB, with game apps often exceeding 1GB. The computational and storage costs of analyzing and learning from the complete FCG using AI models can be substantial in such cases.
- The third category of methods, such as those proposed in [4], [22], and [8], utilize features related to sensitive API calls within the FCGs as application representations. These methods overlook the contextual information of function calls, specifically the information within the call chains of sensitive APIs.

(2) Ignoring structured code semantics. In existing graph-based detection methods, API semantics extraction can be broadly categorized into the following types.

- The first category of work [4], [5], and [11] uses only node-related statistical or structural features, ignoring code semantics. This makes their classification results susceptible graph structural attacks [23].
- The second category of work uses One-hot encoding to represent API usage patterns. However, the encoding vector is highly sparse and does not contain function code semantics information.
- The third category of work [9] and [7] employs natural language processing techniques to learn the code semantics. However, source code and binary files are more structured and logically organized than natural language. Abstract syntax trees (ASTs), control flow graphs, and data flow graphs are more suitable for representing structured code semantics.

To address the abovementioned issues, we proposed an Android malware detection method called AndroAnalyzer, which focuses on sensitive behavior chains to reduce the computational and storage overhead. It also incorporates structured code semantics in the chains to be resistent to structural attacks. Specifically, the sensitive behavior chain is a chain of function calls that are closely associated with sensitive behaviors. All the sensitive behavior chains are merged to construct the sensitive function call graph (SFCG). AndroAnalyzer utilizes the SFCG to characterize the macroscopic behavior of the application, providing a view of the execution flow of malicious behaviors. Next, AndroAnalyzer extracts structural code semantics via ASTs to represent the micro-behavior of each function, delving into the code logic within the functions. Furthermore, by incorporating AST code semantics features generated by the proposed AST2Vec algorithm and combining them with API semantics features and structural information features obtained from social network analysis, we can obtain a SFCG with fused node features. Finally, this graph is input into a GNN with graph self-attention pooling for learning, resulting in an intelligent classifier for Android malware detection.

In summary, the major contributions of this work include:

- We proposed an effective method for representing the behavior of Android applications. It uses FCGs to represent the macroscopic behavior of applications and structured code semantics to represent the microscopic behavior of functions. This approach strikes a balance between modeling granularity and storage cost. Additionally, we designed a SFCG generation algorithm to reduce the graph size and focus on sensitive behavior chains that are related to malicious behaviors. This effectively reduces the computational overhead when analyzing complex APK files.
- We proposed a structured code semantics extraction algorithm called AST2Vec, based on ASTs. This algorithm effectively extracts structured code semantics from smali code, providing comprehensive behavioral information for Android malware detection. Furthermore, the classification model exhibits improved generalization and robustness in binary and multi-class classification by integrating API semantics features and structural features obtained from social network analysis.
- We conducted extensive performance evaluation experiments on two datasets constructed from CICMalDroid [24] and AndroZoo [25]. The experimental results demonstrate that AndroAnalyzer outperforms the baseline methods in binary and multi-class classification tasks. Furthermore, it (trained with samples in 2010-2018) exhibits good generalization ability in the detection of samples in 2019-2022.

The remaining sections are organized as follows. Section II provides an overview of graph-based malicious software detection works. In Section III, we introduce the design of AndroAnalyzer, and in Section IV, we present the evaluation results. Finally, we discuss briefly in Section V and offer concluding remarks and future research directions in Section VI.

II. Related Work

A. Detection Methods Based on Graph Analysis

This category of methods models static or dynamic features of Android applications using graph. Subsequently, it employs graph matching or graph feature extraction in conjunction with machine learning techniques to perform detection.

In 2019, Arora et al. proposed PermPair [2], which models the usage patterns of permission pairs (pairing of two dangerous permissions used simultaneously to perform malicious or benign behavior) in applications using a permission graph. During detection, the method calculates benign and malicious weight scores based on an app's usage of permission pairs and compares these scores to detect malwares. Similarly, Fan et al. introduced GefDroid [3] in 2019, which extracts API usage patterns by analyzing the structural features of sensitive APIs within subgraphs corresponding to code classes in apps. This approach analyzes graph similarity between applications and performs unsupervised clustering of malicious app families, incorporating community detection techniques. Wu et al. also employed social network analysis to analyze FCGs and introduced MalScan [4]. This method analyzes the centrality of sensitive API calls in FCGs to generate feature vectors. It relies on the centrality distributions of sensitive API calls for classification. MaMaDroid [5] models API call sequences (or abstracted sequences at class or package levels) in function call graphs as Markov chains for detection and classification.

In 2020, Surendran et al. introduced Gsdroid [6], which represents the behavior of applications using system call graphs. They normalize the call frequency of system calls as their proposed graph signals and combine these graph signal features with machine learning techniques for classification. Also in 2020, Niu et al. [7] extracted API call sequences from FCGs, followed by further extraction of opcodes associated with API calls. Finally, they employed LSTM to train and learn from opcode-level call sequences.

In 2021, Wu et al. introduced HomDroid [8]. It begins with community detection on FCGs and employs homogeneity analysis to identify the most suspicious subgraphs. It then generates features from the sensitive APIs' occurrences, quantities, and proportions in the subgraphs. Machine learning is subsequently applied to learn from the feature vectors.

These methods often break the feature correlations during the feature vectorization. Thus, we employ graphs and GNNs to represent and learn these features respectively.

B. Behavioral Analysis Detection Methods Based on GNNs

This category of methods models behaviors of applications using graph. They utilize GNNs or graph embedding to learn the topological and node features of graphs, and they typically belong to the task of graph classification.

In 2021, Xu et al. [9] proposed an Android malware detection method based on FCG embedding. They used Word2Vec to vectorize opcodes and combined it with the SIF network for function embedding. This embedding was used as node features in the FCG to generate graph embeddings through Struct2Vec, ultimately leading to malware detection based on these embeddings. Similarly, Cai et al. [10] used API call sequences as a corpus to obtain function embeddings using natural language processing techniques. They fed the FCGs with these embeddings to GNNs to perform classification.

In 2022, Yumlembam et al. [11] utilized GNNs to generate API graph embeddings based on centrality measures. They combined these embeddings with permissions and intents for malware detection.

In 2023, Wu et al. introduced DeepCatra [12], which tracks the call traces of key APIs in FCGs. It considers relationships such as intent and ICC (Inter-Component Communication) and connects edges accordingly. DeepCatra employs a Bi-LSTM to learn call traces and utilizes GNNs to learn abstract flow graphs, combining information for detection. In the same year, Wu et al. [13] presented another approach in which they encoded opcodes in functions using one-hot encoding. They calculated node importance based on centrality and weighted APIs based on their protection levels corresponding to permissions. The construction of the graph treated the FCG as an undirected graph. They used a breadth-first algorithm to create sensitive function subgraphs of sensitive APIs and their neighbors within a two-hop distance. They combined API features with graph structure and employed GNNs for malware detection. Shi et al. proposed SFCGDroid [14], where they used API call sequences as a corpus to obtain function semantics using the Skip-gram method. They also incorporated social network triple information of sensitive APIs as function node features and combined them with FCG structures for malware detection. Addressing the remarkable challenge in graph-based Android malware detection methods known as graph structural attacks, Li et al. introduced RGDroid [15]. This method initially generates embeddings of API entities based on an API relationship graph derived from official Android documentation. These embeddings are used as node features in FCGs. Additionally, RGDroid employs community detection to partition FCGs into functional subgraphs, reducing redundant edge connections and mitigating the impact of graph structural attacks. Finally, it uses Graph Neural Networks to learn and detect function call subgraphs.

Most of these methods face the issues mentioned in Section I. Therefore, we integrate SFCGs and AST code semantics for malware detection.

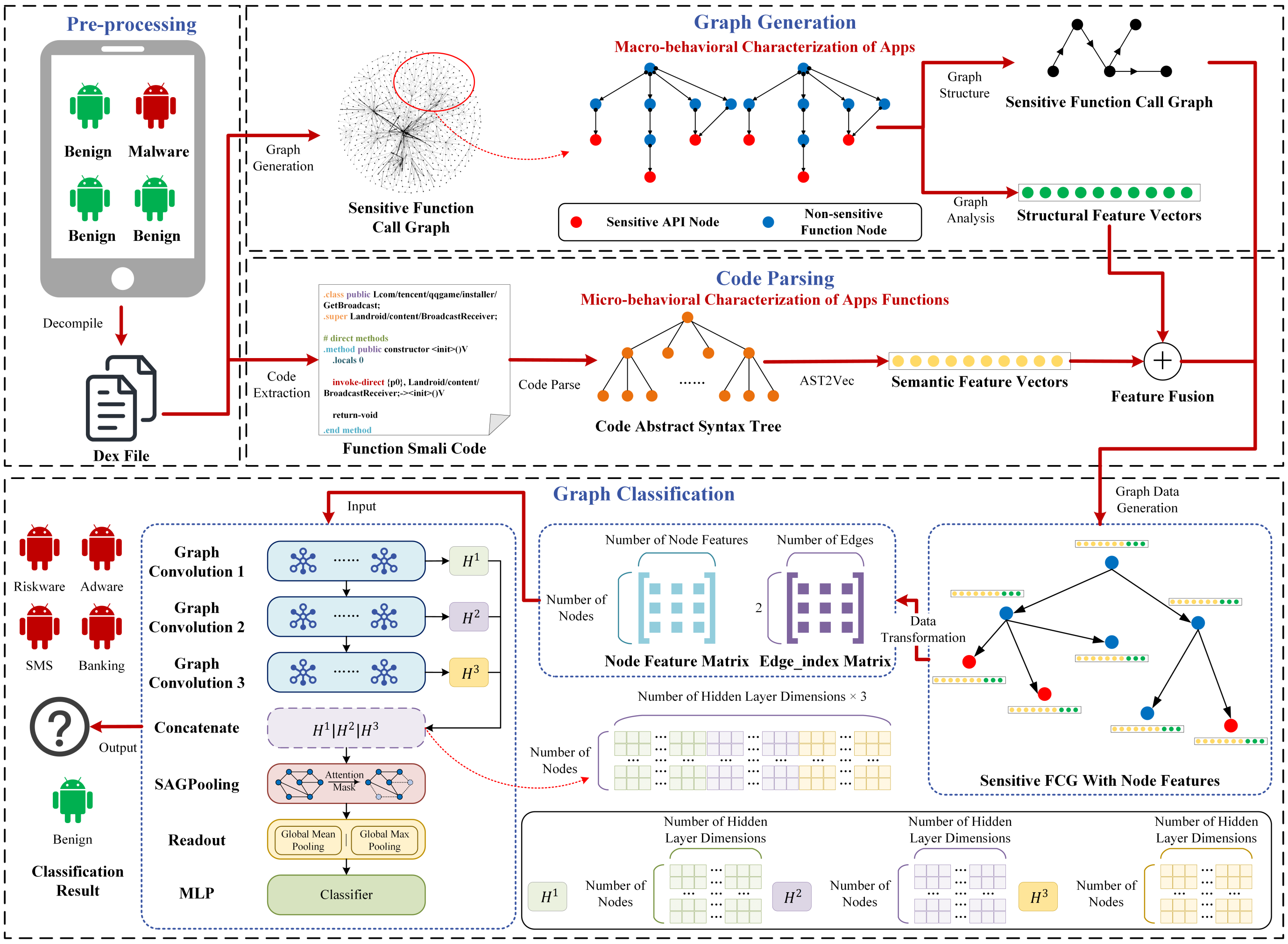
C. Association Analysis Detection Methods Based on GNNs

This category of methods models the relationships between applications using graph. They employed GNNs or graph embedding to learn the topological (relationship) features and node (application) features, making this category suitable for node classification tasks.

In 2021, Gao et al. introduced GDroid [16], which transforms the problem of malware detection into a graph node classification task. It constructs edges between applications (APPs) and APIs based on the call relationships among APIs and the patterns of API usage. This maps APPs and APIs to a large heterogeneous graph and employs Graph Convolutional Neural Networks (GCNs) to detect malware. Hei et al. presented HAWK [17] in the same year. This method builds a heterogeneous graph by considering more entities such as APIs, permissions, permission types, classes, interfaces, and shared object (so) files. It utilizes a heterogeneous graph attention network to learn relationships under different meta-paths for the final detection. Fan et al. developed a method [18] that constructs a heterogeneous graph using entities like applications, app markets, publishing companies, app names, app signatures, and developers. It also incorporates information from different versions of the heterogeneous graph and performs learning and detection based on spatiotemporal heterogeneous graph information of applications.

In 2023, Huang et al. introduced WHGDroid [19], which also builds a heterogeneous graph using multiple entities and learn relationships through meta-paths. Additionally, WHGDroid incorporates features to mitigate the impact of malware evolution and computes weights based on entity importance.

These methods emphasize inter-app relationships. However, we focus on analyzing individual app behaviors to perform malware detection.



```
.class public Lcom/tencent/qqgame/installer/
GetBroadcast;
.super Landroid/content/BroadcastReceiver;

# direct methods
.method public constructor <init>()V
    .locals 0

    invoke-direct {p0}, Landroid/content/
BroadcastReceiver;-><init>()V

    return-void
.end method
```

Fig. 1. The framework of AndroAnalyzer.

III. Methodology

To address the issues mentioned in Section I, we proposed an intelligent analysis and detection method for Android malware called AndroAnalyzer. Its framework, as shown in Figure 1, consists of four main stages: pre-processing stage, graph generation stage, code parsing stage, and graph classification stage. In the following sections, we will provide detailed descriptions of each stage.

A. Pre-Processing Stage

In the preprocessing stage, we utilize Androguard [21] to perform reverse engineering on Android APK files. We extract the code from their Dex files by decompiling these files. The extracted code will be used for subsequent graph generation and function code parsing. It's worth noting that AndroAnalyzer does not necessarily rely on function call graphs generated by Androguard [21]. Depending on specific analysis requirements, different analysis tools can be used to obtain function call graphs at various levels of granularity. The choice to use Androguard here primarily considers the convenience of implementing the solution.

Androguard [21] is an widely used Python tool for statically analyzing Android applications. It can handle tasks like analyzing Dex, ODex, APK files, Android's binary XML files, and Android resources. We chose Androguard as our decompilation tool for its accuracy in generating FCGs.

Furthermore, this research did not specifically focus on applications with obfuscation or packing techniques. For applications that have been packed, we can employ methods like those presented in [26], [27], and [28] to unpack them and obtain valid Dex files. After obtaining the valid Dex files, we can continue the analysis using androguard.misc.AnalyzeDex(). This function will still provide us with DalvikVMFormat and Analysis objects, allowing us to proceed with the subsequent analysis steps.

B. Graph Generation Stage

During the graph generation stage, we utilize the function call information extracted from the Dex code files to construct a Sensitive Function Call Graph (SFCG) using a sensitive API-based algorithm. Subsequently, we conduct social network analysis on function nodes within this FCG, to generate corresponding structural feature vectors. By focusing on sensitive APIs, we concentrate on malicious behaviors, thereby reducing the graph's scale and mitigating computational and storage overhead. Simultaneously, we calculate structural feature vectors for each node in the FCG via social network analysis. This compensates for the limited capability of GNNs in learning topological features.

1) Sensitive Function Call Graph Generation: We utilize the FCG to characterize the macro-level behavior of Android applications. However, considering the substantial scale of a complete FCG, it imposes remarkable computational and storage overhead for subsequent processing. Hence, we narrow our focus to sensitive behaviors and initiate the graph construction from sensitive API calls. We then trace their ancestor nodes while disregarding other branches unrelated to sensitive APIs in the FCG.

The definition of the Sensitive Function Call Graph (SFCG) is presented as Definition 1.

Definition 1: Sensitive Function Call Graph (SFCG): A directed graph $SFCG_f = (V, E)$ for one Android application f, where V is a set of function nodes and E is a set of function call relation directed edges. $V = \{V_{internal}, V_{external}\}$, $V_{internal}$ represents the internal nodes in the SFCG, which are nodes corresponding to functions for which the corresponding Smali code can be obtained. $V_{external}$ represents the external nodes in the SFCG, which are nodes corresponding to external APIs for which the corresponding Smali code cannot be obtained. Most external nodes are Android API nodes; among them, APIs strongly associated with malicious behavior or sensitive operations are referred to as sensitive APIs. $E = \{(v_i, v_j)|v_i, v_j \in V, v_i\ calls\ v_j\}$. If f invokes sensitive APIs, V only includes sensitive APIs and their ancestors, otherwise, V includes all nodes of complete FCG.

In selecting sensitive APIs, previous work [29] proposed API-Permission mapping to obtain sensitive API lists. However, this list contains tens of thousands of APIs, which can introduce remarkable computational and storage overhead during subsequent analysis and feature generation. Additionally, APIs in the PScout list may suffer from outdated versions. Therefore, considering these factors, we opted for the Dangerous API list from CamoDroid [30], which is proposed in 2022 and includes a total of 401 sensitive APIs.

Based on the aforementioned sensitive APIs, we have devised an algorithm for generating the SFCG, as described in Algorithm 1. Notes that if an Android application does not invoke any sensitive APIs, its complete FCG will be returned. The core aim of this methodology is to discern and categorize the distinct behaviors of benign versus malicious entities, primarily through the analysis of sensitive API usage patterns. However, regardless of how the list of sensitive APIs is chosen, it is possible for applications not to invoke any of these sensitive APIs. This could be due to updates and iterations of APIs or because the application does not involve any sensitive operations (purely benign software). If applications that do not invoke any sensitive APIs are simply classified as benign software, it may result in false negatives for some malicious software. It is imperative, therefore, to conduct a thorough analysis of such cases to accurately differentiate between benign and malicious behaviors rather than generating an empty graph devoid of nodes and edges.

2) Structural Feature Vectors Generation: Once we obtain the SFCG, we further analyze its structural features via social network analysis. Considering that the FCG is a graph network with community characteristics, and several works [4], [8], [31] have achieved good detection results by performing social network analysis on FCGs, we adopt this design to enrich topological information. These properties are used as node structural features to allow the GNN to learn the global topological structure of the graph better. The specific graph structural properties include degree_centrality, in_degree_centrality, out_degree_centrality, katz_centrality, closeness_centrality, betweenness_centrality, harmonic_centrality, clustering, square_clustering and pagerank. The above structural properties can be obtained via NetworkX [32].

Algorithm 1 SFCG Generation Algorithm

```
Input: Sensitive API list $List_{api}$, Androguard Analysis object $als_f$ for Android application $f$
Output: Final Sensitive Function Call Graph $SFCG_f$
1: Initialization $SFCG_f \leftarrow empty\ directed\ graph$
2: for each $api \in List_{api}$ do
3:   if $als_f.find\_methods(api) = True$ then
4:     $Graph_{temp} \leftarrow empty\ directed\ graph$
5:     Queue $unaccessed\_nodes \leftarrow empty\ queue$
6:     List $accessed\_nodes \leftarrow empty\ list$
7:     $unaccessed\_nodes.put(api)$
8:     while $unaccessed\_nodes$ is not empty do
9:       $node \leftarrow unaccessed\_nodes.get()$
10:      if $node \notin accessed\_nodes$ then
11:        $xref\_nodes \leftarrow node.get\_xref\_from()$
12:        for $xnode \in xref\_nodes$ do
13:          $Graph_{temp}.add\_node(xnode)$
14:          $Graph_{temp}.add\_edge(xnode, node)$
15:          if $xnode \notin accessed\_node$ then
16:            $unaccessed\_nodes.put(xnode)$
17:          end if
18:        end for
19:        $accessed\_nodes.append(node)$
20:      end if
21:    end while
22:    $SFCG_f.add\_nodes\_from(Graph_{temp}.nodes)$
23:    $SFCG_f.add\_edges\_from(Graph_{temp}.edges)$
24:  end if
25: end for
26: if $f$ does not invoke any $api \in List_{api}$ then
27:   $SFCG_f \leftarrow als_f.get\_call\_graph()$
28: end if
29: Return $SFCG_f$
```

C. Code Parsing Stage

In the code parsing stage, the Smali code from the Dex files is parsed into abstract syntax trees (ASTs). ASTs can represent the structured information of the code, making it easier to learn the code semantics. They also offer some resistance to code obfuscation and have been used in code representation-related tasks [33], [34] such as code classification, code search, and clone detection. After generating the AST corresponding

```
private  i(com.tencent.qqgame. installer .QQGameInstaller
     p1)
    {
        this .a = p1;
        return ;
    }
```

Listing. 1: An example of Smali code snippet.

```
{'body': ['BlockStatement',
               None,
               [['ExpressionStatement',
                 ['Assignment',
                  [['FieldAccess',
                    [['Local', 'this' ]],
                    (com/tencent/qqgame/ installer /i,
                     a,
                     Lcom/tencent/qqgame/ installer /
                          QQGameInstaller;)],
                   ['Local', 'p1' ]],
                  '' ]],
                ['ReturnStatement', None]]],
 'comments': [],
 ' flags ': [' private ' ],
 'params': [[['TypeName', (com/tencent/qqgame/ installer /
       QQGameInstaller, 0)],
                  ['Local', 'p1' ]]],
 ' ret ': ['TypeName', ('.void', 0)],
 ' triple ': (com/tencent/qqgame/ installer /i,
               <init>,
               (Lcom/tencent/qqgame/ installer /
                    QQGameInstaller;)V)}
```

Listing. 2. The AST corresponding to the code snippet in Listing 1.

to a function, the AST2Vec model generates structured code semantic feature vectors. Additionally, API feature vectors are generated using one-hot encoding. These two feature vectors are then combined to create fused semantic feature vectors, which represent the micro-behavior of the function node itself.

1) Abstract Syntax Tree Generation: For internal nodes in the FCG, their code can be obtained through decompilation, and further parsing of the code can be done using androguard.decompiler.dad.decompile with DvMethod. This process helps obtain the AST corresponding to the code of the internal function nodes.

Listing 1 shows the Smali code snippet for a function in a particular sample (MD5: bca0902dd49b2ae45ca493691d8 6956a), while Listing 2 displays the AST obtained after parsing this code. Specifically, the parsed AST consists of six parts including body, comments, flags, params, ret and triple.

2) AST Semantic Feature Vectors Generation: To vectorize the parsed AST and enable GNNs to learn the structured code semantics, we design a structured code semantics extraction algorithm called AST2Vec, based on Natural Language Processing (NLP) technologies. In developing the AST embedding generation algorithm, we considered several key factors:

First, parsed ASTs vary in length, posing challenges for traditional Word2vec [35], [36] or TF-IDF-augmented Word2vec methods, which produce long code semantic vectors requiring padding or truncation to standardize lengths. Such adjustments can disrupt AST structures and lead to semantic loss. Second, introducing specialized neural networks to extract structured code semantics from ASTs [37], [38] can improve performance and behavior understanding. However, applying these methods in Android applications involves important considerations:

1) Specialized semantic extraction networks need clearly defined downstream tasks. For instance, the code2vec [37] model is trained with function functionality prediction tasks, and the tree-based [38] model is trained with defect prediction tasks. In Android environments, it is difficult to rapidly create a task-specific dataset for internal function code. In contrast, text-based models only require a substantial corpus for training.
2) Neural network-based models take longer to generate semantic vectors than text-based models, significantly increasing analysis time for Android applications with many function nodes.

Therefore, algorithms were narrowed down to Doc2Vec [39] and the pre-trained model SBert [40]. While the pre-trained SBert model excels in capturing lexical semantics, it may exhibit remarkable disparities when applied to the code analysis domain and has limited learning abilities regarding code structure and syntax. On the other hand, the Doc2Vec model is well-suited for handling long-text scenarios like ASTs. Furthermore, after training on a large corpus of ASTs, it demonstrates a good capability to capture key terms and syntactic structures within the AST. Consequently, it can generate more suitable structured code semantic vectors for ASTs.

Therefore, we opted for the Doc2Vec-based approach to design the AST Structured Code Semantic Generation Algorithm, denoted as AST2Vec. The AST2Vec algorithm takes as input a pre-trained Doc2Vec model m and the parsed AST text ast, and it outputs the structured code semantic vector $Vector_{ast}$. First, the AST text is tokenized, splitting it into a sequence of tokens, and punctuation used for separation, such as “”’, and “,”, is removed from the sequence. Secondly, as seen in Listing 2, the structure of the AST text is represented using nested “[” and “]” symbols. To preserve the structural information of the AST without disruption, the “[” and “]” symbols are retained. These operations yield the processed AST sequence, and subsequently, the corresponding AST semantic feature vector is generated using the semantic vector generation function of the model m. Notes that in this research, the Doc2Vec model’s implementation is done using Gensim [41].

D. Graph Classification Stage

In the graph classification stage, the previously obtained structural feature vectors and code semantic feature vectors are fused as node features. The SFCG with the node features then converted into matrix form for learning by the GNN model.

1) Node Features Fusion: The node features are formed by integrating semantic features from code parsing with structural features acquired through social network analysis. The

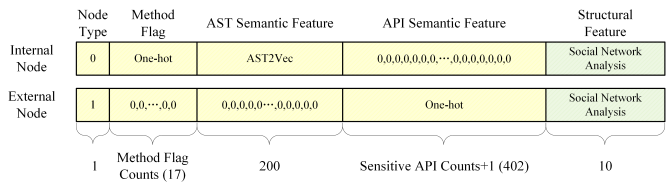


Fig. 2. Structure of node feature vectors.

structure of node feature vectors is illustrated in Figure 2, encompassing comprehensive information about code and structure for each node.

The design of each component is as follows:

- Node type: This component is used to label different node types. Internal nodes are labeled 0, while external nodes are labeled 1.
- Method flag: This component represents flag information for functions. For internal nodes, they have flag information encoded using one-hot encoding. The vector has a total length of 17. For external nodes that lack flag information, a zero vector of length 17 is used.
- AST semantic feature: This component represents the structured code semantics of functions. For internal nodes, their Smali code is available, and the corresponding feature vector with length of 200 is generated using AST2Vec algorithm. For external nodes, the valid code cannot be obtained, and a zero vector of length 200 is used.
- API semantic feature: This component encodes the API calls, as complementary to the semantic information of external nodes. It is set as a zero vector of length 402 for internal nodes. For external nodes, it is one-hot encoded based on their specific API information with respect to the sensitive API list. The sensitive API list contains a total of 401 APIs. If an external node is not within this list, the 402nd position is set to 1, representing "other APIs". Consequently, the vector has a total length of 402.
- Structural feature: This component represents the topological information of the function, which can be obtained through social network analysis.

2) Graph Neural Network Model: The GNN employed in AndroAnalyzer, adopting the model architecture from $POOL_g$ [42] is depicted in the lower-left portion of Figure 1. Initially, it performs three consecutive graph convolution operations and concatenates the features produced at each layer. This aggregation uses skip connections to alleviate the common problem of over-smoothing in graph neural networks [43]. Subsequently, a self-attention graph pooling operation [42] is conducted to retain important nodes, enhancing the detection model's generalization capability. The output of the graph pooling is further subjected to a readout operation (concatenation of average pooling and max pooling). Finally, the readout features are fed through a multi-layer perceptron (MLP) for classification. In fact, the detection performance of AndroAnalyzer is not contingent upon a specific GNN model architecture. It demonstrates good performance in combination with various GNNs. Based on the comprehensive performance in different tasks during the evaluation section IV, we selected $POOL_g$ [42].

TABLE I

CONSTRUCTION OF $Dataset_{CIC}$

Category	Samples	Label (bi-classification)	Label (multi-classification)
Benign	4004	0	0
Adware	1220	1	1
Banking	1107	1	2
Riskware	1072	1	3
SMS	1221	1	4
Total	**8624**	**0, 1**	**0, 1, 2, 3, 4**

IV. EVALUATION

We evaluated the detection performance and generalization capability of AndroAnalyzer. The experiment results are presented in this section.

A. Experimental Setup

The AndroAnalyzer is evaluated on a 16-node GPU cluster, where each node has an Intel (R) Core (TM) i9-10920X CPU @3.50 GHz with 256GB RAM and two NVIDIA RTX 3080 GPUs, and runs Ubuntu 20.04 LTS with Linux kernel v.5.4.0.

We deploy AndroAnalyzer in Python 3.8.

B. Datasets Description

To evaluate the various aspects of AndroAnalyzer's performance, we constructed two experimental datasets, namely $Dataset_{CIC}$ and $Dataset_{AZ}$. We considered the randomness in sample selection and the appropriateness of the sample distribution, and obtained a total of 32,200 Android samples from CICMalDroid2020 [24] and AndroZoo [25].

1) $Dataset_{CIC}$: The CICMalDroid2020 [24] dataset comprises application samples from five different categories: Adware, Banking Malware, SMS Malware, Mobile Riskware, and Benign.

In the experiment, we constructed the dataset $Dataset_{CIC}$ based on CICMalDroid2020. Specifically, to ensure a balanced dataset for binary classification while taking into account the number of samples in each class in CICMalDroid2020, we randomly selected 8624 samples from the dataset in a rough proportion of 1(Benign): 1 (Malware). we used it for evaluating the performance of AndroAnalyzer in binary classification (malware detection) and multi-class classification (malware family recognition) tasks, and details can be found in Table I.

2) $Dataset_{AZ}$: AndroZoo [25] is a continuously growing collection of Android applications. It gathers apps from various sources, including the official Google Play Store, and currently contains 23,081,907 distinct APKs. Each APK undergoes analysis by numerous antivirus products to determine which apps are detected as malware.

In the experiment, we constructed the dataset $Dataset_{AZ}$ based on AndroZoo. Specifically, We utilized AndroZooDownloader (https://github.com/E0HYL/AndrozooDown

TABLE II
CONSTRUCTION OF $Dataset_{AZ}$

Year	Benign	Malware	Year	Benign	Malware
2010	823	997	2017	959	983
2011	999	964	2018	962	914
2012	998	1000	2019	951	933
2013	907	959	2020	897	956
2014	963	803	2021	914	980
2015	902	812	2022	867	168
2016	981	984	**Total**	**12123**	**11453**

loader) to download 23,576 samples from the latest AndroidZoo dataset from 2010 to 2022. The samples for each year were filtered based on Vt_scan_date. Furthermore, to ensure the nature of the samples, benign samples were selected with the condition $vt\_detection = 0$, meaning that all detection engines in VirusTotal did not classify them as malicious software. Malicious samples were selected with the condition $vt\_detection >= 10$, indicating that at least ten detection engines in VirusTotal identified them as malicious software. Notes that the maximum download limit was set to 1000 when downloading samples to ensure dataset balance for the binary classification task. However, some samples couldn't be downloaded successfully due to network issues, resulting in the distribution shown in Table VII. The decrease in the number of malicious samples in 2022 is because only around 300 samples met the criteria in the latest AndroZoo list, and only 168 of them were successfully downloaded.

We used it to evaluate AndroAnalyzer's detection and learning capabilities in an environment close to the real world and the model's generalization ability over time. The details of its construction are shown in Table VII.

C. Evaluation Metrics

The widely used metrics for evaluating the performance of our detection method are Accuracy, Precision, Recall, and F1(F-score).

$$\begin{aligned} Accuracy &= (TP + TN)/(ALL) \\ Precision &= TP/(TP + FP) \\ Recall &= TP/(TP + FN) \\ F1 &= 2 * Precision * Recall/(Precision + Recall) \end{aligned} \tag{1}$$

where TP (True Positive) is the number of malicious Android apps that are correctly labeled as malicious, FN (False Negative) is the number of benign Android apps that are falsely labeled as malicious apps. FP (False Positive) is the number of malicious Android apps falsely labeled as benign, and TN (True Negative) is the number of benign Android apps correctly labeled as benign.

Notes that the specific calculation of evaluation metrics was implemented using the metrics module from the sklearn library. In the calculation process, the average parameter was set to 'binary' for binary classification tasks, as defined in Equation 1. For multi-class classification tasks, the average parameter was set to 'weighted', meaning that the metrics for each class were computed as a weighted average based on the number of samples in each class.

D. Comparison Methods

To evaluate our approach, we selected a set of baseline methods and multiple GNNs for comparison. Specifically, the following methods were included:

1) Baseline Methods: DroidDetector [44]: The method extracted 192 features from the static and dynamic analysis results of Android applications and used DBN deep learning models to detect and classify malware.

HMMDetector [45]: The method was based on the Hidden Markov Model and structural entropy for Android malware detection.

ICCDetector [46]: The method was based on ICC-related features to detect Android malware.

MaMaDroid [5]: The method built Markov chains from the sequences obtained in the function call graph and used them for Android malware detection.

PermPair [2]: The method extracted the permission pairs in the application manifest file to construct a graph distinguishing benign from malware.

The methods described above were all implemented by modifying the code provided in the work of Molina et al. [47].

2) Multiple GNNs: Combine our method with multiple GNNs for detection and classification, including GAT [48], GATv2 [49], GCN [50], GCNv2 [51], GIN [52], and GraphSAGE [53]. The basic GNNs mentioned above were all implemented using PyG (PyTorch Geometric).

E. Evaluation Results

This section will analyze the results of various experiments to answer the research questions (RQs) as follows.

RQ1: How does the sensitive function call graph generation algorithm perform in reducing the graph size?

To verify the performance of the SFCG generation algorithm in reducing the graph size, we collected relevant scale information of the complete FCGs and the SFCGs on the dataset $Dataset_{CIC}$. Specifically, we recorded various metrics, including the APK size, Dex size, the number of nodes, and edges in the complete FCG, and the number of nodes and edges in the SFCG for different categories and the dataset. The detailed data is presented in Table III.

When comparing the statistics of the complete FCG and the generated SFCG, we notice that the data for the sensitive function call graph is nearly one percent of the complete function call graph, effectively reducing storage and computational costs during subsequent processes.

To further discuss the effectiveness of the SFCG generation algorithm, we conducted a detailed analysis and visualization of the complete FCG and the SFCG for a trojan sample (MD5: bca0902dd49b2ae45ca493691d86956a). The complete FCG contains 484 nodes and 1467 edges, while the SFCG contains 20 nodes and 22 edges.

Then, we used centrality (degree centrality) analysis to visualize the results of the FCG after social network analysis. This visualization is shown in Figure 3. The complete

TABLE III
AVERAGE STATISTICS FOR DIFFERENT FUNCTION CALL GRAPHS AND SAMPLE SIZES ON $Dataset_{CIC}$

Category	APK Size	Dex Size	Complete FCG Nodes	Complete FCG Edges	SFCG Nodes	SFCG Edges
Benign	13.36MiB	6.06MiB	46806	131210	461	963
Adware	1.32MiB	0.60MiB	4603	15840	181	536
Banking	1.74MiB	0.69MiB	5918	13944	70	187
Riskware	4.55MiB	0.90MiB	6895	20057	167	634
SMS	0.45MiB	0.07MiB	517	1930	70	647
Total	7.24MiB	3.11MiB	24073	67716	279	717

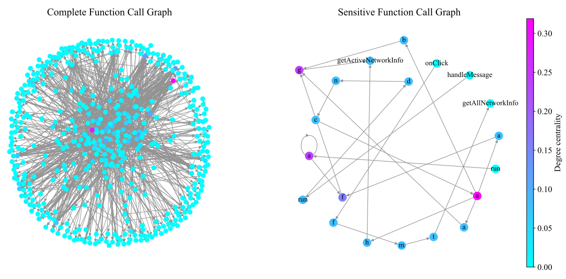


Fig. 3. Distribution of degree centrality of complete and sensitive function call graphs generated by a trojan sample.

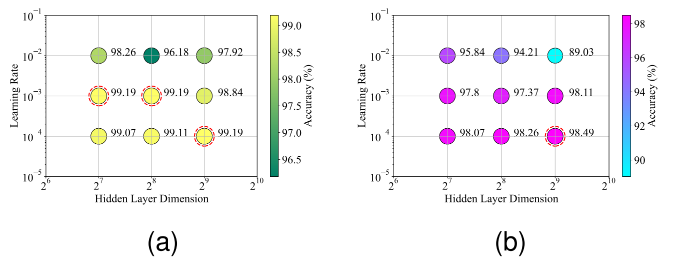


Fig. 4. Performance of different hyperparameters under biclassification and multiclassification tasks.

FCG has numerous nodes and edges, and most nodes have similar degree centrality. This complexity adds remarkable analysis difficulty and computational cost for further analysis. In contrast, the SFCG has removed most irrelevant nodes, focusing on two sensitive behavioral chains related to network behavior: getAllNetworkInfo and getActivateNetworkInfo. This is consistent with the behavior report in VirusTotal (only network communication with no other sensitive behaviors) and the characteristics of Trojan behavior.

Furthermore, the distribution of degree centrality in the SFCG is clearer. Light blue nodes represent sensitive API nodes, or starting nodes of sensitive behavioral chains, dark blue nodes are process nodes within the chains, and purple nodes are composite nodes that are part of multiple behavioral chains. This clear visualization allows for a better understanding of the functionality and importance of different nodes within the behavioral chains. Combined with different social network analysis results, it provides a more granular distinction between nodes in the behavioral chains.

RQ2: What are the optimal parameter settings for the AndroAnalyzer model?

In the deployment of AndroAnalyzer, we employed the following parameters:

For training the AST2Vec model based on Doc2Vec: Considering a balance between rich semantic expression and vector quality, $vector\ size$ is set to 200, $epochs$ is set to 50 and $workers$ is set to 8. The training corpus consisted of the AST content parsed from the samples in the experimental dataset.

For training the detection and classification model based on GNN: DataLoader's $batch\ size$ is set to 32, the optimizer is Adam, and the loss function is CrossEntropyLoss, widely used for classification tasks.

In addition to the aforementioned parameters, it is necessary to determine the hyperparameters, namely the learning rate (lr) and the hidden layer dimension. To find the optimal values for these hyperparameters, we conducted experiments using a training-to-testing data split ratio of 7:3 (a commonly used ratio) and 300 training iterations (to ensure convergence and find the best performance). We employed a grid search approach to systematically test several commonly used values for the learning rate, including 0.01, 0.001, and 0.0001. Similarly, we explored various dimensions for the hidden layers, including 128, 256, and 512. This comprehensive search strategy allowed us to identify the hyperparameter values that yield the best model performance.

AndroAnalyzer's performance on malware detection (binary classification) accuracy on $Dataset_{CIC}$ is illustrated in Figures 4a. The red circles denote the hyperparameter combinations that correspond to the best detection results in these figures. Except 0.01, the remaining learning rates showed similar performance, with three sets of hyperparameters achieving an accuracy of 99.19%. Considering that the 128-dimensional hidden layer is noticeably different from the node feature dimension of 630, implying weaker abstract feature capabilities and fitting abilities, and that the 512-dimensional training result exhibited some degree of overfitting, a balanced choice would be a learning rate of 0.001 and a hidden layer dimension of 256 for binary classification tasks.

AndroAnalyzer's performance on malware family recognition (multi-class classification) accuracy on $Dataset_{CIC}$ is depicted in Figures 4. The optimal hyperparameters were found to be a learning rate of 0.0001 and a hidden layer dimension of 512, while the second-best option was a learning rate of 0.0001 with a hidden layer dimension of 256. Taking into consideration the overfitting observed with a hidden layer dimension of 512 and the hyperparameter selection in binary classification, we opted for a final set of hyperparameters in

TABLE IV
PERFORMANCE OF ABLATION EXPERIMENTS WITH DIFFERENT FEATURES ON $Dataset_{CIC}$

Features	Bi-classification (%)				Multi-classification (%)			
	Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score
Null Feature (Zero Vector)	94.02	93.08	95.97	94.50	89.85	89.85	89.85	89.80
AST Semantic Feature	99.11	98.85	99.50	99.17	98.19	98.18	98.19	98.18
API Semantic Feature	97.64	97.09	98.56	97.82	95.95	95.97	95.95	95.91
Structural Feature	97.49	97.28	98.05	97.67	95.02	94.99	95.02	94.96
Code Semantic Feature	98.84	98.99	98.85	98.92	98.15	98.15	98.15	98.14
AST Semantic + Structural Feature	98.84	99.06	98.78	98.92	97.84	97.83	97.84	97.83
API Semantic + Structural Feature	97.26	97.27	97.62	97.45	95.64	95.62	95.64	95.59
Full Feature (AndoAnalyzer)	**99.15**	99.28	99.14	**99.21**	**98.46**	98.45	98.46	**98.45**

the multi-class classification task, which consists of a learning rate of 0.0001 and a hidden layer dimension of 256.

RQ3: How effective are the fused features of AndroAnalyzer? (Ablation study)

To evaluate the effectiveness of the proposed structured code semantics on detection performance, we conducted experiments with different feature combinations and feature fusion settings under the optimal hyperparameters obtained in RQ2. The training set to test set ratio was maintained at 7:3. The specific results are shown in Table IV.

From the results of binary classification, we observed that the top-performing feature combinations in terms of accuracy all include the AST structured code semantic features, with the "All Features" (Our method) being the best.

The results of multi-class classification show that the overall trend is similar to binary classification. The key difference is that the gap in accuracy between the "All Features" and the single AST semantic feature has increased. This suggests that the fusion of API semantic features and structural features allows the classification model to better distinguish between complex application types, thus adapting more effectively to diverse and real-world application environments.

RQ4: How does the performance of the AST2Vec algorithm compare to the pre-trained model SBert and Code2Vec in the extraction of structured code semantics?

To assess the effectiveness of implementing the structured code semantics extraction module based on AST, we compared the performance of AndroAnalyzer using a self-trained Doc2Vec model, a pretrained SBert model and a pretrained Code2Vec model. We conducted a series of tests covering various training ratios and classification tasks to evaluate the detection and classification performance. The specific results are presented in Figure 5.

The results of the binary classification task show that the AST2Vec algorithm based on Doc2Vec outperforms the SBert-based and Code2Vec-based semantic extraction modules. However, the SBert-based approach still achieves reasonably good detection performance. This is because pre-trained models can extract word semantics, allowing them to capture some degree of semantics present in the AST. However, code is not just plain text. It also contains programming language-specific syntax and semantics. Moreover, ASTs encapsulate structured code semantics. Therefore, AST2Vec, trained on an AST corpus, is more targeted at extracting structured code semantics, leading to better detection performance.

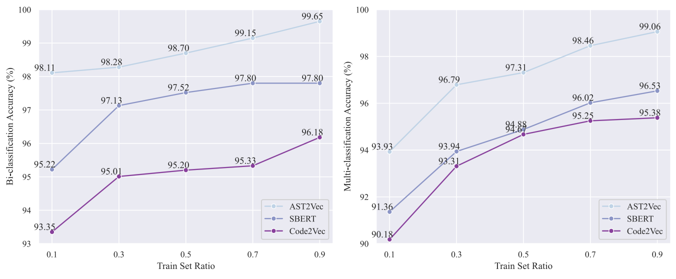


Fig. 5. Accuracy of binary and multi-class classification with different model-based semantic extraction modules on $Dataset_{CIC}$.

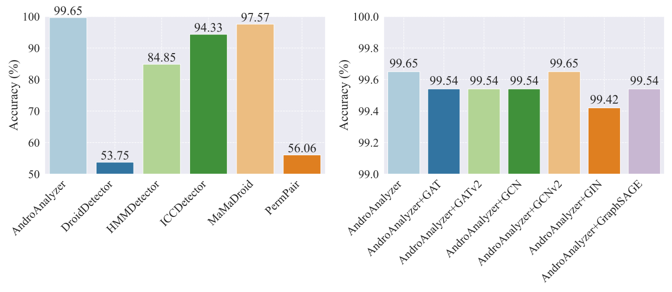


Fig. 6. Comparison of bi-classification performance of different methods on $Dataset_{CIC}$.

The performance of the Code2Vec-based semantic extraction module is slightly inferior to other methods for several reasons. Firstly, it was trained on a generic Java code dataset, making it less aligned with the target scenario's code understanding tasks compared to AST2Vec. More crucially, the Java parser within the Code2Vec model encounters errors with a subset of Java code transformed by Androguard, leading to some loss of semantic understanding and consequently impacting detection performance. The results for multiclass classification are consistent with those for binary classification; however, the difference between Code2Vec-based and SBert-based models has narrowed. This reduction in disparity reflects Code2Vec's enhanced capability for understanding code semantics in more complex scenarios.

RQ5: How does the overall performance of AndroAnalyzer compare to the baseline methods?

To assess the overall performance of AndroAnalyzer, we compared its detection performance with five baseline methods and six GNNs combined with AndroAnalyzer under the optimal parameters obtained in RQ2. The specific results are presented in Table V and Table VI.

TABLE V
PERFORMANCE OF ANDROANALYZER AND BASELINE METHODS ON $Dataset_{CIC}$

Method	Ratio	Bi-classification (%)				Multi-classification (%)			
		Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score
AndroAnalyzer (Our Method)	0.9	99.65	99.78	99.57	99.68	99.08	99.08	99.08	99.08
	0.7	99.15	99.28	99.14	99.21	98.46	98.45	98.46	98.45
DroidDetector [44]	0.9	53.75	53.70	100.00	69.87	23.35	37.83	23.35	17.77
	0.7	53.65	53.63	100.00	69.81	23.21	67.39	23.21	16.64
HMMDetector [45]	0.9	84.85	83.77	89.00	86.31	64.27	75.66	64.27	62.55
	0.7	86.01	85.77	88.61	87.17	64.50	75.44	64.50	62.33
ICCDetector [46]	0.9	94.33	91.75	98.27	94.90	87.63	88.92	87.63	87.66
	0.7	94.24	92.40	97.26	94.77	89.84	90.49	89.84	89.92
MaMaDroid [5]	0.9	97.57	97.84	97.62	97.73	97.22	97.21	97.22	97.18
	0.7	98.10	98.20	98.27	98.23	96.87	96.86	96.87	96.83
PermPair [2]	0.9	56.06	100.00	18.10	30.65	46.35	21.49	46.35	29.36
	0.7	57.16	100.00	20.10	33.47	46.38	21.51	46.38	29.39

TABLE VI
PERFORMANCE OF ANDROANALYZER AND MULTIPLE GNNS ON $Dataset_{CIC}$

Method	Ratio	Bi-classification (%)				Multi-classification (%)			
		Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score
AndroAnalyzer (Our Method)	0.9	99.65	99.78	99.57	99.68	99.08	99.08	99.08	99.08
	0.7	99.15	99.28	99.14	99.21	98.46	98.45	98.46	98.45
AndroAnalyzer+GAT [48]	0.9	99.54	99.57	99.57	99.57	98.73	98.74	98.73	98.72
	0.7	99.34	99.71	99.06	99.39	98.42	98.42	98.42	98.41
AndroAnalyzer+GATv2 [49]	0.9	99.54	99.36	99.78	99.57	98.84	98.84	98.84	98.84
	0.7	99.34	99.57	99.21	99.39	98.30	98.30	98.30	98.30
AndroAnalyzer+GCN [50]	0.9	99.54	99.78	99.35	99.57	98.96	98.97	98.96	98.96
	0.7	99.54	99.57	99.57	99.57	98.61	98.61	98.61	98.61
AndroAnalyzer+GCNv2 [51]	0.9	99.65	99.36	1.00	99.68	98.96	98.97	98.96	98.96
	0.7	99.46	99.50	99.50	99.50	98.57	98.57	98.57	98.57
AndroAnalyzer+GIN [52]	0.9	99.42	99.57	99.35	99.46	98.73	98.73	98.73	98.72
	0.7	99.42	99.35	99.57	99.46	98.42	98.41	98.42	98.41
AndroAnalyzer+GraphSAGE [53]	0.9	99.54	99.78	99.35	99.57	98.96	98.96	98.96	98.96
	0.7	99.42	99.50	99.42	99.46	98.49	98.49	98.49	98.49

First, we compared its performance in the binary classification task. At a ratio of 0.9, the detection accuracy is compared in Figure 6. In the comparison with baseline methods, it can be observed that AndroAnalyzer achieved the highest accuracy. ICCDetector and MaMaDroid also demonstrated decent detection performance, while HMMDetector performed moderately. Additionally, DroidDetector exhibited noticeable overfitting during the detection, achieving an accuracy of 99.49% on the training set but dropping to 53.75% on the testing set. PermPair may suffer from its need for a sufficient number of learning samples to construct a permission pair graph that effectively represents the behavior of that type of samples.

In the comparison of AndroAnalyzer with typical GNN-based methods, it was observed that all methods achieved detection accuracy of over 99.40%. This suggests that AndroAnalyzer is not reliant on a specific GNN model design. Its rational modeling approach and effective feature fusion guarantee its fundamental detection performance. Different GNNs can be chosen based on various detection requirements and real-world scenarios. Additionally, AndroAnalyzer and AndroAnalyzer+GCNv2 achieved the highest detection accuracy of 99.65%.

In the performance comparison for the multi-classification task, also at a ratio of 0.9, the detection accuracy comparison is illustrated in Figure 7. The overall situation is similar to that in the binary classification task. However, it can be observed that both Androanalyzer and MaMaDroid maintained performance similar to that in the binary classification task. This suggests that both possess good learning capabilities for samples' behavior.

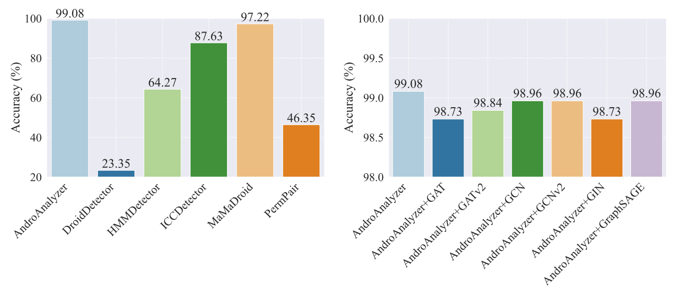


Fig. 7. Comparison of multi-classification performance of different methods on $Dataset_{CIC}$.

Moreover, compared with multiple GNN-based methods, all methods achieved detection accuracy of over 98.5%. AndroAnalyzer demonstrated the best detection performance at 99.08%. This validates that the self-attention pooling layer in the graph filters out some noise or redundant nodes, allowing for the learning of more fundamental behaviors that represent a particular type of program.

TABLE VII
GENERALIZATION ABILITY OF THE COMBINATION OF ANDROANALYZER AND DIFFERENT GNNS ON $Dataset_{AZ}$

Method	2010-2018		2019		2020		2021		2022	
	Accuracy	F1-score	Accuracy	F1-score	Accuracy	F1-score	Accuracy	F1-score	Accuracy	F1-score
AndroAnalyzer (Our Method)	96.76	96.73	96.18	96.23	92.66	92.76	94.19	94.44	95.64	87.47
AndroAnalyzer+GAT [48]	94.06	94.10	95.17	95.27	92.44	92.61	93.40	93.83	92.05	78.87
AndroAnalyzer+GATv2 [49]	93.94	93.97	95.22	95.31	92.17	92.33	93.56	93.94	92.25	79.27
AndroAnalyzer+GCN [50]	95.65	95.70	95.43	95.53	91.09	91.73	89.65	90.70	86.34	67.44
AndroAnalyzer+GCNv2 [51]	96.12	96.17	95.65	95.73	91.20	91.71	92.03	92.69	88.76	72.38
AndroAnalyzer+GIN [52]	96.88	96.88	89.17	88.64	83.69	82.21	90.18	90.25	81.49	25.10
AndroAnalyzer+GraphSAGE [53]	97.00	96.99	95.49	95.57	92.82	92.99	93.98	94.26	93.12	80.00

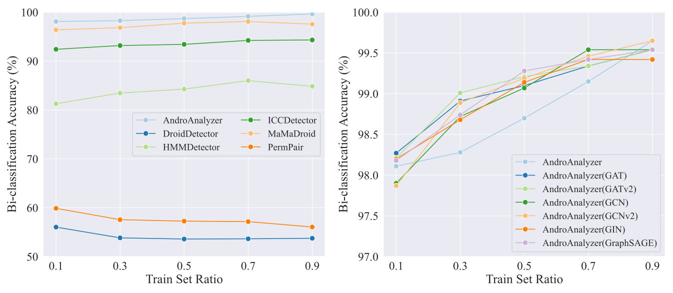


Fig. 8. Comparison of bi-classification performance of different ratios on $Dataset_{CIC}$.

In comparing detection accuracy for the binary classification task across different training set ratios, the results are illustrated in Figure 8. In the comparison with baseline methods, AndroAnalyzer consistently exhibited the best detection performance across all ratios. Compared with multiple GNNs combined with AndroAnalyzer, it can be observed that the overall performance of all methods increases with the growth of the training set ratio. AndroAnalyzer, while showing relatively poorer performance at lower ratios, becomes one of the top-performing models as the ratio increases. This behavior may be attributed to the graph self-attention pooling discards some of the node information to enhance the model's generalization ability. Hence, it may not be able to learn enough about sample behaviors at lower ratios. However, the model achieves remarkable performance once the sample size is sufficient, allowing for a more comprehensive learning of behavioral essence. This also underscores the model's demand for training sample quantity. The overall situation for the multi-classification task is similar to the binary classification task and will not be repeated here.

RQ6: How does AndroAnalyzer generalize to detect new samples?

To evaluate the generalization capability of AndroAnalyzer on new samples at different time points, we conducted evaluation on the $Dataset_{AZ}$. We used a training set proportion 0.9 and trained and tested the model using samples from 2010 to 2018. Subsequently, the trained model was used to test samples from 2019 to 2022. The specific results are presented in Table VII.

Firstly, in terms of the generalization performance of AndroAnalyzer, the effectiveness achieved in training from 2010 to 2018 reached 96.76%. Although this is a decrease compared to the performance achieved on $Dataset_{CIC}$, it is expected due to the diverse and randomly selected samples in $Dataset_{AZ}$, which are closer to a real-world environment. We cannot control whether the collected dataset has the same distribution or covers all malicious families. Therefore, a decrease in performance is inevitable, and the magnitude of the decrease remains acceptable, maintaining a high detection accuracy. Secondly, looking at the detection performance on samples from 2019 to 2022, although there is a slight decrease, the overall detection performance remains above 92%. There is no remarkable drop in performance. Thirdly, AndroAnalyzer exhibit the comparable performance with multiple GNN-based methods in the initial training stage, achieving 96.76% accuracy. Furthermore, regarding generalization performance from 2019 to 2022, AndroAnalyzer outperforms multiple GNNs in terms of accuracy and stability.

V. DISCUSSION

A. Robustness Against Structured Attacks

Graph-based Android malware detection is susceptible to graph structural attacks [23]. These attacks alter the graph structural features by adding nodes, deleting nodes, adding edges, and rerouting, thereby affecting the detection. This is largely attributed to the prevalent usage of node statistical or structural features in existing graph-based detection methods [4], [5], which tend to overlook the code semantics.

To tackle the challenge of graph structural attacks, Li et al. [15] employed community detection on FCGs to partition functional subgraphs, aiming to reduce redundant connections between benign and malicious parts. They also integrated API information extracted from the official Android documentation into node features to mitigate the impact of graph structural attacks.

Similarly, AndroAnalyzer reduces redundancy by focusing on sensitive behavior chains. Furthermore, in the context of semantic fusion, besides API semantics, AndroAnalyzer leverages ASTs to extract structured code semantics for self-written nodes along the behavior chains. This approach allows the detection model to learn functional information. In this way, AndroAnalyzer achieves efficient detection while reducing the influence of graph structural changes on node features.

B. Graph Neural Network Interpretability

Although artificial intelligence technologies enable rapid and accurate malware detection, neural networks are often utilized as black boxes. Regardless of the detection model's accuracy, it is crucial to understand the reasons behind the model's decisions and the nature of the learned features, particularly when dealing with unknown malicious software in real-world scenarios. Therefore, researchers [54] have explored the use of graph interpreters to generate relevant information. In future work, we will attempt to employ graph interpreters to explain the results of the trained graph classification model. This will involve generating subgraph structures and nodes that play a remarkable role in the model's decisions. This approach can reflect the call chains of malicious behavior and key functions, providing valuable insights for malware analysis.

VI. Conclusion

In this paper, we proposed AndroAnalyzer, a method for Android malware detection that focuses on sensitive behavioral chains and integrates AST-based code semantics. Initially, Androanalyzer analyzes the APK files to obtain the FCGs, and extracts the SFCG via the proposed algorithm. It characterizes the macro-level behavior of the application. Subsequently, AndroAnalyzer parses the code of functions in the SFCG. It utilizes AST to extract structured code semantics, enabling to understand the micro-level behavior of each function. Furthermore, AndroAnalyzer generates code semantic features via AST2Vec, and combines them with API semantic and structural features, resulting in SFCG with fused node features. Finally, these graphs are input into a GNN with self-attention pooling for training, yielding an intelligent classifier for Android malware detection. Experimental results demonstrate that AndroAnalyzer achieves superior detection performance in binary and multiclass tasks compared to the baseline methods on two datasets. Additionally, it exhibits good generalization capabilities in detecting samples of 2019-2022. Moreover, in the future, we plan to employ graph interpreters to generate auxiliary information for analyzing malicious behaviors and identifying key functions.

References

[1] A. Kivva. (2023). *Smartphone Malware Statistics, Q1*. [Online]. Available: https://securelist.com/it-threat-evolution-q1-2023-mobile-statistics/109893/

[2] A. Arora, S. K. Peddoju, and M. Conti, "PermPair: Android malware detection using permission pairs," *IEEE Trans. Inf. Forensics Security*, vol. 15, pp. 1968–1982, 2019.

[3] M. Fan et al., "Graph embedding based familial analysis of Android malware using unsupervised learning," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, May 2019, pp. 771–782.

[4] Y. Wu, X. Li, D. Zou, W. Yang, X. Zhang, and H. Jin, "MalScan: Fast market-wide mobile malware scanning by social-network centrality analysis," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2019, pp. 139–150.

[5] L. Onwuzurike, E. Mariconti, P. Andriotis, E. D. Cristofaro, G. Ross, and G. Stringhini, "MaMaDroid: Detecting Android malware by building Markov chains of behavioral models (extended version)," *ACM Trans. Privacy Secur.*, vol. 22, no. 2, pp. 1–34, 2019.

[6] R. Surendran, T. Thomas, and S. Emmanuel, "GSDroid: Graph signal based compact feature representation for Android malware detection," *Exp. Syst. Appl.*, vol. 159, Nov. 2020, Art. no. 113581.

[7] W. Niu, R. Cao, X. Zhang, K. Ding, K. Zhang, and T. Li, "OpCode-level function call graph based Android malware classification using deep learning," *Sensors*, vol. 20, no. 13, p. 3645, 2020.

[8] Y. Wu, D. Zou, W. Yang, X. Li, and H. Jin, "HomDroid: Detecting Android covert malware by social-network homophily analysis," in *Proc. 30th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2021, pp. 216–229.

[9] P. Xu, C. Eckert, and A. Zarras, "Detecting and categorizing Android malware with graph neural networks," in *Proc. 36th Annu. ACM Symp. Appl. Comput.*, Mar. 2021, pp. 409–412.

[10] M. Cai, Y. Jiang, C. Gao, H. Li, and W. Yuan, "Learning features from enhanced function call graphs for Android malware detection," *Neurocomputing*, vol. 423, pp. 301–307, Jan. 2021.

[11] R. Yumlembam, B. Issac, S. M. Jacob, and L. Yang, "IoT-based Android malware detection using graph neural network with adversarial defense," *IEEE Internet Things J.*, vol. 10, no. 10, pp. 8432–8444, May 2023.

[12] Y. Wu, J. Shi, P. Wang, D. Zeng, and C. Sun, "DeepCatra: Learning flow- and graph-based behaviours for Android malware detection," *IET Inf. Secur.*, vol. 17, no. 1, pp. 118–130, Jan. 2023.

[13] H. Wu, N. Luktarhan, G. Tian, and Y. Song, "An Android malware detection approach to enhance node feature differences in a function call graph based on GCNs," *Sensors*, vol. 23, no. 10, p. 4729, May 2023.

[14] S. Shi, S. Tian, B. Wang, T. Zhou, and G. Chen, "SFCGDroid: Android malware detection based on sensitive function call graph," *Int. J. Inf. Secur.*, vol. 22, no. 5, pp. 1115–1124, Oct. 2023.

[15] Y. Li, Y. Hu, Y. Wang, Y. He, H. Lu, and D. Gu, "RGDroid: Detecting Android malware with graph convolutional networks against structural attacks," in *Proc. IEEE Int. Conf. Softw. Anal., Evol. Reengineering (SANER)*, Mar. 2023, pp. 639–650.

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

[17] Y. Hei et al., "Hawk: Rapid Android malware detection through heterogeneous graph attention networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 35, no. 4, pp. 4703–4717, Apr. 2024.

[18] Y. Fan et al., "Heterogeneous temporal graph transformer: An intelligent system for evolving Android malware detection," in *Proc. 27th ACM SIGKDD Conf. Knowl. Discovery Data Mining*, Aug. 2021, pp. 2831–2839.

[19] L. Huang, J. Xue, Y. Wang, Z. Liu, J. Chen, and Z. Kong, "WHGDroid: Effective Android malware detection based on weighted heterogeneous graph," *J. Inf. Secur. Appl.*, vol. 77, Sep. 2023, Art. no. 103556.

[20] H. Zhang, S. Luo, Y. Zhang, and L. Pan, "An efficient Android malware detection system based on method-level behavioral semantic analysis," *IEEE Access*, vol. 7, pp. 69246–69256, 2019.

[21] A. Desnos. *Androguard/androguard: Reverse Engineering and Pentesting for Android Applications*. Accessed: Sep. 20, 2023. [Online]. Available: https://github.com/androguard/androguard

[22] M. Fan, J. Liu, W. Wang, H. Li, Z. Tian, and T. Liu, "DAPASA: Detecting Android piggybacked apps through sensitive subgraph analysis," *IEEE Trans. Inf. Forensics Security*, vol. 12, no. 8, pp. 1772–1785, Aug. 2017.

[23] K. Zhao et al., "Structural attack against graph based Android malware detection," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2021, pp. 3218–3235.

[24] S. Mahdavifar, D. Alhadidi, and A. A. Ghorbani, "Effective and efficient hybrid Android malware classification using pseudo-label stacked autoencoder," *J. Netw. Syst. Manag.*, vol. 30, no. 1, pp. 1–34, Jan. 2022.

[25] K. Allix, T. F. Bissyandé, J. Klein, and Y. L. Traon, "AndroZoo: Collecting millions of Android apps for the research community," in *Proc. IEEE/ACM 13th Work. Conf. Mining Softw. Repositories (MSR)*, New York, NY, USA, May 2016, pp. 468–471, doi: 10.1145/2901739.2903508.

[26] L. Xue et al., "Happer: Unpacking Android apps via a hardware-assisted approach," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2021, pp. 1641–1658.

[27] L. Xue et al., "PackerGrind: An adaptive unpacking system for Android apps," *IEEE Trans. Softw. Eng.*, vol. 48, no. 2, pp. 551–570, Feb. 2022.

[28] L. Xue et al., "Parema: An unpacking framework for demystifying VM-based Android packers," in *Proc. 30th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2021, pp. 152–164.

[29] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the Android permission specification," in *Proc. ACM Conf. Comput. Commun. Secur.*, Oct. 2012, pp. 217–228.

[30] F. Faghihi, M. Zulkernine, and S. Ding, "CamoDroid: An Android application analysis environment resilient against sandbox evasion," *J. Syst. Archit.*, vol. 125, Apr. 2022, Art. no. 102452.

[31] D. Zou et al., "IntDroid: Android malware detection based on API intimacy analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 3, pp. 1–32, 2021.

[32] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using NetworkX," in *Proc. 7th Python Sci. Conf.*, G. Varoquaux, T. Vaught, and J. Millman, Eds., Pasadena, CA USA, 2008, pp. 11–15.

[33] N. D. Q. Bui, Y. Yu, and L. Jiang, "InferCode: Self-supervised learning of code representations by predicting subtrees," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng. (ICSE)*, May 2021, pp. 1186–1197.

[34] N. D. Bui, L. Jiang, and Y. Yu, "Cross-language learning for program classification using bilateral tree-based convolutional neural networks," in *Proc. 32nd AAAI Conf. Artif. Intell. (AAAI)*, 2018, pp. 1–4.

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

[36] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 26, 2013.

[37] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," *Proc. Program. Lang. (ACM)*, vol. 3, pp. 1–29, Jan. 2019.

[38] Z. Cai, L. Lu, and S. Qiu, "An abstract syntax tree encoding method for cross-project defect prediction," *IEEE Access*, vol. 7, pp. 170844–170853, 2019.

[39] Q. V. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proc. JMLR*, 2014, pp. 1188–1196.

[40] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence embeddings using Siamese BERT-networks," 2019, *arXiv:1908.10084*.

[41] R. Řehůřek and P. Sojka, "Software framework for topic modelling with large corpora," in *Proc. LREC Workshop New Challenges NLP Frameworks*. Valletta, Malta: ELRA, May 2010, pp. 45–50. [Online]. Available: http://is.muni.cz/publication/884893/en

[42] J. Lee, I. Lee, and J. Kang, "Self-attention graph pooling," in *Proc. Int. Conf. Mach. Learn.*, 2019, pp. 3734–3743.

[43] D. Chen, Y. Lin, W. Li, P. Li, J. Zhou, and X. Sun, "Measuring and relieving the over-smoothing problem for graph neural networks from the topological view," in *Proc. AAAI Conf. Artif. Intell.*, 2020, vol. 34, no. 4, pp. 3438–3445.

[44] Z. Yuan, Y. Lu, and Y. Xue, "Droiddetector: Android malware characterization and detection using deep learning," *Tsinghua Sci. Technol.*, vol. 21, no. 1, pp. 114–123, Feb. 2016.

[45] G. Canfora, F. Mercaldo, and C. A. Visaggio, "An HMM and structural entropy based detector for Android malware: An empirical study," *Comput. Secur.*, vol. 61, pp. 1–18, Aug. 2016.

[46] K. Xu, Y. Li, and R. H. Deng, "ICCDetector: ICC-based malware detection on Android," *IEEE Trans. Inf. Forensics Security*, vol. 11, no. 6, pp. 1252–1264, Jun. 2016.

[47] B. Molina-Coronado, U. Mori, A. Mendiburu, and J. Miguel-Alonso, "Towards a fair comparison and realistic evaluation framework of Android malware detectors based on static analysis and machine learning," *Comput. Secur.*, vol. 124, Jan. 2023, Art. no. 102996.

[48] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," 2017, *arXiv:1710.10903*.

[49] S. Brody, U. Alon, and E. Yahav, "How attentive are graph attention networks?" 2021, *arXiv:2105.14491*.

[50] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2016, *arXiv:1609.02907*.

[51] C. Morris et al., "Weisfeiler and Leman go neural: Higher-order graph neural networks," in *Proc. AAAI Conf. Artif. Intell.*, vol. 33, 2019, pp. 4602–4609.

[52] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" 2018, *arXiv:1810.00826*.

[53] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, 2017, pp. 1–11.

[54] Y.-H. Chen, S.-C. Lin, S.-C. Huang, C.-L. Lei, and C.-Y. Huang, "Guided malware sample analysis based on graph neural networks," *IEEE Trans. Inf. Forensics Security*, vol. 18, pp. 4128–4143, 2023.