Computer Modeling in Engineering & Sciences

Tech Science Press

https://doi.org/10.32604/cmes.2026.077799

ARTICLE

A Graph-Based Interpretable Framework for Effective Android Malware Detection[#]

Chun-I Fan[1,2], Sheng-Feng Lu[1], Cheng-Han Shie[1], Ming-Feng Tsai[1], Tomohiro Morikawa[3,*], Takeshi Takahashi[4] and Tao Ban[4]

[1]Department of Computer Science and Engineering, National Sun Yat-sen University, Kaohsiung, Taiwan
[2]Information Security Research Center, National Sun Yat-sen University, Kaohsiung, Taiwan
[3]Graduate School of Information Science, University of Hyogo, Kobe, Japan
[4]Cybersecurity Research Institute, National Institute of Information and Communications Technology, Tokyo, Japan

*Corresponding Author: Tomohiro Morikawa. Email: morikawa@gsis.u-hyogo.ac.jp
[#]A preliminary version of this work was presented orally at the 8th International Conference on Mobile Internet Security (MobiSec 2024)

Received: 17 December 2025; Accepted: 02 February 2026; Published: 27 April 2026

ABSTRACT: Due to its partly open-source architecture, which allows for application analysis and repackaging, along with its large market share, the Android operating system is a main target for malware. In recent years, researchers have widely adopted neural network-based methods for detecting Android malware, achieving impressive results but without interpretability. Interpretability is crucial for showing how models behave and identifying biases in their predictions, which helps in validating and improving them. Additionally, in urgent malware analysis situations, interpretability lets analysts quickly assess harmful behaviors and aids in future malware development and investigation. Therefore, interpretability is vital for ensuring that neural network-based malware detection models are trustworthy, predictable, and strong. To address these issues, we propose an interpretable Graph Attention Network (GAT)-based framework for Android malware detection. This framework includes data flow analysis of Android applications to identify malicious behaviors, providing clarity through the attention mechanism of GAT. Analysts and researchers can access detailed information, such as the names and execution order of the involved Android APIs, allowing for better validation and security checks. Experimental results show that our framework achieves a precision of 97.4%. Additionally, case studies highlight the insights that researchers can gain by using this framework.

KEYWORDS: Android malware detection; graph neural network; explainable artificial intelligence

1 Introduction

Due to the advent of smartphones, many applications have been available for various fields such as social, finance, education, and technology, providing services that cater to everyday human needs. According to statistics, most mobile network traffic now originates from mobile phones, with devices running the Android operating system accounting for over 70% of this usage [1]. The Android operating system has thus become the most widely used mobile platform in today's market. Consequently, it has also become a prime target for malware. In 2022, at least 200,000 Android malware have been discovered by researchers [2]. The proliferation of Android malware poses a significant threat to the privacy and security of mobile users. In light of this, lots of research has been proposed to detect the Android malware. However, the traditional signature-based malware detection approaches are not effective in dealing with the vast number of malware.

What's worse, malware may evade detection by using various techniques, such as encryption and obfuscation. Hence, researchers have started shifting towards heuristic-based detection methods. Machine learning-based detection has gained popularity. Various approaches have been proposed, including applications of convolutional neural networks (CNNs) [3], recurrent neural networks (RNNs), multi-layer perceptrons, and auto-encoders. Those methods transform the binary code of the apps into feature vectors and apply calculations to determine whether they are benign or malicious.

According to the study conducted by Senanayake et al. [4], in addition to analyzing binary code, Android malware can be detected by leveraging knowledge of the Android system. Such as the Android system configuration, intent filters, or sensitive Android application programming interfaces (APIs) can assist in detecting malware. The code-semantic-based approaches go much deeper. They recover the higher-level code like Smali bytecode or Java source code from the binary data of Apps. These approaches extract semantic features, including control flows and data flows, yielding promising performance.

Recently, researchers [5,6] have proposed graph neural networks (GNNs) for graph analysis. Some existing research [7–10] also uses GNNs to detect malware by mapping the function call and registers to the graphs. By leveraging the advantages of GNNs in handling topological structures, these approaches successfully capture the pattern within the App, further enhancing the accuracy of Android malware detection.

However, current machine learning-based Android malware detection approaches encounter explainability issues. Popular models with high performance often come with complex structures. As a result, understanding the decision process of the model is challenging. Due to the lack of transparency, landing these approaches may raise concerns about the model's reliability. Moreover, how a machine-learning-based approach considers an App to be malicious is still ambiguous.

Based on the above issue, the Android malware detection field requires a high-performance and explainable approach. Therefore, this research proposes an innovative graph attention network-based Android malware detection framework that provides explainability. This method enables early detection directly from mobile phones, proactively preventing threats from spreading across the broader mobile internet. The contributions are summarized below:

- This study proposes a novel Android malware detection framework based on graph attention networks (GATs) that achieves both high detection performance and explainability.
- The proposed framework determines the presence of malicious behavior by analyzing data flows within Android applications, and leverages the attention mechanism of graph attention networks to output both the probability of an application being malicious and the corresponding suspicious API call sequences as interpretable evidence.
- This study provides illustrative examples demonstrating how researchers can gain insights from the generated explanations, thereby enhancing the overall interpretability of the malware detection process.
- Experimental results show that the proposed GAT-based model achieves competitive performance compared to recent state-of-the-art approaches, while employing a relatively simpler model architecture that facilitates easier implementation.

2 Preliminaries

This section provides an overview of the background knowledge in the Android malware analysis field. It is divided into four subsections: Word Embedding, Graph Neural Networks, Android Malware Detection, and Explainable Artificial Intelligence. Each subsection provides a detailed explanation of the background knowledge related to that topic.

2.1 Frequency Based Embedding

This method constructs word vectors by the frequency of words. There are two specific approaches to this topic. The first is to treat different documents as dimensions and calculate the frequency of words within each document. This involves computing a word-context matrix. The other approach involves counting the co-occurrences of words across all documents. Co-occurrences are the relationships for multiple words appearing in the relative position. This approach generates a co-occurrence matrix. Let's denote the set of all words as $\mathcal{W}$, which contains $\mathcal{N}$ different words. Both methods take a document as input and output an array of word vectors. The calculation processes of these two methods are described in detail below.

The first approach involves defining a context as a sentence within the document and counting the occurrences of each word in each sentence. It creates a word-context matrix in the end. This matrix has each independent sentence as a row and all the words as columns, where each element records the frequency of a specific word appearing in a specific sentence.

Eq. (1) shows an example of a word-context matrix. The second column indicates the occurrences of the word "computer" in each sentence, the third row indicates the occurrences of each word in sentence S_3, and the element in the second column and third row indicates the word "computer" appears 3 times in sentence S_3.

$$X = \begin{array}{c|cccc} & S_1 & S_2 & S_3 & S_4 \\ \hline \text{plant} & 1 & 1 & 0 & 1 \\ \text{computer} & 2 & 1 & 3 & 0 \\ \text{shoe} & 0 & 1 & 0 & 0 \end{array} \tag{1}$$

The second approach involves defining a fixed-length window of l words, known as the context window. This window slides from the beginning to the end of the document, moving one word at a time. For each position, the window covers l words, which form a context, and the approach computed the occurrences of words within the context window. It finally creates a co-occurrence matrix.

Table 1 shows an example of a co-occurrence Matrix. The second row indicates the co-occurrence frequencies of the word "computer" with each word, and the third row indicates the co-occurrence frequencies of the word "boot" with each word. The element in the second column and third row indicates that the words "computer" and "boot" co-occur 4 times in this document.

Table 1: An example of a co-occurrence matrix.

	Plant	Computer	Boot	Chair
Plant	0	3	2	3
Computer	0	0	4	1
Boot	0	1	0	2
Chair	2	3	0	0

However, this technique requires large vocabulary matrixes, which demand significant storage space and high computational complexity. Additionally, the results are prone to be influenced by the corpus size and the data sparsity, making computing accurate semantic relationships difficult. Considering these issues, modern word embedding techniques often employ prediction-based methods.

2.2 Predicted Based Embedding

This method uses machine learning models to predict the current context. The goal is to increase the prediction accuracy of the next word. Word2vec is a widely used prediction-based word embedding method introduced by Tomas Mikolov et al. in 2013. The Word2vec approach involves two main steps: training a word prediction model using an existing corpus. Then, extract the model's weights to be the final word vectors. The approach proposes two training methods for the word prediction model: the Skip-Gram framework and the Continuous Bag-of-Words framework. The Skip-Gram framework is good at context-aware prediction, while the Continuous Bag-of-Words model is good at syntax-aware prediction.

Let N be the total number of words, and $V = v_i \mid i \in Z,\ 0 \leq i < N$ represents the set of all words. Both models consist of a neural network with N layers, and each layer has N neurons. The core idea of the Skip-Gram model is to predict the words within a certain range before or after a given word. The model takes a word as input and predicts a set of l words, where l represents the length of the context window. The model moves through the document word by word, selecting a group of words as the context for training data. For example, if w words are selected and represented as $W = (W_1, W_2, W_3, ..., W_l)$, the corresponding training data is generated as follows:

$$(W_1,\ (W_2,\ W_3, ...,\ W_w)),\ (W_2,\ (W_1,\ W_3, ...,\ W_w,)), ...(W_w,\ (W_1,\ W_2, ..., W_{w-1})) \tag{2}$$

As shown in Fig. 1, each word is represented as an N-dimensional vector. For a word W_i, the model uses one-hot encoding to transform it into an N-dimensional vector, where the ith element is 1 and the rest are 0. The hidden layer of the model does not use an activation function. For each output layer, the $Softmax$ function is used to transform the output into a vector with values ranging from 0 to 1, representing a probability distribution. The model has $l - 1$ outputs, corresponding to l blanks denoted as $U_1, U_2, \ldots, U_{l-1}$. Each blank U_i is an N-dimensional vector that records the probability of each word appearing in that blank. Typically, for each blank, the word with the highest probability is chosen as the representative word for that blank.

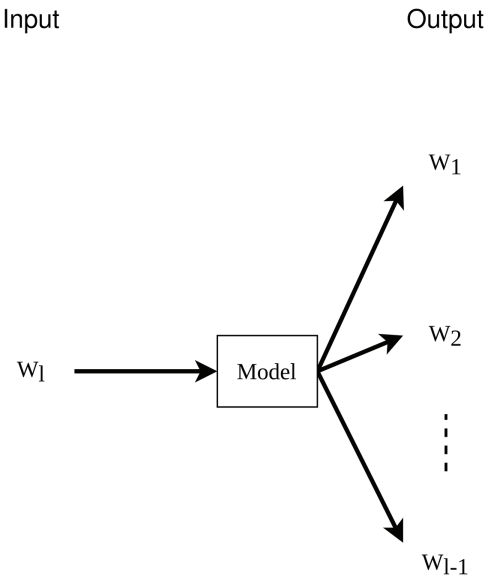


Figure 1: The architecture of the skip-gram model.

As shown in Fig. 2, the core idea of the Continuous Bag-of-Words model is to predict the most suitable word given a fixed range of words within a sentence. Therefore, the model takes $l-1$ words as input and predicts a single word as output, where l is the length of the context window. The model moves through the document word by word, selecting the l words within the window and choosing each of them as the expected output one by one, while summing up the remaining words as the input. Specifically, the selected l words within the window are represented as $W = (W_1, W_2, W_3, ..., W_l)$, and the corresponding training data is generated as follows:

$$(W_2 + W_3 + ... + W_l),\ W_1),\ ((W_1 + W_3 + ... + W_l),\ W_2), ...((W_1 + W_2 + ... + W_{l-1}),\ W_l) \quad (3)$$

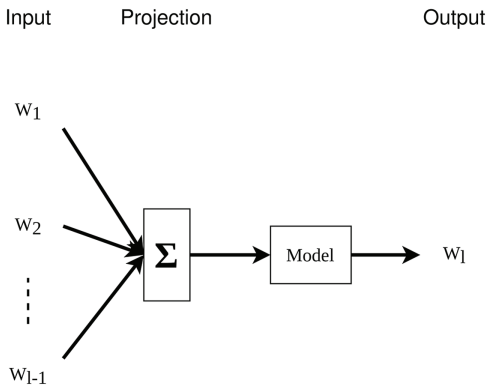


Figure 2: The architecture of the continuous bag-of-words model.

As shown in Fig. 2, $W_1, W_2, W_3, ..., W_{l-1}$ are the inputs to the model, and W_l is the expected output. Each word is represented as an N-dimensional vector. For word W_i, the model uses one-hot encoding to convert it into an N-dimensional vector where the ith value is 1 and the rest are 0. The hidden layer of the model also does not use an activation function. The output layer applies the $Softmax$ function to convert the output into a vector W_l with values ranging from 0 to 1. Each value in this vector corresponds to the probability of occurrence for a specific word. Typically, the word with the highest probability is selected as the final output word.

The Skip-Gram and Continuous Bag-of-Words model have different performance characteristics. The Skip-Gram model performs better in handling rare words, while the Continuous Bag-of-Words model is proficient in dealing with common words. Once completing the model training, we can extract the weights from the hidden layer to obtain the vectors.

2.3 Graph Neural Network

Graph, a data structure consisting of vertices and edges, is widely used to describe abstract relationships between entities. It finds applications in various fields, such as social networks, chemical analysis, recommendation systems, transportation networks, and neural networks, where graphs can record the underlying abstract logic. Due to the ability to precisely describe relationships, research in GNN has gained increasing attention from researchers.

GNNs are machine learning models designed specifically for graphs. Unlike traditional machine learning models, GNNs can handle complex relationships like social networks and chemical molecules. There are types of GNN applications, including node classification, node regression, node clustering, edge classification, edge prediction, and graph classification [11]. We further describe these tasks as follows:

- **Node Classification:** The goal is to classify the nodes in the graph into different categories. An example is the social network user classification task. GNN can utilize the user attributes and the connections between users to categorize unknown users.
- **Node Regression:** The goal is to predict specific features for nodes. An example is the traffic flow prediction task. GNN can predict future traffic flows by leveraging traffic flow data and the connectivity between intersections.
- **Node Clustering:** The goal is to partition the nodes into groups according to their similarity. An example is the social network community detection task. GNN can utilize these connectivity patterns of individuals to identify communities.
- **Edge Classification:** The goal is to categorize the edges in the graph. An example is the social network relationship classification task. GNN can predict the relationship category of the edges, such as family relationships or friendships. This information is valuable for social network analysis or recommendation systems.
- **Edge Prediction:** The goal is to predict the existence of new edges or their weights based on the graph structure. An example is the chemical bond strength prediction task. GNN can utilize the atom properties and the connectivity between atoms to predict the strength of chemical bonds.
- **Graph Classification:** The goal is to classify the graph. An example is the stock trend prediction task. GNN can utilize company information and their interaction patterns to predict stock market fluctuations.

In practice, a model are constructed by multiple layers of GNNs. At each layer, message propagation is performed, which includes sampling, propagation, and pooling operations. The propagation methods can be categorized into convolution, recurrent, or attention-based approaches. Based on the propagation methods, GNNs can be further classified into three types: graph convolutional networks (GCNs), graph recurrent networks, and GATs.

2.3.1 Graph Attention Network

GATs are a type of GNN that introduces the attention mechanism to the graph [12]. The mechanism assigns different attention weights to neighboring nodes during the aggregation process. With that, each node can dynamically adjust the importance of its neighbors based on the specific task. The attention mechanism is inspired by the idea of human attention, where certain elements are given more focus than others.

Assuming there is a graph $G = (V, E)$, where V is the set of nodes and E is the set of edges. Each node i has a feature representation h_i. For each node i and j, the computation of the attention weight α_{ij} typically involves using the similarity of features between nodes i and j.

$$\alpha_{ij} = softmax_j(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}_i} \exp(e_{ik})} \tag{4}$$

Here, $\mathcal{N}_i$ represents the neighbors of node i, and e_{ij} is the similarity of features between node i and neighbor j:

$$e_{ij} = \text{LeakyReLU}(\vec{a}^T [\mathbf{W}\vec{h}_i \| \mathbf{W}\vec{h}_j]) \tag{5}$$

where $\vec{a}$ is the learnable weight vector, $\mathbf{W}$ is a learnable weight matrix, and $\|$ is the concatenation operation. With the weight matrix $\mathbf{W}$, the model can project the node features into a space where the attention mechanism effectively operates. After that, the new feature of node i is obtained by:

$$h_i' = \sigma\left(\sum_{j\in\mathcal{N}_i} \alpha_{ij} \cdot \mathbf{W}\vec{h}_j\right) \tag{6}$$

where σ is the activation function, typically ReLU or LeakyReLU. Through this attention mechanism, GAT can capture useful structural information by adaptively allocating attention based on relationships between nodes.

2.3.2 Static Analysis

Static analysis extracts hard-coded information like App components, permissions, or hard-coded strings from the binaries. It detects malicious code without executing the application, making it a low-cost and efficient method. According to [13], there are 4 types of static analysis methods: Android characteristic-based method, opcode-based method, program graph-based method, and symbolic execution-based method.

- **Android Characteristic-Based Method:** This method predicts malware based on the existence of Android characteristics, such as permissions, API calls, intents, and hardware components.
- **Opcode-Based Method:** This method extracts opcodes from Apps, treats opcodes as text, and cooperates with natural language processing or deep learning models.
- **Program Graph-Based Method:** This method extracts the program graph from Apps to reveal the syntax or semantic information, such as control flow graphs (CFGs), function call graphs (FCG) or data dependency graphs (DDGs).
- **Symbolic Execution-Based Method:** This method simulates App execution by replacing abstract symbols with variables. Then, compare those variables with a rule library.

Machine learning models applied in static analysis often use features such as bytecode, binary code, application components, permissions, or hard-coded strings [9,10,14–16]. These features is susceptible to application obfuscation, limiting the applicability of the method. Some approaches rely on the Android API [17], CFG [7], or FCG [18]. In the Android system, the binary data of an application can be transformed into Smali bytecode or Java code with a small loss. Due to the design of Android application packages (APKs), the Android API is challenging to hide or replace. The obfuscation techniques typically only affect a portion of the graph structure. Therefore, code features are less susceptible to obfuscation techniques. In the context of static analysis for malware detection, preprocessing steps such as decompilation, bytecode analysis, and feature extraction are typically performed before the actual analysis of the application.

3 Related Works

Android malware analysis has always been an area of great interest for researchers. And to this day, many researchers continue to propose various approaches based on different perspectives and techniques. This section will explore the related research on Android malware detection frameworks based on CFGs and FCGs, and discuss their accuracy, robustness, and explainability.

3.1 GDroid

GDroid, proposed by Gao et al. in 2021, is an Android malware detection method based on FCGs [8]. This approach maps multiple Android Apps and Android APIs to a large undirected graph. The graph is then fed into a GCN to generate node features. Finally, all nodes are classified for Android malware detection and family classification. GDroid is the first research to apply GNNs to Android malware detection. Its innovation lies in using word embeddings and the K-nearest neighbor algorithm to filter out Android APIs

highly correlated with malicious behavior. Based on this result, an excellent-performance Android malware classification method is constructed.

The undirected graph defined by GDroid is shown in Fig. 3. The graph nodes consist of Android APIs highly correlated with malicious behavior, categorized Apps, and unknown Apps. The edges connecting Apps and APIs represent App calls to Android APIs, while the edges connecting APIs represent the use of two Android APIs in the same function.

$$G = (V, \mathrm{E}) \tag{7}$$

$$V = V_{app} \cup V_{api} \tag{8}$$

$$E = (V_{app} \times V_{api}) \cup (V_{api} \times V_{api}) \tag{9}$$

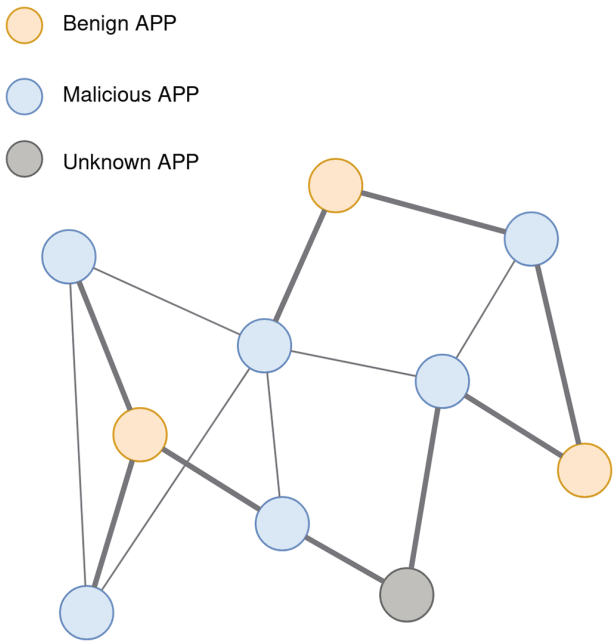


Figure 3: An example of the GDroid proposed graph.

The heterogeneous graph constructed as described above is passed through several layers of GNNs for learning. The training process of each layer of the GNN is defined by the following equation:

$$H_{(i)} = \hat{A} \times H_{(i-1)} \times W_{(i-1)} \tag{10}$$

Let n be the number of nodes and d be the feature dimension. $H_{(i-1)} \in \mathbf{R}^{n \times d}$ represents the feature matrix, which denotes the node features. $\hat{A} \in \mathbf{R}^{n \times n}$ is the adjacency matrix, also known as the Laplacian matrix. $W_{(i-1)} \in \mathbf{R}^{(d \times d)}$ is the weight matrix that represents the weights for propagating the features. $H_{(i)} \in \mathbf{R}^{n \times d}$ represents the propagation result. To balance the amount of information propagated for each node, the adjacency matrix needs to include self-loops and distribute the information based on the number of edges connecting each node to other nodes.

$$\hat{A} = D^{-\frac{1}{2}} A' D^{-\frac{1}{2}} = \frac{A'_{i,j}}{\sqrt{D_{i,i} D_{j,j}}} \tag{11}$$

where $A' = A + I$, $D_{i,i}$ represents the number of edges (degree) for the source node V_i, and $D_{j,j}$ represents the number of edges for the destination node V_i. This method uses categorical cross-entropy error as the loss function $\mathcal{L}$, as shown in the following equation. $\mathcal{Y}\mathbf{A}$ represents the set of categorized Apps, and f represents the total number of App categories. If this method is used for malware detection, $f = 0, 1$ denotes whether the App is malicious. If it is used for malware family classification, f represents the set of all family categories. Finally, Zaf represents the feature vector of the App to be categorized, which is the final result after the propagation in the neural network. Before being input to the loss function, the feature vector is processed by the $Softmax$ function, which distributes the values in the vector between 0 and 1 and can be interpreted as the probability of the App belonging to each category.

$$\mathcal{L} = -\sum_{a \in \mathcal{Y}_\mathbf{A}} \sum_{f=1}^{F} Y_{a\ f} \ln\ Z_{a\ f} \tag{12}$$

Since this method does not rely on existing research to find Android APIs highly correlated with malware, it performs well on datasets collected at different time periods. Experimental results demonstrate that the GNN outperforms existing research in terms of accuracy, precision, recall, and other metrics, confirming the feasibility of applying GNNs to Android malware detection.

3.2 DeepCatra

DeepCatra is a multi-perspective Android malware detection method proposed by Wu et al. in 2022 [7]. It is based on FCGs and CFGs. The approach filters Android APIs that are related to malicious behavior and cross-App or cross-component calls. It extracts the calling sequences of the corresponding APIs and constructs an API call graph that captures the interactions between different components of an Android App and malicious behaviors. The opcode of each API call in the graph is used as training data for a Bi-directional Long Short-Term Memory (BiLSTM) model to obtain the code space features of the App. Then, the entire graph is used as training data for a GNN. Finally, the analysis results from both models are aggregated using a fully connected neural network.

The graph generation process in this method differs from other related studies and involves five steps. First, the entry point of the App and the call paths of the identified APIs are determined using the FCG. Second, for each adjacent function call pair ($Call_a$, $Call_b$) along the call path, the necessary opcodes executed from $Call_a$ to $Call_b$ are extracted as additional information for $Call_b$. Third, the directed graph is constructed by collecting all the call paths. Fourth, to strengthen the connections between APIs and APIs, and APIs and the entry point, DeepCatra replaces the call paths with five types of edges. Finally, due to the limitations of the selected models, the directed graph is transformed into an undirected graph. The five types of edges defined by DeepCatra are as follows:

- **Critical Path:** Describes the relationship between the entry point and a highly correlated Android API used in the program.
- **Intent-Sending Edge:** Describes the API used by the entry point to communicate with other application components.
- **Neighbor Edge:** Describes the relationship between two function or API calls within the same function.
- **Inter-Component Communication Edge:** Describes the relationship between a highly correlated Android API and the activation of a specific application component.

- **Implicit Neighbor Edge:** Describes the relationship between the execution of a highly correlated Android API and obtaining the result of the application component's execution.

Based on the graph generation algorithm described above, DeepCatra addresses the complex issues associated with using CFGs for GNN analysis. The experimental result shows an outstanding performance. Moreover, it confirms that the topological structure derived from the function call and CFGs needs to be simplified and highlighted with critical information to be suitable for GNNs.

4 The Proposed Framework

This setcion introduces the proposed Android malware detection framework. In the Android malware detection field, approaches based on the Android API are popular. Android APIs serve as a bridge connecting the application and the Android system. They provide access to the device and construct most of the application's business logic, such as accessing the files, control the sensors, and communicate with external entities. Hence, they have been focal in past research on Android malware detection. However, despite achieving remarkable accuracy, related researchers have encountered issues when landing their approaches in practical applications. That is, make the inference results explainable.

Existing researches based on GNNs often use heuristic features (e.g., application bitmaps) or complex model architectures (e.g., fully connected networks and LSTMs). The complexity causes the decision processes to be unachievable. In light of this, this research proposes a GAT-based detection framework with explainability. Among the Android APIs, we utilize the attention mechanism of GATs to locate the API where the model pays the most attention. Therefore, given an App, we can find the most suspicious Android API sequence relating to the malicious behavior. This design makes the model output explainable.

Here is the organization of this section. Section 4.1 briefly introduces the overall architecture of the framework. Section 4.2 focuses on the data preprocessing of the framework. Section 4.3 discusses the design of the GAT-based model deeply.

4.1 The Architecture

The whole architecture is in Fig. 4. Arrows indicate the data flow. Rectangles indicate the detailed procedure performed in the stage. Cards are the input or output data of the procedures. Note that the two cards on the right are the output. The framework takes an App (in the APK format) as input and outputs two results, the probability of the App being malicious and the suspicious API call sequences. The entire detection process can be divided into three stages: graph extraction (Stage 1), API feature extraction (Stage 2), and malicious behavior detection (Stage 3). Stage 1 aims to extract the data dependency of suspicious API with reaching definition analysis. This stage outputs a directed graph called Suspicious API Call Data Dependency Graph (SACDDG). Stage 2 incorporates an NLP technique, word embedding, to encode Android APIs into API features. We adopt a Skip-Gram model to do the process. The framework then proceeds to Stage 3. This stage adopts a proposed GAT-based model to detect malicious behavior in the SACDDG. For each data flow on the SACDDG, the model analyzes it using API features and outputs the probability of involving any malicious behavior. Ultimately, we aggregate these probabilities with $Softmax$, outputting the probability of the Apps being malicious.

Stages 1 and 2 are the two steps of data preprocessing, which we will discuss further in Section 4.2. Stage 3 involves a proposed GAT-based model, which we will discuss further in Section 4.3.

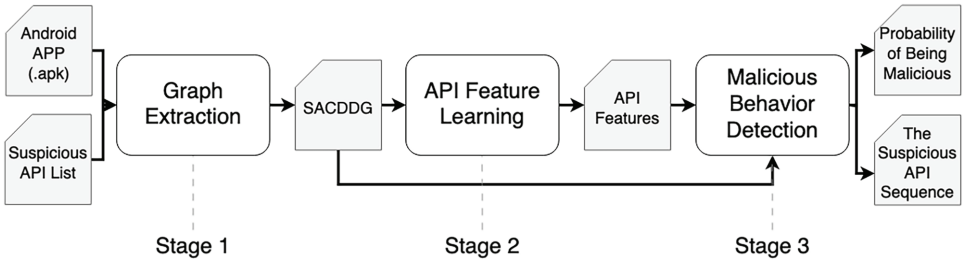


Figure 4: The architecture of the framework.

4.2 Data Preprocessing

Data preprocessing has a significant impact on the model, especially its effectiveness. A high-performance model often requires a large amount of training data. However, collecting high-quality data has always been a challenge. Thus, we process the data carefully to ensure the model achieves optimal performance. Additionally, converting the application into a suitable format for a machine-learning model is necessary. The proposed GAT-based model takes two types of data as inputs. One is a list of API features representing the behavioral characteristics of suspicious APIs. The other is a directed graph recording the data flows in the App, which we refer to as SACDDG.

The data preprocessing involves three steps. First, to help the model focus on the critical APIs, we select a list of suspicious APIs highly related to malicious behavior. We will discuss this in Section 4.2.1. Second, to prepare a graph representative of the App, we utilize reaching definition analysis to extract SACDDG. The process will be discussed in Section 4.2.2. Finally, a Skip-Gram model is used to obtain the API features, which will be discussed in Section 4.2.3.

4.2.1 Suspicious API List

Android APIs are defined by class, method, and descriptors, serving as the bridge between the Android system and the Apps. An Android API is a fundamental behavior of the Android App (e.g., enabling Wi-Fi, reading SMS messages, and launching applications). An App can use a sequence of Android APIs to build its business logic. Thus, one can observe the API sequence to know the business logic of an App, which enables a precise determination of the App performing malicious behaviors. Hence, tracking Android APIs is one of the popular approaches in the Android malware detection field. However, machine learning models encounter two issues when utilizing Android APIs. The first issue is the sheer number of Android APIs. Tracking all APIs in the model is non-practical due to the complexity. The second issue is APIs with the same class and method names typically share the same behavior. Hence, tracking all APIs does not necessarily improve model performance.

To address the first issue, this research selects the top 50% frequent APIs (11,135 APIs) from the dataset as the suspicious APIs. In the following data preprocessing process, we will focus on only these APIs and drop others. For more details, please take a look at Sections 4.2.2 and 4.2.3.

To address the second issue, this research combines those APIs sharing the same class and method names. After this process, there are 631 suspicious APIs remaining. Subsequent data preprocessing will focus on these APIs.

4.2.2 Suspicious API Call Data Dependency Graph

To accurately describe API behavior, this research uses the API call data dependency graph (ACDDG). As a representation of the APK, this graph takes APIs as nodes and the relationship of two APIs operating on the same data flows as edges. For any two connected nodes in the graph, the edge points to the API that occurs later. The process of graph generation involves two steps. First, for each method in the APKs, we perform the reaching definition analysis to find the data flows. A data flow consists of several Android APIs as nodes. It describes the lifespan of data in the App. The data is returned by the first API and processed by the rest. After finding all the data flows, we combine them to build an ACDDG with the following two steps: First, we create an empty graph. Second, we add the data flows to the graph one by one. For each path, if any node in the flow exists in the graph, we combine the node in the flow and the node in the graph. The resulting node inherits the successors and predecessors of the two original nodes. Otherwise, we directly add the node to the graph. This process is repeated until we add all data flows to the graph. After generating an ACDDG, we perform graph pruning to drop the non-suspicious APIs (refer to Section 4.2.1). When removing a non-suspicious API, an edge is created between the removed API's successors and predecessors to keep the graph structure as less changed as possible. After that, we obtain a SACDDG. The above process is shown in Fig. 5.

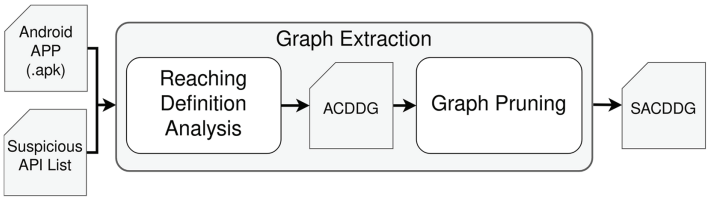


Figure 5: The graph extraction stage in the proposed framework.

4.2.3 Suspicious API Features

A common approach that extracts the behavior characteristic of Android APIs is using NLP techniques, such as word embedding. For example, GSDroid uses a Skip-Gram model and API co-occurrences to obtain API features [8]. By GSDroid's definition, an API co-occurrence is a tuple composed of two APIs under the same method. However, this definition can cause an issue. If the method processes multiple data flows concurrently, the definition will consider two unrelated APIs in the method (i.e., two unrelated APIs processing the different data flows) to be an API co-occurrence. This case will lead the Skip-Gram model to generate distorted API features. Therefore, we use a stricter definition based on GSDroid, which requires the two APIs to operate on the same data flow. This strict definition ensures the APIs contribute to the same behavior. The process of extracting API co-occurrences involves two steps. First, we get the data flows from the SACDDG. Then, we construct the API co-occurrence from the data flows. After that, we use a Skip-Gram model to get the API features from the API co-occurrences. The whole process is shown in Fig. 6.

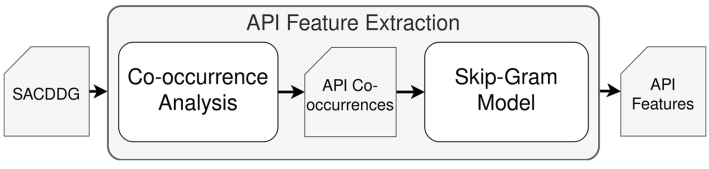


Figure 6: The API feature extraction stage in the proposed framework.

The structure of the Skip-Gram model is illustrated in Fig. 7. This model is modified from PyTorch. PyTorch is an open-source machine learning library that provides researchers with numerous community-maintained building blocks for machine learning [19]. The model consists of a lookup table that stores API features and two fully connected neural network layers.

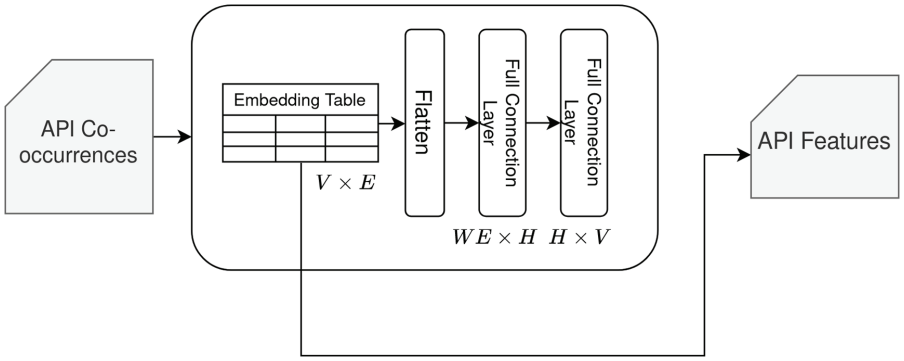


Figure 7: The modified skip-gram model.

4.3 The GAT-Based Suspicious Data Flow Detection Model

The structure of the GAT-Based model is shown in Fig. 8. The model takes SACDDG as input and uses GAT layers to output two results, the probability of the App committing malicious behavior and the suspicious API sequences related to the malicious behavior. The model consists of two parts. The first part in the blue rectangle aims to compute the probability of the data flows involving malicious behavior. Hence, the first part uses multiple GAT layers with descending dimension sizes. The second part in the red rectangle aims to aggregate the probability of the data flows into the probability of the App being malicious. Hence, the second part applies a graph pooling technique and an additional GAT layer. Consequently, the model outputs the probability computed by the second part and a suspicious API call sequence by tracking the distribution of the attention weights of the first part.

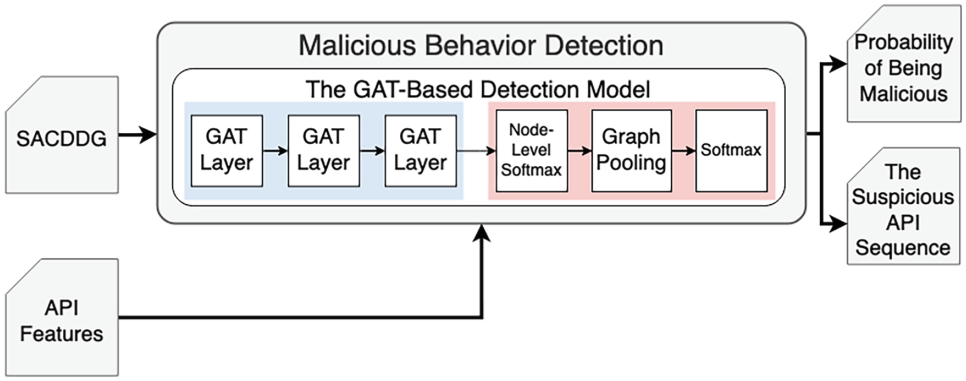


Figure 8: The GAT-based detection model.

The first part consists of GAT layers with descending dimension sizes. Each layer additionally uses LeakyReLU as the activation function and layer normalization. The utilization of GAT layers is motivated by three main reasons:

- First, through the attention and the propagation mechanism, the model computes the feature of the data flow by simulating the execution of the data flow. For example, assume there is an unknown App having

a data flow that invokes API A and API B sequentially. The SACDDG will consist of at least two nodes. One represents API A and the other represents API B. Because API A and API B both operate the same data flow, a directed edge is created between the two nodes. When a node carrying the API features of API A is propagated through a GAT layer, the node spreads its API feature. After that, the other node will receive the features and combine the features with its own features. This propagation simulates the execution of API A and API B sequentially. The feature of the execution result will be stored in the last node. Thus, through the propagation of the GAT layer, the model obtains the features of the data flows.

- Second, we use the attention mechanism from the GAT layer to focus on those suspicious data flows. For each node, the attention mechanism applies higher weights to the data flows that are highly related to suspicious behavior. Hence, the attention mechanism can migrate the feature vanishment during the propagation.
- Third, we decrease the sizes of feature dimensions per layer to force the model to focus on those features highly related to suspicious behavior.

After the whole feature propagation, the feature dimension will eventually reduce to 2 and all the nodes should store suspicious data flow features. We then use $Softmax$ to convert the node features into a probability of the data flow being malicious. This design contributes to the explainability of the model, as we will discuss in Section 5.4.

The second part utilizes a single layer of the GAT to aggregate the probability in the nodes. There are three steps involved. First, the model performs Graph Pooling, which creates a virtual node outside the graph and multiple edges pointing from the nodes in the graph to the virtual node. Next, a GAT layer is applied to aggregate the probability of all nodes to the virtual node. In the end, we use $Softmax$ to transform the aggregate result into a value distributed between 0 and 1. We then consider it as the probability of the APK being malicious. Also, we follow the procedure discussed in Section 5.4 to output the suspicious API sequences.

5 Evaluation

This section further describes the hybrid-parameters setup and dataset used for the evaluation. Then, we evaluate the performance of the proposed framework and compare it with other approaches. Section 5.1 discusses the datasets and overall process of the whole experiment. Section 5.2 presents the experimental results of the framework and the hyperparameter settings that yield the best performance. Section 5.3 compares the system with other existing approaches. Section 5.4 discusses the explainability of the prediction results.

5.1 Experiment Setup

The evaluation uses two datasets, Genome and play.google.com, collected from AndroZoo as the training data. AndroZoo is a well-known platform for Android App datasets that proposed by Allix in 2016 and maintained up to now [20]. The platform collects applications from the Google Play Store and other sources. Each sample comes with structural information such as package name, SHA256, and first appearance date. In this research, Apps flagged by VirusTotal are considered malicious, while Apps not marked are considered benign. The two datasets contain a total of 17,743,917 samples, with 17,742,671 benign and 1246 malicious samples.

Due to the significant disparity in the number of benign and malicious Apps, this research drops benign Apps randomly to balance the dataset. The final dataset consists of 1219 benign Apps and 1181 malicious ones (some Apps were excluded due to the parsing library, Androguard, reports errors). The final ratio of malicious and benign Apps is 49.2%. Fig. 9 is the distribution of the target Android API levels of those Apps.

The value was collected from the manifest files. If an App did not provide this information, we set the value to −1.

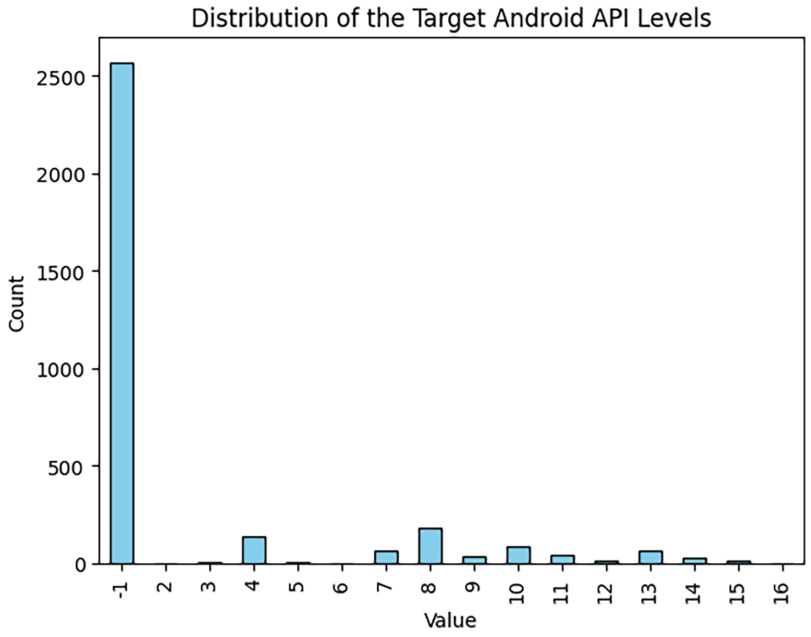


Figure 9: The distribution of the target android API levels.

Subsequently, multiple GAT-based models are trained using the balanced dataset, which is divided into training and validation sets in a 70%–30% ratio. After training on the 70% training portion, the best-performing model is selected and integrated into the proposed detection framework.

5.2 Experiment Results

This section uses the dataset mentioned in Section 5.1. The experiments ran on an Ubuntu 22.04 host machine with an AMD Ryzen 9 5950X processor and 20 GB of memory. To accelerate the model training, an NVIDIA GeForce RTX 3080 was used. The environment setup for the experiments is summarized in Table 2.

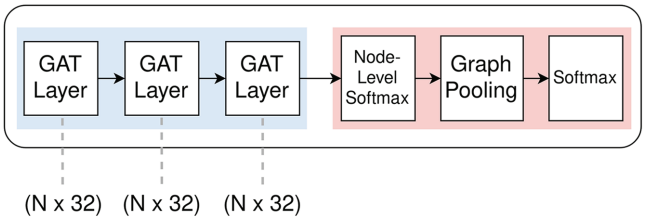
Table 2: Environment setup.

Operation System	**Linux Ubuntu 22.04 LTS**
CPU	AMD Ryzen 9 5950X
Memory	20 GB
GPU	NVIDIA GeForce RTX 3080

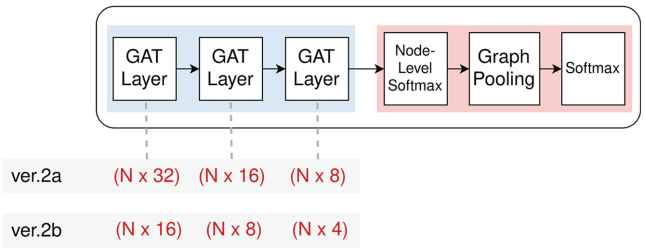
5.2.1 The GAT-Based Detection Model

The suspicious data flow detection model is the most important part of this proposed framework. Therefore, we evaluate various model candidates. All the candidates are listed in Fig. 10. Table 3 shows the experimental result.

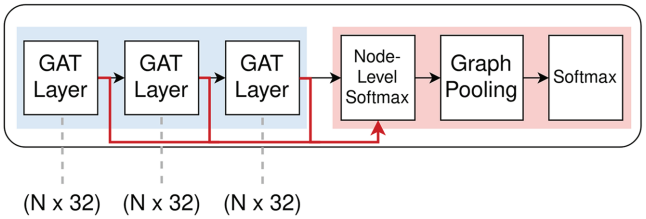
- **Candidate 1** (refer to Fig. 10a) was the initial design. It utilized three GAT layers with constant feature dimensions to detect suspicious data flows. However, in the preliminary experiment, Candidate 1 showed unsatisfactory results. Therefore, two following candidates were proposed to address this issue.
- **Candidate 2** (refer to Fig. 10b) addressed the problem from the first candidate by decreasing feature dimensions layer by layer. We believed that decreasing feature dimensions would guide the model to focus on suspicious behavior-related features. This candidate includes two patterns: decrease linearly and decrease steeply.
- **Candidate 3** (refer to Fig. 10c) attempted to solve the problem by using jumping knowledge. Since deep neural networks often suffer from feature vanishing issues, Candidate 3 aggregated the nodes from each neural network layer when generating the probability of malicious behavior for the application.
- **Candidate 4** (refer to Fig. 10d) combined the techniques of decreasing features and jumping knowledge.



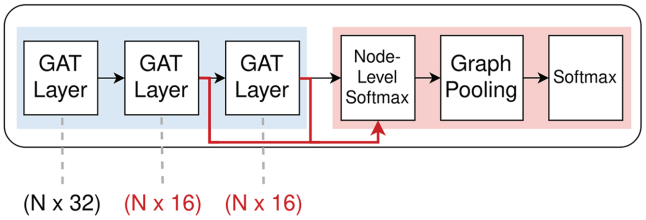
(a) Candidate 1



(b) Candidate 2a and 2b



(c) Candidate 3



(d) Candidate 4

Figure 10: Model candidates for the proposed framework.

Table 3: Comparison of using different model structures.

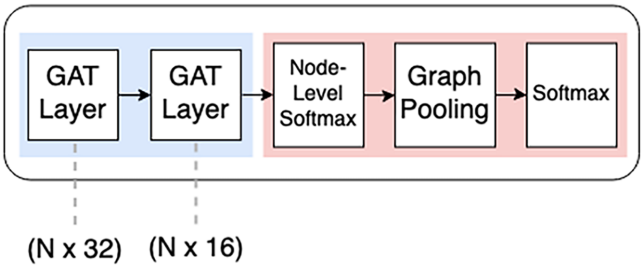
Candidates	Accuracy	Precision	Recall (TPR, Sensitivity)	TNR (Specificity)	F1-Score
ver.1	51.69%	50.87%	98.44%	7.96%	67.08%
ver.2a	**96.61%**	**96.13%**	**97.14%**	**96.25%**	**96.63%**
ver.2b	81.25%	74.39%	95.31%	68.27%	83.56%
ver.3	50.65%	50.33%	100.00%	4.45%	66.96%
ver.4	90.76%	89.82%	91.93%	89.93%	90.86%

Note: The bolded entry indicates the selected candidate based on the experimental results.

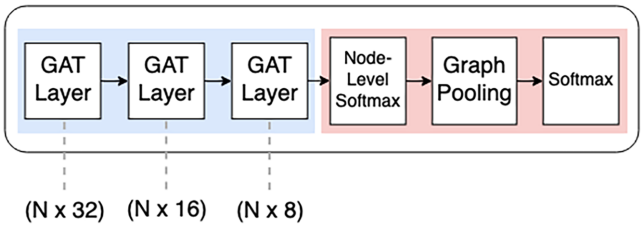
From the experimental results, it was observed that the candidates with decreasing features performed better in terms of F1-Score compared to others. Candidate 2a showed a better result than Candidate 2b, suggesting that the magnitude of the feature decrease in Candidate 2b might have been too large. The architecture using jumping knowledge exhibited a significant decrease in performance. Based on the experimental results, Candidate 2a was selected.

5.2.2 The Number of Network Layers

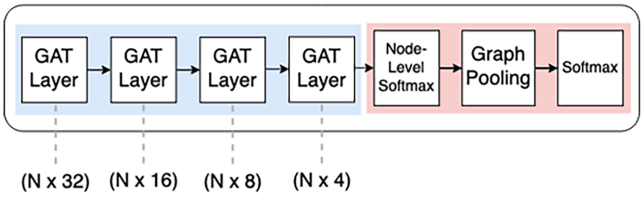
The number of GAT layers also impacts the model's performance. To find the best setup, we evaluate the framework with 5 different layer configurations. All the configurations are listed in Fig. 11 and the results are listed in Table 4.



(a) Candidate 2a-2

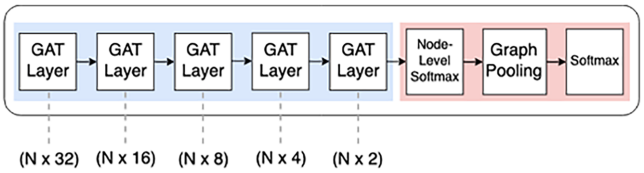


(b) Candidate 2a-3

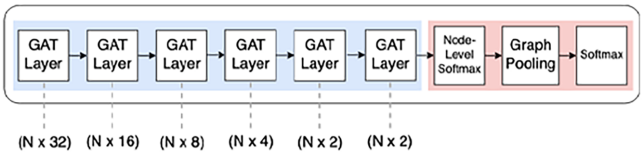


(c) Candidate 2a-4

Figure 11: (Continued)



(d) Candidate 2a-5



(e) Candidate 2a-6

Figure 11: Layer configuration candidates for the proposed framework.

Table 4: Comparison of using different hyperparameters.

Candidate	Accuracy	Precision	Recall (TPR, Sensitivity)	TNR (Specificity)	F1-Score
ver 2a-2	96.0%	96.6%	95.3%	96.7%	95.9%
ver 2a-3	**96.6%**	**96.2%**	**97.1%**	**96.2%**	**96.6%**
ver 2a-4	83.2%	80.0%	88.5%	78.6%	84.1%
ver 2a-5	54.0%	52.1%	98.7%	12.2%	68.2%
ver 2a-6	50.0%	50.0%	100.0%	3.2%	66.7%

Note: The bolded entry indicates the selected candidate based on the experimental results.

The experimental result shows that Candidate 2a-3 had the best performance in terms of F1-Score. We observe that the F1-Score decreases as the number of layers increases, suggesting that the feature vanishing issue appeared. Also, the result of Candidate 2a-2 shows the impact of insufficient layers. Based on the experimental results, Candidate 2a-3 was selected.

5.3 Comparisons

Existing studies have also proposed approachs based on GNNs. For example, MsDroid, proposed by He et al. [21], also utilizes a call graph constructed from Android APIs to detect malicious behavior and achieves remarkable performance. The difference between He et al. and our approach lies in the focus. He et al. emphasize the topological features of the call graph, while our approach focuses on the propagation results of suspicious behaviors. In this section, we compare our approach with He et al. using the dataset introduced in Subsection 5.1. The hybrid parameters used are in Table 5. And, the experimental results are shown in Table 6.

The experiment shows a similar performance between our approach and He et al., which is at most 1.9%, with an average difference of approximately 1.05%. However, with this similar performance, ours requires fewer resources. Figs. 12 and 13 demonstrate the number of graph nodes and edges extracted by He et al. and ours. Fig. 12 shows that our approach uses approximately half the number of nodes compared to He et al. Meanwhile, Fig. 13 shows that our approach does not lose a significant number of edges, of which only about one-third are lost. This statistic indicates that our approach extracts information accurately from the unknown application. Additionally, ours achieves similar performance with fewer resources.

Table 5: Hyper-parameters setup.

Hyper-parameters	Ours	He et al. [21]
k	20%	–
Epoch	500	800
The Dimension of the API Features	**32**	**128**
Batch Size	64	64
Learning Rate	0.1	0.8

Note: Bold denotes our setting, which reduces the API feature dimension from 128 to 32 (75% reduction) compared with He et al.

Table 6: Comparison of the proposed framework with other works.

	Accuracy	Precision	Recall (TPR)	TNR	F1-Score
He et al. [21]	98.60%	99.13%	97.45%	–	98.29%
Ours	97.50%	97.14%	97.88%	97.26%	97.51%

In addition, our approach also provides the explanation result discussed in He et al., which is called Calling Heat Graphs. By He et al.'s definition, a Calling Heat Graph is a graph providing weights of the data flow. The graph allows researchers to identify the data flow that performs malicious behavior. He et al. treats this problem as an optimization problem that maximizes mutual information. In other words, it calculates the weights by removing edges on the graph and re-examining the detection results. Meanwhile, our approach naturally provides this information. We use the attention weights during propagation as the weights, which we discuss further in Section 5.4.

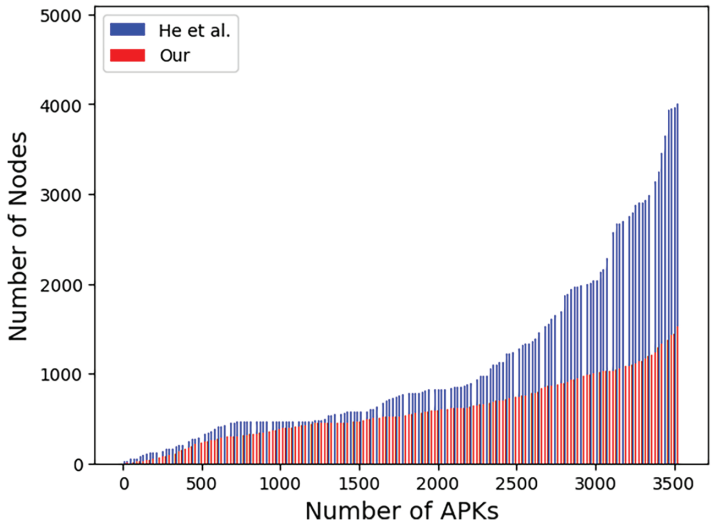


Figure 12: Comparison of nodes extracted from genome [21].

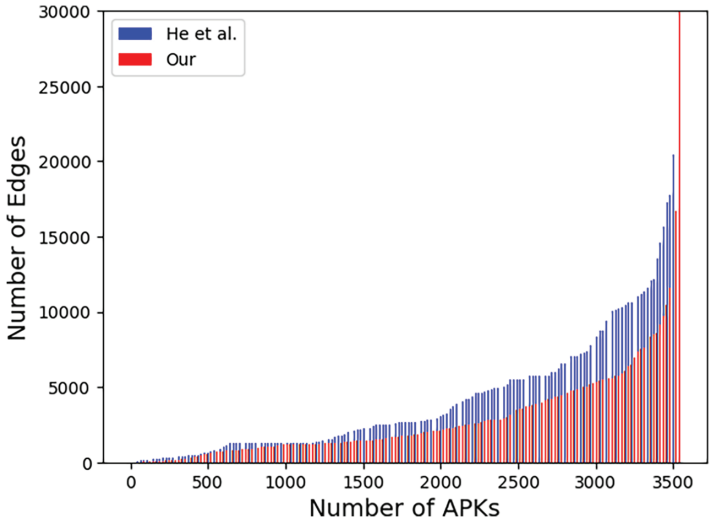


Figure 13: Comparison of edges extracted from genome [21].

5.4 Explainability

This subsection discusses the explainability of the framework. The framework provides explainability by leveraging the attention mechanism of graph attention networks. Researchers can obtain information on the involved Android APIs, such as their names and execution orders. With this information, researchers can validate the framework's prediction and conduct further analysis on the Apps. We will discuss the procedure the framework uses to provide explanations.

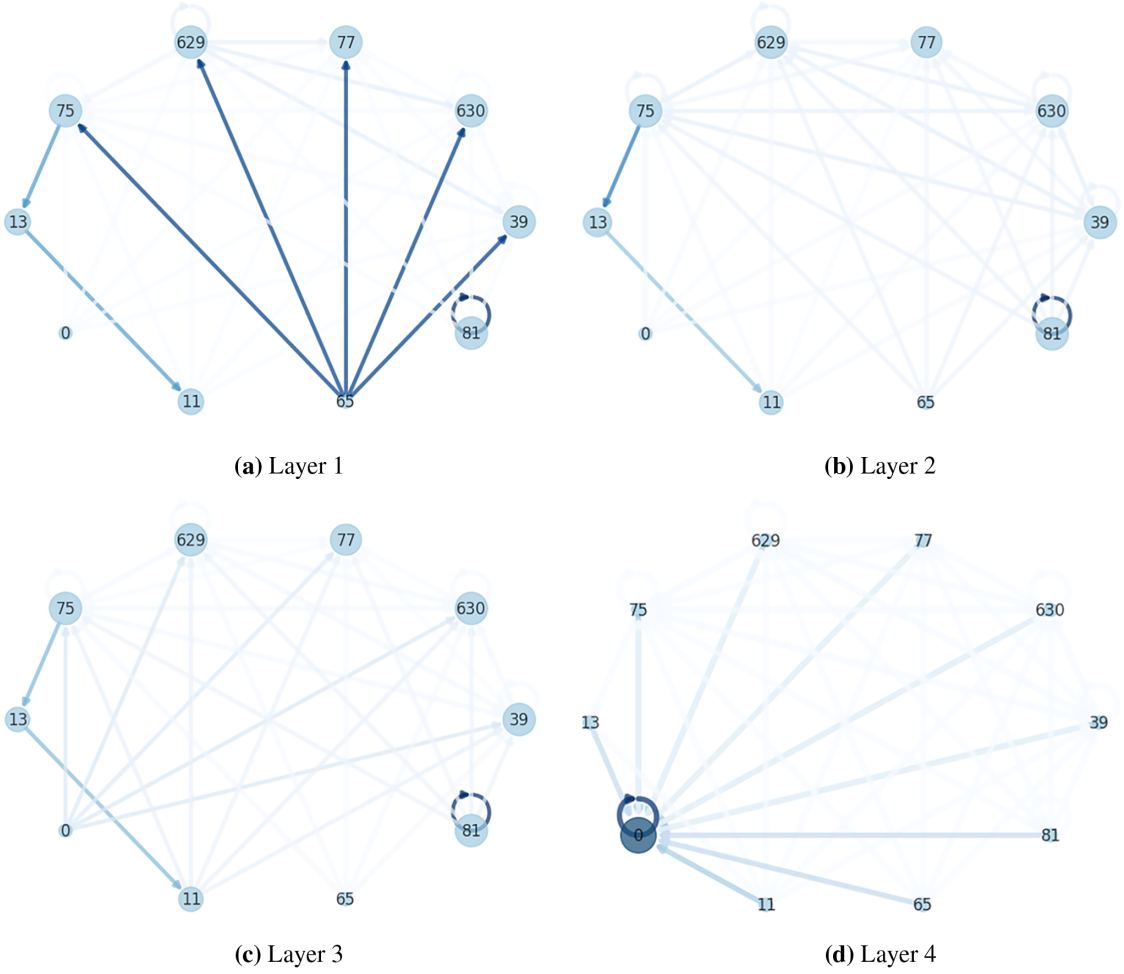
The explainability is built on the attention mechanism of the GAT layers. Therefore, given the attention weights of the GAT layer, we know which nodes on the graph receive the most attention. As nodes with higher weights contribute more to the result, we consider the nodes with the highest weight as the critical nodes. By determining critical nodes from each GAT layer, we can identify a path. Next, because the nodes are the representations of the Android API, we can treat the path as an API call sequence, which reveals the behavior of the App.

This research uses malware from the Genome dataset to illustrate the above steps. The SHA256 hash value of the Apps is as follows:

$$F8302D1AF34DFD0089A2D160B9ADF752D6CF392451316B3B5B205C159FC0B500$$

We visualizes the attention weights in Fig. 14. The figure contains four parts, corresponding to the four GAT layers in the model (refer to Section 4). For each subgraph, nodes represent APIs, and the numbers on the nodes represent API indices (refer to Table 7). For easy analysis, we assign larger sizes to nodes with higher weights. Nodes may have self-loop edges and edges connecting. Darker edges indicate more attention. It is important to note that Fig. 14d is the last layer, responsible for aggregating all node features to a virtual node. Therefore, Fig. 14d only visualizes the attention weights related to the virtual node, node 0.

In Fig. 14, we analyze the propagation of each node feature in a reverse manner. According to Fig. 14b, nodes 0, 11, and 13 impact the output the most. There are two nodes with the highest attention weights, nodes 0 and 11. Based on that, the App may perform two suspicious behaviors. Next, we will discuss the actual behaviors of these two data flows individually.



(a) Layer 1 **(b)** Layer 2

(c) Layer 3 **(d)** Layer 4

Figure 14: The visualization of attention weights in each layer.

Table 7: Portion of the API-to-index mapping.

Index	API
0	Landroid/telephony/TelephonyManager; getDeviceId
11	Lorg/apache/http/client/HttpClient; execute
13	Landroid/media/MediaRecorder; setVideoSource
39	Landroid/net/wifi/WifiInfo; getSSID
65	Landroid/content/pm/PackageManager; getInstalledApplications
75	Lsun/net/www/protocol/http/HttpURLConnection; connect
77	Ljava/net/URL; openConnection
81	Landroid/os/Vibrator; vibrate
629	Landroid/telephony/gsm/SmsMessage; getMessageBody
630	Landroid/content/pm/PackageManager; getPackageInfo

According to Table 7, node 0 calls the API $Landroid/telephony/TelephonyManager;getDeviceId$. The API returns the unique device identifier, commonly used in the mobile communication network. Malware exploits this information to track users. According to Fig. 14a–c, node 0 does not pass information to other nodes. It means the framework only finds other evidence that the App utilizes the device ID. If we focus on tracing the upstream nodes with the highest attention, the derived data flow is as Table 8.

Table 8: The detailed information on the data flow with highest weights.

Node Idx.	The Corresponding APIs	Description
65	Landroid/content/pm/PackageManager; getInstalledApplications	Get the installed App list of the device.
75	Lsun/net/www/protocol/http/HttpURL Connection; connect	Establish connections with external servers.
13	Landroid/media/MediaRecorder; setVideoSource	Configure for screen recording.
11	Lorg/apache/http/client/HttpClient; execute	Execute an HTTP request.

From Table 8, we can infer that the application may exhibit two suspicious behaviors. First, the application collects the list of installed applications on the device and sends it to an external server. Second, the application prepares for screen recording and waits for instructions from an external server to initiate the recording. Regardless of the actual behavior, the evidence obtained is sufficient for researchers to conduct further analysis. Researchers can utilize the API information mentioned above to identify the code snippets in the Apps, thereby evaluating the detection results.

In summary, the design of this framework allows researchers to identify suspicious behaviors in the application. Since the suspicious behaviors consist of nodes and nodes represent APIs, researchers can obtain API information. Furthermore, by using this API information, researchers can make an initial evaluation of the suspicious behavior. Through this process, the framework achieves the result explainability. This section also demonstrates the process by an App.

6 Conclusion

Since the widespread of mobile devices, the Android system has been a prime target for malware. The transitional approach to detect the malware is not effective. Although the machine-learning-based approaches have achieved remarkable performance, the lack of explainability can raise concerns about the reliability of prediction results, blocking the machine-learning-based approaches from production use.

Hence, this research proposed an Android malware detection framework with explainability. This framework detects malicious behaviors in Apps by tracking data flows. It first performs static analysis on the App to extract the usage of APIs, such as the call sequences and the name of APIs. Then, it deploys a Skip-Gram model to learn the API features from the usage. Finally, it calculates the probability of the App being malicious and the related data flows in the App with a proposed GAT-based model. The framework also outputs the suspicious API call sequence as the explanation. Experiments show that this framework achieved 96.4% accuracy, 97.3% precision, 95.5% recall, and 96.4% F1 score. Moreover, an illustration shows the explanation helps researchers to know the App's suspicious behavior. However, this approach still has the following limitations:

- **Not Optimized for Model Structures:** In the experiment, we evaluated the impact of hyper-parameters and model structures separately. Then, we selected the best to build the model. However, due to time

restrictions, we did not conduct all hyper-parameter combinations with other model structures, such as evaluating other structures with varying layer numbers. Consequently, the model structures are not optimal.

- **Lack of Support for the External Library:** The Android system allows using C/C++ compiled external libraries for complex calculation speed-up or advanced functions. These external libraries can bypass most Android APIs and directly interact with the Linux kernel. Since our approach is Android API-based, our approach cannot detect applications' behaviors in the library.
- **Lack of Support for the Internal Communication Mechanism:** The Android system provides an additional mechanism for communication between the App's internal components. However, SACDDG cannot record the use of this mechanism. Hence, our approach may overlook applications that leverage this mechanism.

There are three directions to improve this framework. First, to optimize the performance, we can use techniques like neural architecture search to conduct the proper configuration of hyper-parameters. Second, we plan to extend our approach to C/C++ compiled libraries by leveraging recent advances in native-code reachability analysis for Android (e.g., DroidReach++ [22]), including selecting relevant Linux kernel APIs, constructing SACDDGs for the libraries, and integrating these library SACDDGs with the original SACDDG. Third, consider the communication between components as a special kind of edge in SACDDG, which has been proved feasible by Wu et al. [7].

Additionally, the core idea of this framework is to detect malicious Apps based on their system call data flows. The major challenge of porting the frameworks to other platforms is identifying the system calls and data flows. However, due to the variety of the platforms, some challenges exist. For example, PC allows more optimization options than Android, including embedding the system call in the binary. It may make the process of recognizing system calls harder. Therefore, verifying the effectiveness of this framework on other platforms is also one of the future directions of this research.

Acknowledgement: Not applicable.

Funding Statement: This work was supported in part by the National Science and Technology Council of Taiwan under Grant NSTC 114-2634-F-110-001-MBK, and in part by the Information Security Research Center at National Sun Yat-sen University, Taiwan. This work was also supported in part by the Ministry of Education, Science, Sports, and Culture, Grant-in-Aid for Scientific Research (C) 22K12038, Japan and in part by the Telecommunications Advancement Foundation (TAF) of Japan.

Author Contributions: Sheng-Feng Lu contributed to the conceptualization of the study, conducted the investigation and experiments, and prepared the original draft of the manuscript. Chun-I Fan provided overall supervision, acquired research funding, and performed major manuscript review and editing. Cheng-Han Shie contributed to manuscript review and editing, validation of the technical content, refinement of the manuscript, and provided technical guidance in network security. Ming-Feng Tsai contributed to manuscript review and editing and provided technical guidance in artificial intelligence. Tomohiro Morikawa contributed through discussions of technical details and manuscript review and editing. Takeshi Takahashi and Tao Ban provided expert consultation and professional advice. All authors reviewed and approved the final version of the manuscript.

Availability of Data and Materials: All data generated or analyzed during this study are included in this published article. The source code is available from the corresponding author upon reasonable request.

Ethics Approval: Not applicable.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Mobile operating system market share worldwide. [cited 2026 Feb 1]. Available from: https://gs.statcounter.com/os-market-share/mobile/worldwide.
2. 200,000 new mobile banking trojan installers discovered, double the 2021. 2023 [cited 2026 Feb 1]. Available from: https://www.kaspersky.com/about/press-releases/2023.
3. Shu L, Dong S, Su H, Huang J. Android malware detection methods based on convolutional neural network: a survey. IEEE Trans Emerg Top Comput Intell. 2023;7(5):1330–50. doi:10.1109/tetci.2023.3281833.
4. Senanayake J, Kalutarage H, Al-Kadri MO, Petrovski A, Piras L. Android source code vulnerability detection: a systematic literature review. ACM Comput Surv. 2023;55(9):187. doi:10.1145/3556974.
5. Meng Z, Xu F, Zhao W, Wang W, Huang W, Cui J, et al. MalFlows: context-aware fusion of heterogeneous flow semantics for android malware detection. IEEE Trans Dependable Secure Comput. 2026. doi:10.1109/tdsc.2026.3656972.
6. Hei Y, Yang R, Peng H, Wang L, Xu X, Liu J, et al. Hawk: rapid android malware detection through heterogeneous graph attention networks. IEEE Trans Neural Netw Learn Syst. 2024;35(4):4703–17. doi:10.1109/TNNLS.2021.3105617.
7. Wu Y, Shi J, Wang P, Zeng D, Sun C. DeepCatra: learning flow- and graph-based behaviours for android malware detection. IET Inf Secur. 2023;17(1):118–30. doi:10.1049/ise2.12082.
8. Gao H, Cheng S, Zhang W. GDroid: android malware detection and classification with graph convolutional network. Comput Secur. 2021;106(6):102264. doi:10.1016/j.cose.2021.102264.
9. Arslan RS, Tasyurek M. AMD-CNN: android malware detection via feature graph and convolutional neural networks. Concurr Comput. 2022;34(23):e7180. doi:10.1002/cpe.7180.
10. Amin M, Shah B, Sharif A, Ali T, Kim KI, Anwar S. Android malware detection through generative adversarial networks. Trans Emerg Telecomm Technol. 2022;33(2):e3675. doi:10.1002/ett.3675.
11. Zhou J, Cui G, Hu S, Zhang Z, Yang C, Liu Z, et al. Graph neural networks: a review of methods and applications. AI Open. 2020;1(1):57–81. doi:10.1016/j.aiopen.2021.01.001.
12. Velickovic P, Cucurull G, Casanova A, Romero A, Liò P, Bengio Y. Graph attention networks. arXiv:1710.10903. 2017.
13. Alharbi FA, Alghamdi AM, Alghamdi AS. A systematic review of android malware detection techniques. Int J Comput Sci Secur. 2021;15(1):1–19. doi:10.14569/ijacsa.2015.060120.
14. Pektaş A, Acarman T. Learning to detect android malware via opcode sequences. Neurocomputing. 2020;396(99):599. doi:10.1016/j.neucom.2018.09.102.
15. Mathur A, Podila LM, Kulkarni K, Niyaz Q, Javaid AY. NATICUSdroid: a malware detection framework for android using native and custom permissions. J Inf Secur Appl. 2021;58(1):102696. doi:10.1016/j.jisa.2020.102696.
16. Zhu D, Xi T, Jing P, Wu D, Xia Q, Zhang Y. A transparent and multimodal malware detection method for android apps. In: Proceedings of the 22nd International ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems, MSWIM '19. New York, NY, USA: Association for Computing Machinery; 2019. p. 51–60.
17. Taheri R, Ghahramani M, Javidan R, Shojafar M, Pooranian Z, Conti M. Similarity-based android malware detection using hamming distance of static binary features. Future Gener Comput Syst. 2020;105(6):230. doi:10.1016/j.future.2019.11.034.
18. Ma Z, Ge H, Wang Z, Liu Y, Liu X. Droidetec: android malware detection and malicious code localization through deep learning. arXiv:2002.03594. 2020.
19. Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, et al. PyTorch: an imperative style, high-performance deep learning library. In: Proceedings of the 33rd International Conference on Neural Information Processing Systems. Red Hook, NY, USA: Curran Associates Inc.; 2019. p. 8026–37.
20. Allix K, Bissyandé TF, Klein J, Le Traon Y. AndroZoo: collecting millions of android apps for the research community. In: Proceedings of the 13th International Conference on Mining Software Repositories, MSR '16. New York, NY, USA: Association for Computing Machinery; 2016. p. 468–71.

21. He Y, Li Y, Wu L, Yang Z, Ren K, Qin Z. MsDroid: identifying malicious snippets for android malware detection. IEEE Trans Dependable Secure Comput. 2023;20(3):2025. doi:10.1109/tdsc.2022.3168285.
22. Borzacchiello L, Cornacchia M, Maiorca D, Giacinto G, Coppa E. DroidReach++: exploring the reachability of native code in android applications. Comput Secur. 2025;159(4):104657. doi:10.1016/j.cose.2025.104657.