The experimentation with different CNN architectures, including ResNet and DenseNet variants, highlighted the strengths and limitations of each approach. For instance, ResNet50 consistently outperformed other models in terms of accuracy, precision, and F1 score, suggesting its superior ability to learn complex features from the datasets. On the other hand, models like AlexNet and VGG, while still effective, showed slightly lower performance, indicating potential areas for further optimization.

Furthermore, the use of data augmentation techniques was crucial in enhancing the generalizability of the models, ensuring robust performance across varied and unseen samples. This aspect is particularly important in real-world scenarios where the diversity of malware is constantly evolving. Overall, the findings from this study not only validate the efficacy of CNNs in malware detection but also emphasize the importance of careful hyperparameter tuning and architecture selection. Future work could further investigate the integration of additional data sources and advanced augmentation techniques to continue improving model performance and adaptability.

Chapter 9

A Graph Neural Network Based Malware Detection Method

9.1 Introduction

IT-enabled services are deeply integrated into every aspect of our society. With the rapid growth of technology and IT-enabled services, the lifestyles of people have changed. People make use of services available on top of the Internet for numerous purposes. Unfortunately, these services can be exploited by enemies and cyber criminals to fulfill their ulterior motives. Attackers can launch various types of attacks using malicious software called malware to break into interconnected systems by exploiting the vulnerabilities found in devices, systems, and software applications. As a result, they can disrupt the ongoing services, block access to critical services, steal sensitive information, corrupt important files and exhaust the resources of servers. Malware are continuously evolving with the change in technology. Day by day, cyber-attacks involving malware are targeting systems with continuously improved evading techniques. The methods used in cyber attacks are becoming more complex with enhanced stealth, and frequencies of attacks. Cyber threats are not limited to computer networks where all devices are desktop computers or mobile devices. They are moving into Internet-of-Things (IoT) and critical infrastructure networks. Attackers can now compromise IoT devices to become a part of a zombie network to launch targeted attacks. Malware encompasses a wide range of malicious code designed to compromise the security, integrity, and functionality of computer systems. The proliferation and sophistication of malware pose significant threats to personal, industrial as well as national security. The rapidly evolving nature of malware like ransomware necessitates robust and adaptive detection systems to ensure the safety of digital environments.

Traditional anti-malware solutions are mainly based on signature-based detection techniques, which rely on the analysis and comparison of malware attack signatures to a list of pre-identified signatures. However, this method of traditional detection methods cannot effectively detect unknown malware variants, such as zero-day malware. In contrast, machine learning-based methods and deep learning-based methods have emerged as widely used approaches for the detection of malware variants. While machine learning-based methods have demonstrated good predictive performance, the utilization of deep learning models such as Convolutional Neural Networks [105] (CNN), and Graph Neural Networks [106] (GNN) in malware detection have further improved the predictive accuracy of malware detection.

Recently, GNN-based models have garnered significant attention and achieved promising results based on the extracted high-level structure features. Unlike traditional neural networks that operate on fixed-size inputs, GNNs are designed to work with data represented as graphs, where entities and their relationships are naturally encapsulated. This capability makes GNNs particularly well-suited for malware detection, as the behavior and structure of malicious software can be effectively captured and analyzed through graph representations.

9.1.1 Motivation

The escalating frequency and sophistication of malware attacks pose a severe and persistent threat to individuals, organizations, and national security infrastructures worldwide. As malware authors employ increasingly advanced techniques to create and deploy malware, traditional malware detection systems, primarily reliant on signature-based methods, are becoming less effective. These conventional approaches often fail to recognize new and polymorphic malware variants that do not match known signatures, leaving systems vulnerable to zero-day attacks. To address these limitations, there is a pressing need for innovative and more resilient detection methodologies. Machine learning has shown considerable promise in enhancing malware detection by learning patterns and behaviors from data. However, many machine learning techniques struggle to capture the complex and dynamic relationships inherent in malware behavior. By utilizing GNNs, which are specifically designed to operate on graph-structured data, we can capture these relationships more accurately and robustly than traditional machine learning methods. The motivation behind this research is to harness the power of GNNs to develop a more effective and adaptive malware detection system.

9.1.2 Contribution

In this chapter, we explore a novel approach to malware detection utilizing Graph Neural Networks (GNNs). The goal of this work is to investigate malware classification techniques using graph-based learning, relying solely on graphs generated from raw malware binaries, without typical features such as permissions and API calls. By representing malware as graphs, where nodes signify various function calls and edges denote the interactions between them, we leverage the strengths of GNNs to uncover intricate patterns and behaviors indicative of malicious activity. This method enables a more nuanced and comprehensive analysis compared to traditional techniques, which often miss subtle and complex relationships within the data.

9.2 Background

9.2.1 Graph

A graph is a mathematical and data structure concept used to represent relationships or connections between objects. It consists of two main components: nodes (or vertices) and edges. Graphs are often used to model and analyze various real-world problems and systems, such as social networks, transportation networks, and computer networks. Nodes represent individual entities or points in the graph, each with specific characteristics. Edges are the connections between nodes and are typically represented by lines or arcs. Edges can be directed or undirected. In a directed graph, an edge from vertex u to vertex v indicates a one-way relationship from u to v. In an undirected graph, an edge between vertices u and v indicates a two-way relationship between them. Some graphs have weights associated with their edges to represent additional information, such as distances or costs [107].

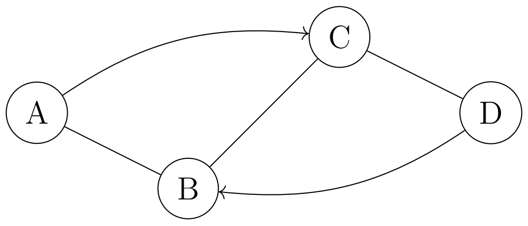


Figure 9-1: A simple graph with directed and undirected edges

9.2.2 Function Call Graph

A Function Call Graph (FCG) is a directed graph $G = (N, E)$, where N is a set of nodes representing functions and E represents the set of inter-procedural calls. An FCG shows the relationships and dependencies between functions or subroutines in a software program. It illustrates how functions call each other within the program, which is valuable for understanding program flow, identifying performance bottlenecks, and debugging. In an FCG, functions are represented as nodes, and the edges between nodes represent the flow of control from one function to another through function calls [108]. For example, when an app sends an SMS message, it performs a series of API calls on the Android platform. An FCG consists of all possible execution paths called during its runtime. For instance, a function to gather sensitive information might call other functions to access phone contacts, SMS, browser bookmarks, etc., and then use another function to send the data back to an attacker.

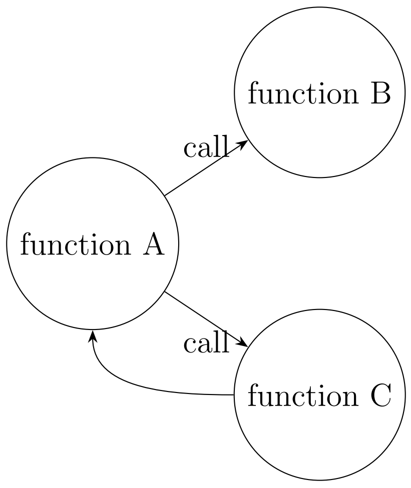


Figure 9-2: Function Call Graph

9.2.3 Graph Neural Networks

Graph Neural Networks (GNNs) are a type of machine learning model that can learn from and make predictions on graph data. Graphs provide a powerful way to represent data with relationships between different entities, such as social networks, road networks, and molecular structures. GNNs can solve a wide range of tasks, including node classification, edge prediction, graph classification, and graph generation.

GNNs work by iteratively aggregating information from neighboring nodes in a graph. In each iteration, each node updates its representation based on the representations of its neighbors. This process allows GNNs to learn global patterns in the graph from local information. This is typically done through a series of layers, similar to traditional neural networks, where each layer refines the node representations, enabling the network to capture complex patterns and dependencies in the graph structure.

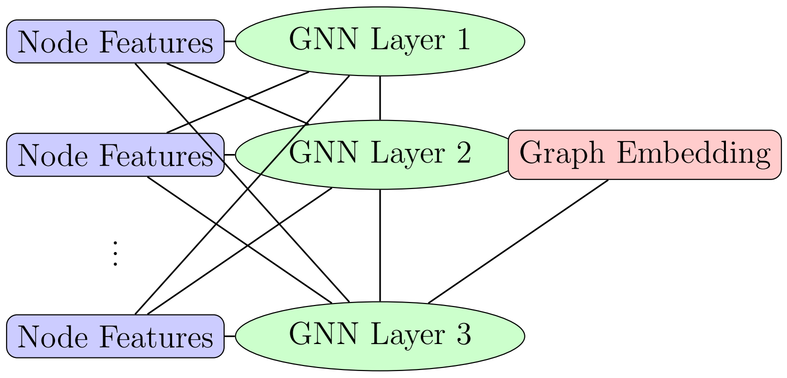


Figure 9-3: Graph Neural Network Architecture

9.2.3.1 Graph Convolutional Networks

Graph Convolutional Networks [109] (GCNs) are a type of GNN used to learn from and make predictions on graph data. GCNs iteratively aggregate information from neighboring nodes in a graph. In each iteration, each node updates its representation based on the representations of its neighbors. This process allows GCNs to learn global patterns in the graph from local information.

GCNs have achieved state-of-the-art results on various graph-related tasks, such as node classification, edge prediction, graph classification, and graph generation. Let $G = (V, E)$ be a graph, where V represents the set of nodes and E represents the set of edges. Each node v_i is associated with a feature vector X_i. The goal of a GCN is to learn a new feature representation H_i for each node v_i by aggregating information from its neighbors, expressed as:

$$H_i = \sigma\left(\sum_{j \in N(v_i)} \frac{1}{c_{ij}} \cdot W X_j\right) \tag{9.1}$$

Here, $N(v_i)$ represents the neighbors of node v_i, W is a learnable weight matrix, σ is an activation function (e.g., ReLU), and c_{ij} is a normalization factor. This process is typically performed iteratively through multiple layers to capture increasingly complex relationships in the graph.

9.2.3.2 Graph Isomorphism Network

Graph Isomorphism Networks [110] (GINs) are designed for graph classification and graph matching. GINs learn graph representations invariant to graph isomorphism, meaning they can determine whether two graphs are structurally equivalent. Let $G = (V, E)$ be a graph, where V represents the set of nodes and E represents the set of edges. The goal of a GIN is to learn a representation h_G for the entire graph G. This is done by iteratively updating node representations based on the aggregation of features from their neighbors and then applying a readout function.

$$h_v^{(k+1)} = \mathrm{MLP}^{(k)}\left(\left(1 + \epsilon^{(k)}\right) \cdot h_v^{(k)} + \sum_{w \in N(v)} h_w^{(k)}\right) \tag{9.2}$$

Here, $h_v^{(k)}$ represents the feature representation of node v at iteration k, $N(v)$ represents the neighbors of node v, $\epsilon^{(k)}$ is a learnable parameter, and $\mathrm{MLP}^{(k)}$ is a multi-layer perceptron applied to update node features. The readout function aggregates the node representations to obtain the graph representation h_G.

9.2.3.3 Graph Attention Network (GAT)

Graph Attention Networks [111] (GATs) extend the concept of self-attention to graphs. GATs compute a new feature representation $h_v^{(l+1)}$ for each node v at layer $l + 1$ based on the features of its neighbors and dynamically computed attention weights.

$$h_v^{(l+1)} = \sigma \left(\sum_{u \in N(v)} \alpha_{vu}^{(l)} \cdot W^{(l)} h_u^{(l)} \right) \tag{9.3}$$

Here, $h_v^{(l)}$ represents the feature representation of node v at layer l, and $\alpha_{vu}^{(l)}$ are attention weights computed using a learnable function that considers the features of connected nodes and a shared attention parameter vector. GAT's ability to capture adaptive relationships among nodes makes it popular for graph-based tasks.

9.2.3.4 GraphSAGE (Graph Sample and Aggregated)

GraphSAGE [112] is a versatile GNN model supporting inductive learning for various graph-based tasks. GraphSAGE defines a node embedding function by first sampling a fixed-size set of neighbors $\mathcal{N}(v_i)$ for a target node v_i and then aggregating their feature representations. The aggregation operation is defined as:

$$h_v^{(l+1)} = \text{AGGREGATE} \left(\left\{ h_u^{(l)}, \forall u \in \mathcal{N}(v_i) \right\} \right) \tag{9.4}$$

The final embedding $h_v^{(l+1)}$ is obtained by projecting the aggregated representation through a neural network function:

$$h_v^{(l+1)} = \sigma \left(W^{(l)} \cdot h_v^{(l+1)} \right) \tag{9.5}$$

GraphSAGE's ability to generalize to unseen nodes and its adaptability to various aggregation functions make it a popular choice for graph-based machine-learning tasks.

9.2.4 Node Feature Generation

In GNN-based malware detection, generating effective node features is crucial for capturing the structural and functional characteristics of nodes within a control flow graph (CFG). Key features include centrality measures like degree centrality, betweenness centrality, closeness centrality, and eigenvector centrality, which help identify the importance and influence of functions in the execution flow. Additionally, Local Proximity Distribution (LPD) captures local structural properties by examining the distribution of distances to a node's neighbors. Other valuable metrics include node degree, PageRank, Graphlet Degree Vector (GDV), and clustering coefficient, each providing insights into a node's connectivity, influence, and local topology. These features collectively enable the GNN to learn patterns indicative of benign or malicious behavior, enhancing the model's accuracy and effectiveness in malware detection.

9.2.5 Types of Tasks in Graph Learning

Graph learning focuses on understanding and analyzing complex structures and relationships in graph data. It includes three main task categories: node-level, edge-level, and graph-level, each with specific objectives and methodologies.

a. Node-level tasks: These tasks focus on individual nodes within the graph.

   - *Node classification*: This task aims to assign each node to a predefined class based on its features and its position within the graph.
   - *Node regression*: Instead of classifying nodes, this task predicts a continuous value for each node, such as a score or measurement.
   - *Node clustering*: The goal here is to group nodes into clusters such that nodes within the same cluster are more similar to each other than to those in other clusters. This can be useful for identifying communities or similar entities within the graph.

b. Edge-level tasks: These tasks involve the edges, or connections, between nodes.

   - *Edge classification*: This task involves determining the type or label of each edge. For example, in a social network, edges could represent different types of relationships such as friends, family, or colleagues.
   - *Link prediction*: The aim here is to predict whether an edge should exist between two nodes. This can be used for recommending new connections in social networks or identifying potential interactions in biological networks.

c. Graph-level tasks: These tasks consider the graph as a whole.

   - *Graph classification*: This task involves assigning an entire graph to a predefined class. For example, different graphs could represent different types of molecules, and the task would be to classify them based on their structure.

- *Graph regression*: Similar to node regression, but for entire graphs. The task is to predict a continuous value for each graph, such as the graph's overall activity or property.
- *Graph matching*: This task involves comparing two graphs to determine their similarity or correspondence. This can be useful in applications like finding similar molecules in drug discovery or matching patterns in social networks.

9.2.6 Loss Function

A suitable loss function for multiclass classification tasks is essential in GNN models for function call graph classification. Cross-entropy loss [113], also known as Log Loss or Softmax Loss, is commonly used for multiclass classification issues. It measures the dissimilarity between predicted class probabilities and true class labels. Given N samples, K classes, predicted class probabilities $\log(P(y_i = j|x_i))$ for each sample i and class j, and true labels y_i, the Cross-Entropy Loss is defined as:

$$\text{Cross-Entropy Loss} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{K}(y_{i,j}\log(P(y_i = j|x_i))) \tag{9.6}$$

Here,

- N is the number of samples.
- K is the number of classes.
- $y_{i,j}$ is an indicator variable that is 1 if the true class of sample i is j and 0 otherwise.
- $P(y_i = j|x_i)$ is the predicted probability of sample i belonging to class j based on the GNN model.

9.3 Problem Statement

The problem at hand is to develop an effective and efficient malware classification system using Graph Neural Networks (GNNs). Given a dataset of executable files represented as control flow graphs, the objective is to classify each file as either benign or malicious. Formally, we can define the problem as follows:

Let $\mathcal{G} = \{G_1, G_2, \ldots, G_N\}$ be a dataset of control flow graphs, where N is the number of samples. Each control flow graph G_i represents an executable file. We aim to design a GNN model $\mathcal{M}$ that can predict the binary label y_i for each control flow graph, where y_i indicates whether the file is benign ($y_i = 0$) or malicious ($y_i = 1$). Mathematically, the classification problem can be stated as:

$$\mathcal{M} : G_i \mapsto y_i, \quad \forall G_i \in \mathcal{G}$$

The goal is to train $\mathcal{M}$ to achieve high accuracy in malware classification while ensuring that the model generalizes well to unseen malware samples.

9.4 Proposed Method

The proposed model for the detection of malware is illustrated in Figure 9-4. The model aims to classify a given data into two categories, malware, and goodware with minimum false alarms.

9.4.1 Function Call Graph Generation

The process of generating the Function Call Graph (FCG) from an APK file starts with decompiling the APK using the Androguard tool to access the embedded DEX files that hold the executable bytecode as shown in Figure 9-6. This step is crucial as it lays the foundation for the entire analysis by providing the raw data necessary for graph construction. Following decompilation, the DEX files are meticulously analyzed to extract details about the methods and their interactions. These interactions are critical for constructing the call graph as they define the edges between nodes, where each node represents a method within the application. Utilizing Androguard's robust analysis capabilities, systematically map out the relationships between these methods, thus forming the backbone of the FCG. Figure 9-5 presents example of Function Calls for various methods, highlighting the interaction between methods across different classes. It showcases the relationships and data flow through these function call graphs, illustrating sequences and dependencies among the methods. This graph is then program-

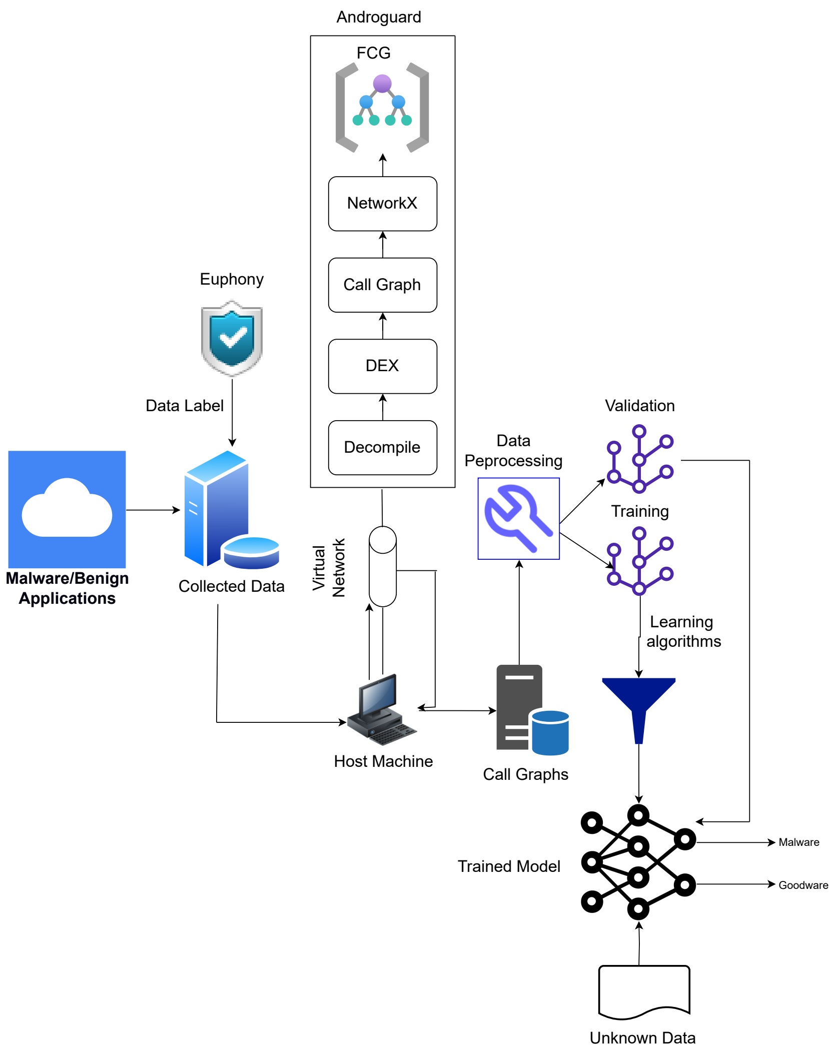
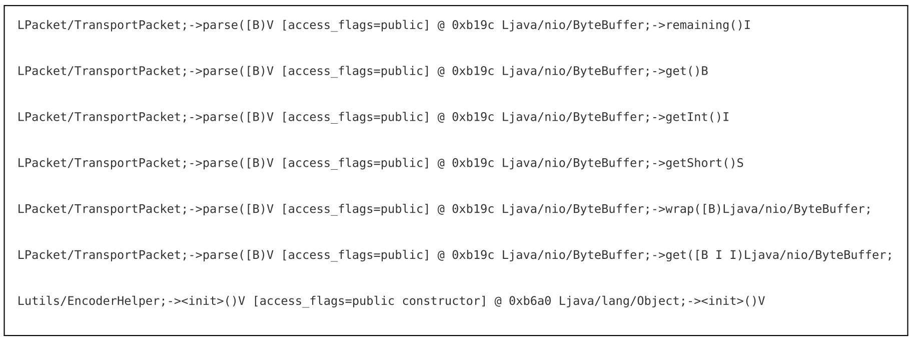


Figure 9-4: Overview of the proposed method

matically constructed using the NetworkX library, ensuring a structured format for visualization. Additionally, the graph is saved as an edge list file, serving as a crucial input for subsequent analysis using Graph Neural Networks (GNN). Furthermore, to streamline this process, a web-based tool has been developed to automate the generation of the FCG. The interface of this tool and the output of the generated FCG are illustrated in Figures 9-8 and 9-9 respectively. Additionally, for this study, an FCG dataset comprising 21000 instances(20000 malware and 1000 goodware) has been created and made available to the public, enhancing the accessibility and applicability of this research. The algorithm for function call graph generation is shown in Algorithm 4.



```
LPacket/TransportPacket;->parse([B)V [access_flags=public] @ 0xb19c Ljava/nio/ByteBuffer;->remaining()I

LPacket/TransportPacket;->parse([B)V [access_flags=public] @ 0xb19c Ljava/nio/ByteBuffer;->get()B

LPacket/TransportPacket;->parse([B)V [access_flags=public] @ 0xb19c Ljava/nio/ByteBuffer;->getInt()I

LPacket/TransportPacket;->parse([B)V [access_flags=public] @ 0xb19c Ljava/nio/ByteBuffer;->getShort()S

LPacket/TransportPacket;->parse([B)V [access_flags=public] @ 0xb19c Ljava/nio/ByteBuffer;->wrap([B)Ljava/nio/ByteBuffer;

LPacket/TransportPacket;->parse([B)V [access_flags=public] @ 0xb19c Ljava/nio/ByteBuffer;->get([B I I)Ljava/nio/ByteBuffer;

Lutils/EncoderHelper;-><init>()V [access_flags=public constructor] @ 0xb6a0 Ljava/lang/Object;-><init>()V
```

Figure 9-5: Example of Function Calls

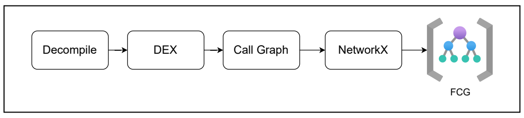
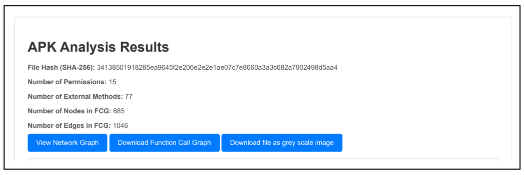


Figure 9-6: Process of binary to FCG generation



APK Analysis Results

File Hash (SHA-256): 34138501918285ea9645f2e206e2e2e1ae07c7e8660a3a3c682a7902498d5aa4

Number of Permissions: 15

Number of External Methods: 77

Number of Nodes in FCG: 685

Number of Edges in FCG: 1046

View Network Graph
Download Function Call Graph
Download file as grey scale image

Figure 9-7: Application analysis results

Algorithm 4: Process APKs and Generate Call Graphs

```
Input: list_of_apks
Output: list_of_call_graphs
forall apk in list_of_apks do
    a, d, dx ← AnalyzeAPK(apk);
    call_graph ← dx.get_call_graph();
    node_to_number ← Encode each function name in call_graph to a
      unique number;
    numbered_graph ← label call_graph using node_to_number;
    return (numbered_graph, call_graph);
end
```

Table 9.1 displays the snapshot of the Function Call Graph of Airpush malware, including the source and target functions along with descriptions of the calls.

Table 9.1: Snapshot of Function Call Graph (FCG) for Airpush Malware

Source Function	Target Function	Description
HttpPostDataTask.<init> (Context, List, String, Listener)	Util.printDebugLog(String)	Constructor of HttpPostDataTask calls the static method printDebugLog in the Util class.
HttpPostDataTask.<init> (Context, List, String, Listener)	AsyncTask.<init>()	Constructor of HttpPostDataTask calls the constructor of AsyncTask from Android OS.
HttpPostDataTask.<init> (Context, List, String, Listener)	StringBuilder.toString()	HttpPostDataTask constructor includes a call to StringBuilder's toString method.
Airpush$3.onTaskComplete(String)	AsyncTaskCompleteListener .lauchNewHttpTask()	onTaskComplete method in Airpush$3 calls launchNewHttpTask in AsyncTaskCompleteListener.

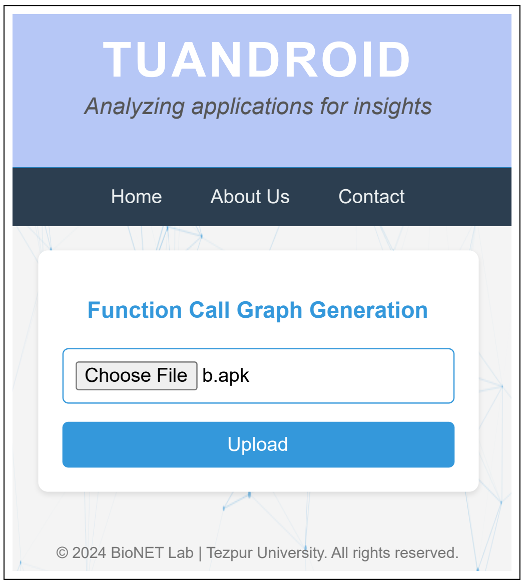

TUANDROID
Analyzing applications for insights
Home
About Us
Contact
Function Call Graph Generation
Choose File
b.apk
Upload
© 2024 BioNET Lab | Tezpur University. All rights reserved.

Figure 9-8: Web-based tool for binary to FCG conversion

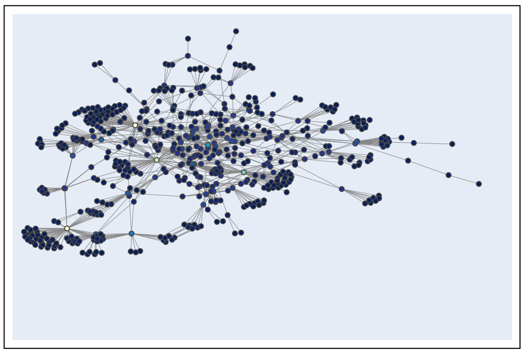


Figure 9-9: Example of malware binary data visualized as a FCG

9.4.2 GNN method

The proposed method aims to develop a Graph Neural Network (GNN) architecture specifically designed for control flow graphs of malicious files, with the goal of enhancing malware detection and classification. Control flow graphs, which illustrate the execution pathways of binary files, are ideal for graph-based analysis due to their comprehensive representation of program behaviors. In this study, we investigate three distinct GNN architectures—Graph Convolutional Network (GCN), GraphSAGE, and Graph Isomorphism Network (GIN)—selected for their proven effectiveness in handling graph data and their potential to accurately detect and classify malware.

The proposed GNN architecture consists of the following components.

1. Feature Extractor: This component extracts relevant features from the control flow graph of a malicious file.

2. Convolutional Layers: The extracted features are processed through multiple convolutional layers to capture hierarchical patterns and relationships within the graph structure.

3. Fully Connected Layer: After feature extraction and graph convolution, the data is passed through a fully connected layer to further refine the representations.

4. Output Layer: The final layer produces binary classification results,

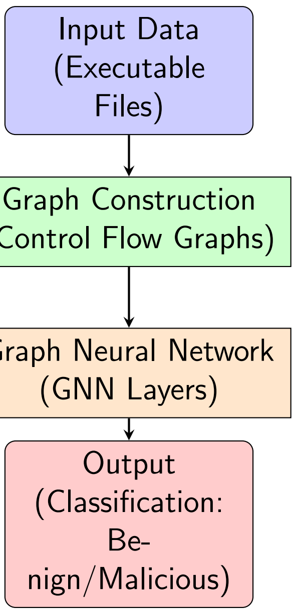

Input Data (Executable Files)
Graph Construction (Control Flow Graphs)
Graph Neural Network (GNN Layers)
Output (Classification: Benign/Malicious)

Figure 9-10: Pipeline of a GNN Model for Malware Detection determining whether the file is malicious or benign.

The GNN consists of multiple layers, each with its own operations. We consider the following components:

- **Node Embedding**: In the first layer, we initialize node embeddings based on their features.
- **Message Passing**: We perform message passing between neighboring nodes to capture dependencies. The aggregation function can be a weighted sum or attention mechanism.
- **Graph Pooling**: We apply graph pooling to reduce graph size while retaining important information.
- **Classification Head**: The final layer includes a classification head with softmax activation to predict whether the CFG represents a malicious or benign file.

The FCGs are featureless, meaning they do not contain any node or edge features. In this method, the graph node feature initialization methods listed below are used.

- Degree Centrality Initialization: It measures the number of edges connected to a node, indicating its level of connectivity within the graph. To initialize node features based on degree centrality, the following method is used.

1. Calculate Degree Centrality $C_d(v)$: For each node v in the graph G, calculate the degree centrality using the formula:

$$C_d(v) = \frac{\text{Degree of } v}{\text{Maximum Degree in } G} \tag{9.7}$$

2. Normalize Degree Centrality Scores: Normalize the degree centrality scores to a range between 0 and 1 by dividing each $C_d(v)$ by the maximum degree centrality score in the graph.
3. Initialize Node Features $X(v)$: Based on the normalized degree centrality scores, initialize the node features as follows:

$$X(v) = \begin{cases} [1,0] & \text{if } C_d(v) \text{ is high} \\ [0,1] & \text{if } C_d(v) \text{ is low} \\ [0.5,0.5] & \text{otherwise} \end{cases} \tag{9.8}$$

- Betweenness Centrality Initialization: It quantifies the extent to which a node lies on the shortest paths between other nodes in the graph. Nodes with high betweenness centrality are critical for information flow. To initialize node features based on betweenness centrality:

1. Calculate the betweenness centrality $C_b(v)$ for each node v in the graph G. The betweenness centrality $C_b(v)$ of a node v in a graph G is given by:

$$C_b(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{9.9}$$

where:

   - σ_{st} is the total number of shortest paths from node s to node t.
   - $\sigma_{st}(v)$ is the number of those shortest paths that pass through node v.

2. Normalize the betweenness centrality scores to a range between 0 and 1 by dividing each $C_b(v)$ by the maximum betweenness centrality score in the graph.
3. Initialize the node features X as follows:

$$X(v) = \begin{cases} [1,0] & \text{if } C_d(v) \text{ is high} \\ [0,1] & \text{if } C_d(v) \text{ is low} \\ [0.5,0.5] & \text{otherwise} \end{cases} \tag{9.10}$$

- Eigenvector Centrality Initialization: Eigenvector centrality measures a node's importance based on its connections to other important nodes. It assigns higher scores to nodes connected to other nodes with high centrality. To initialize node features based on eigenvector centrality:

  1. Calculate the eigenvector centrality $C_e(v)$ for each node v in the graph G. The eigenvector centrality $C_e(v)$ for each node v in the graph G is given by:

     $$C_e(v) = \frac{1}{\lambda_{\max}} \sum_{u \in N(v)} A_{vu} C_e(u) \tag{9.11}$$

     where $\lambda_{\max}$ is the largest eigenvalue of the adjacency matrix A, $N(v)$ denotes the set of neighbors of node v, and A_{vu} is the element of the adjacency matrix representing the edge between nodes v and u.

  2. Normalize the eigenvector centrality scores to a range between 0 and 1 by dividing each $C_e(v)$ by the maximum eigenvector centrality score in the graph.

  3. Initialize the node features X as follows:

     $$X(v) = [C_e(v), 1 - C_e(v)]$$

     Here, $C_e(v)$ represents the eigenvector centrality score of node v, and $[C_e(v), 1 - C_e(v)]$ reflects the relative importance of the node in terms of eigenvector centrality.

9.5 Experimental Results

We implement all Graph Neural Network (GNN) models using PyTorch 1.9.0, specifically leveraging the PyTorch Geometric library. The experiments are performed on a Dell Precision 7810 Tower with 2x Intel Xeon E5-2600 v3 consisting of 8 cores, 64GB RAM, and NVIDIA Tesla K80 GPU with 12GB VRAM.

9.5.1 Dataset

For the evaluation, one relevant publicly available Android malware dataset, consisting of different types of malware categories/families is used. Malnet an Android malware FCG dataset that was created by Scott et al.[26] from Georgia Tech University and the Microsoft APT team. The dataset consists of 4,500 malicious FCGs, belonging to four different malware categories, and 500 benign FCGs. The dataset is split into training, validation, and test sets, ensuring that each contains a representative mix of malicious and benign FCGs.

The second dataset TUANDROMD-FCG was generated using a specific process detailed in Section 9.4.1. To construct this dataset, 20000 samples of raw malware binaries are obtained from [34]. Additionally, we curated a set of the top 1,000 Android applications from Google Play, which served as representative examples of benign applications. The dataset is categorized into 72 distinct classes, with 71 classes dedicated to various types of malware and one class for goodware. This approach allowed us to create a dataset that encompasses a diverse range of data points, including both malicious and benign software instances. The detailed dataset statistics are presented in Table 9.2. To provide a visual representation of the data distribution, Figure 9-11 illustrates the overall class distribution, while Figure 9-12 shows the top 10 categories within the dataset.

Table 9.2: Dataset Statistics

Characteristic	**Count**
Total Instances	21,000
Malware Instances	20,000
Goodware Instances	1,000
Total Classes	72
Malware Classes	71
Goodware Class	1

9.5.2 Results

In this section, the outcomes of the experimentation with Graph Neural Network (GNN) models for malware detection are presented. Three distinct GNN architectures were employed, each configured with various hyperparameters to comprehensively assess their performance in detecting malware. The models tested include the Graph Convolutional Network (GCN), GraphSAGE, and Graph Isomorphism Network (GIN).

The results in Table 9.3 for Dataset 1 show the performance of three Graph Neural Network (GNN) models: GCN, GraphSAGE, and GIN. To optimize these models, hyperparameter tuning was performed using the latest techniques, including Bayesian Optimization [114] and Grid Search [115]. This tuning process aimed to

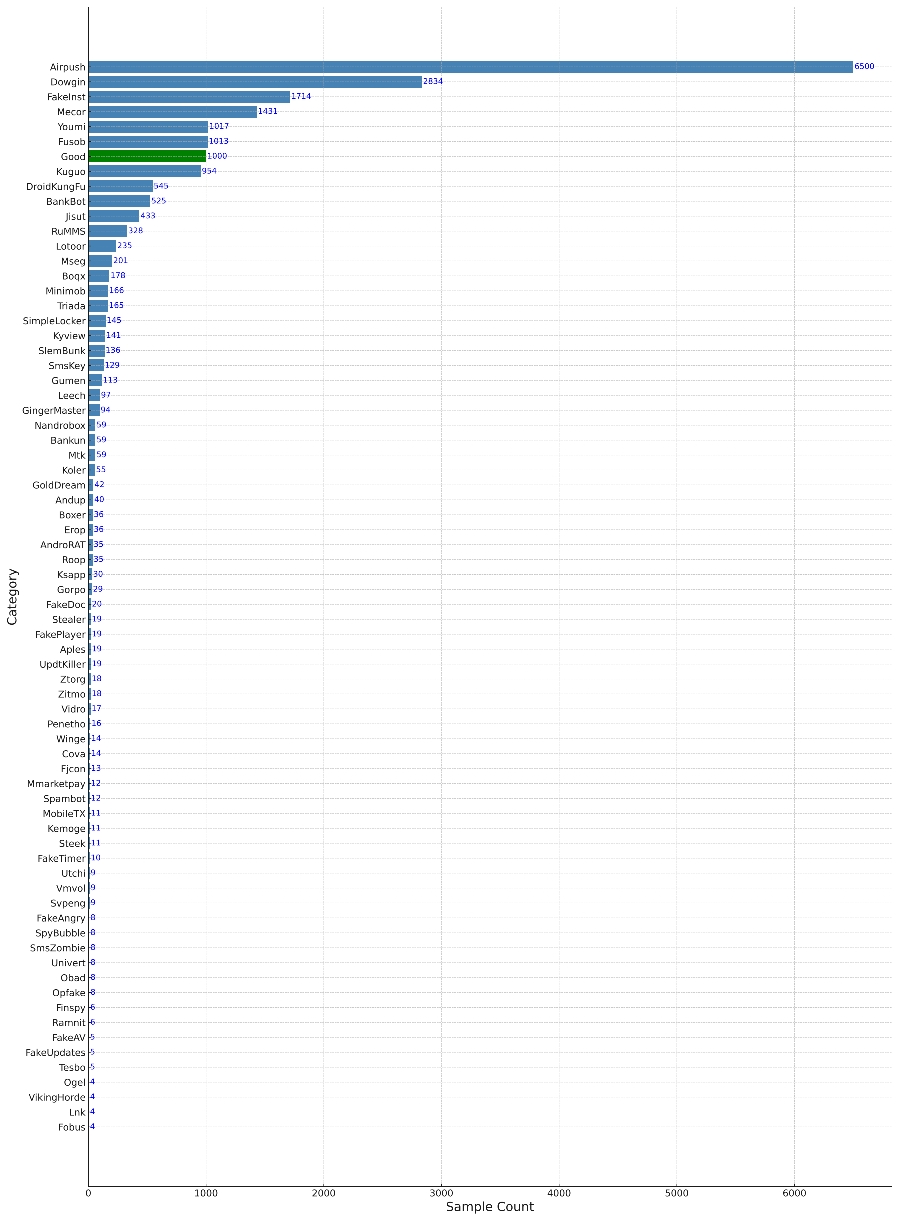


Figure 9-11: Class Distribution of TUANDROMD-FCG

identify the best configurations for each model. As a result, GraphSAGE achieved the highest validation accuracy at 0.9511, along with high precision (0.9375), recall (0.9610), and an F1 score of 0.9491. GIN followed with a validation accuracy of 0.9207, precision of 0.898, recall of 0.928, and an F1 score of 0.914. GCN had the lowest performance, with a validation accuracy of 0.768, precision of 0.729, recall of 0.7590, and an F1 score of 0.7399. These results highlight the importance of both model selection and effective hyperparameter tuning in GNN-based malware detection.

Table 9.4 presents the performance metrics of the GCN, GraphSAGE, and GIN models on Dataset 2, detailing accuracy, precision, recall, and F1 score. Hyperparameter selection played a crucial role in achieving these results, utilizing advanced

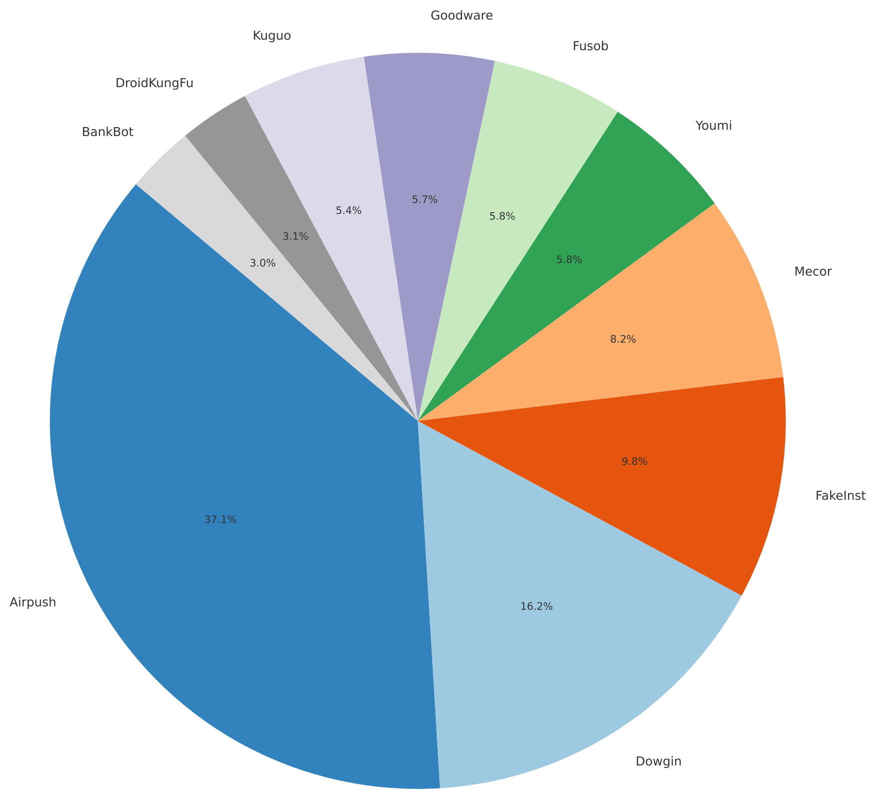


Figure 9-12: Top 10 categories of TUANDROMD-FCG

Table 9.3: Performance of GNN Models on Dataset 1

Model	Accuracy	Precision	Recall	F1 Score
GCN	0.768	0.729	0.7590	0.7399
Graph SAGE	0.9511	0.9375	0.9610	0.9491
GIN	0.9207	0.898	0.928	0.914

tuning techniques such as Bayesian Optimization and Grid Search to identify the optimal configurations for each model. GraphSAGE exhibited the highest performance, achieving an accuracy of 0.9719, precision of 0.94, recall of 0.94, and an F1 score of 0.92, highlighting its robust overall performance. GIN followed closely with an accuracy of 0.907, precision of 0.89, recall of 0.897, and an F1 score of 0.899, demonstrating strong performance, though slightly below that of GraphSAGE. GCN showed the lowest performance among the three, with an accuracy of 0.887, precision of 0.829, recall of 0.864, and an F1 score of 0.8300. These results underscore the importance of model selection and effective hyperparameter tuning in maximizing the performance of GNN-based malware detection models.

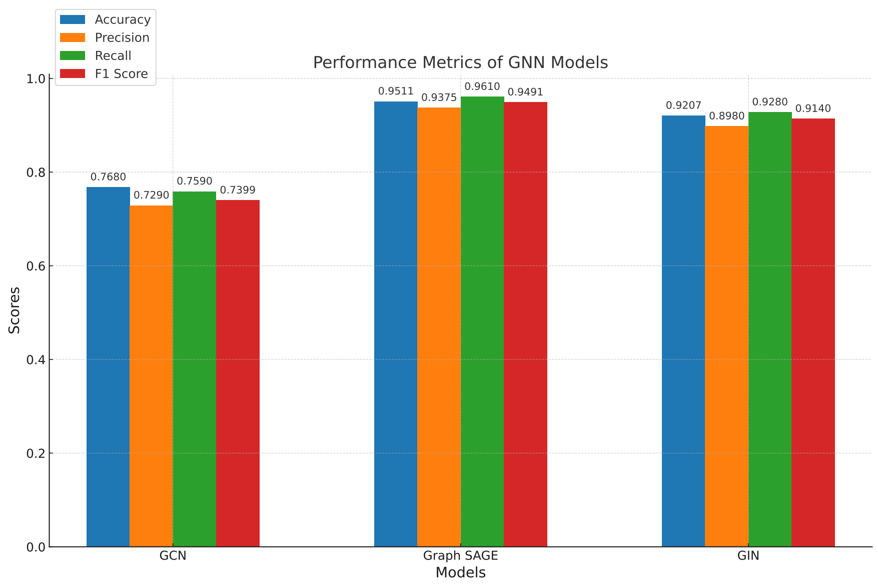


Figure 9-13: Performance of GNN Models on Dataset 1

Table 9.4: Performance of GNN Models on Dataset 2

Model	Accuracy	Precision	Recall	F1 Score
GCN	0.887	0.829	0.864	0.8300
Graph SAGE	0.9719	0.94	0.94	0.923
GIN	0.907	0.89	0.897	0.899

Table 9.5: Method Accuracy Comparison

Method	Accuracy
Feather [26]	86.00%
DGCNN [116]	92.18%
GIN [26]	90.00%
GCN [26]	81.00%
SGC [116]	90.79%
UnetGraph [116]	95.81%
NoG [26]	77.00%
Slaq-VNGE [26]	53.00%
GCN-JK [117]	89.70%
GIN-JK [117]	90.00%
GraphSAGE	95.11%
GIN	92.07%
GCN	76.80%

Table 9.5 highlights the performance of various graph-based neural network models. The models used in our work, namely GraphSAGE, GIN, and GCN, demon-

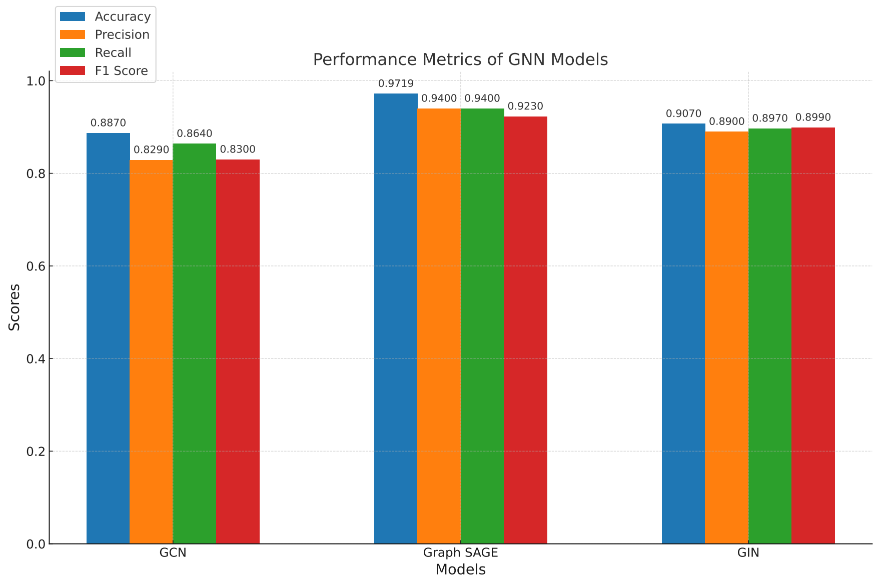


Figure 9-14: Performance of GNN Models on Dataset 2

strate competitive performance compared to other methods. GraphSAGE achieves the highest accuracy at 95.11%, followed by GIN with 92.07%, showcasing their superior capability in handling graph data. Although GCN has a lower accuracy of 76.80%, it still provides valuable insights. Other notable models include Unet-Graph at 95.81%, DGCNN at 92.18%, SGC at 90.79%, and GIN-JK at 90.00%. The comparison underscores the effectiveness of GraphSAGE and GIN, which outperform many existing methods, validating their selection for our study. However, extensive comparison on more data is needed to further validate these findings and ensure the robustness of these models across different datasets and scenarios.

9.6 Discussion

This chapter explores the use of Graph Neural Networks (GNNs) for detecting and classifying malicious files by analyzing their control flow graphs. The specialized architecture and training strategy employed aims to improve both the accuracy and robustness of the detection system. GNNs demonstrated their capacity to capture complex structural dependencies within malware graphs, effectively identifying relationships among various components such as functions and control structures. One of the significant advantages observed during our experiments was the adaptability of GNNs to handle a wide range of malware families and variants. This flexibility is crucial in the ever-evolving malware landscape, where new and diverse threats continually emerge. The experiments confirmed that GNNs could generalize well across different types of malware, providing reliable detection capabilities.