Graph Neural Networks for Malware Classification

Vrinda Malhotra

San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects

Part of the Artificial Intelligence and Robotics Commons, and the Information Security Commons

Recommended Citation
Malhotra, Vrinda, "Graph Neural Networks for Malware Classification" (2022). Master's Projects. 1194.
DOI: https://doi.org/10.31979/etd.znr4-vz7n
https://scholarworks.sjsu.edu/etd_projects/1194

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.
Graph Neural Networks for Malware Classification

A Project
Presented to
The Faculty of the Department of Computer Science
San José State University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

by
Vrinda Malhotra
December 2022
The Designated Project Committee Approves the Project Titled

Graph Neural Networks for Malware Classification

by

Vrinda Malhotra

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

December 2022

Dr. Mark Stamp       Department of Computer Science
Dr. Katerina Potika  Department of Computer Science
Dr. William Andreopoulos  Department of Computer Science
Malware is a growing threat to the digital world. The first step to managing this threat is malware detection and classification. While traditional techniques rely on static or dynamic analysis of malware, the generation of these features requires expert knowledge. Function call graphs (FCGs) consist of program functions as their nodes and their interprocedural calls as their edges, providing a wealth of knowledge that can be utilized to classify malware without feature extraction that requires experts. This project treats malware classification as a graph classification problem, setting node features using the Local Degree Profile (LDP) model and using different graph neural networks (GNN) to generate embeddings for each graph which can then be classified. We particularly highlight Jumping Knowledge-based Graph Isomorphism Network (JK-GIN), Jumping Knowledge-based GraphSAGE (JK-GrahSAGE), UnetGraph, and Deep Graph Convolutional Network (DGCNN) since they performed the best and had similar runtimes. These models performed better than existing state-of-the-art approaches in terms of F1 scores. They also address the over-smoothing problem that is rampant with other GNN models.

Keywords: Malware Classification, Graph Neural Networks, Jumping Knowledge, Graph Kernels, Machine Learning, Graph Embedding
ACKNOWLEDGMENTS

I am obliged to have worked under Dr. Mark Stamp as my advisor. Having taken "Machine Learning" under Dr. Mark Stamp, I was interested in working with him for my writing project. His constant guidance and support helped me a lot. I would also like to thank Dr. Katerina Potika and Dr. William Andreopoulos for not only providing valuable guidance feedback.
# TABLE OF CONTENTS

## CHAPTER

1. Introduction ......................................................... 1

2. Related Works ...................................................... 3
   2.1 Traditional Learning-Based Classification .................. 3
   2.2 Graph Learning-Based Classification ....................... 4
   2.3 Dataset ....................................................... 6

3. Background ........................................................ 8
   3.1 Node Feature Generation .................................... 9
   3.2 Graph Learning Methods .................................... 10
      3.2.1 Multi-Layer Perceptron ............................... 10
      3.2.2 Graph Kernel Methods ................................. 10
      3.2.3 Random Walk-Based Representations ................ 11
      3.2.4 Spectral Distance-based Representations ........... 11
   3.3 Graph Neural Networks ...................................... 11
      3.3.1 Graph Convolutional Networks ....................... 11
      3.3.2 GraphSAGE ............................................... 13
      3.3.3 Graph Isomorphism Network .......................... 13
      3.3.4 Simple Graph Convolution .............................. 14
   3.4 Jumping Knowledge Networks ................................ 14
   3.5 U-NetGNN ....................................................... 15
   3.6 Deep Graph Convolutional Neural Network .................. 16
4 Experiments and Results .................................................. 18

4.1 Malnet-Tiny Dataset ...................................................... 18

4.2 Infrastructure Setup ..................................................... 18

4.3 Graph Learning Methods .............................................. 19

4.3.1 Multi Layer Perceptron ............................................ 19

4.3.2 Graph Kernel Method .............................................. 19

4.3.3 Random Walk-Based Representations ......................... 20

4.3.4 Spectral Distance-based Representations ..................... 21

4.4 Graph Neural Networks .............................................. 22

4.4.1 Graph Convolutional Network .................................. 23

4.4.2 GraphSAGE .......................................................... 23

4.4.3 Graph Isomorphism Network .................................... 23

4.4.4 Simple Graph Convolution ...................................... 24

4.5 Jumping Knowledge Networks ...................................... 25

4.5.1 Jumping Knowledge Graph Convolutional Network ....... 25

4.5.2 Jumping Knowledge GraphSAGE ............................... 25

4.5.3 Jumping Knowledge Graph Isomorphism Network .......... 26

4.6 UnetGraph ............................................................ 27

4.7 Deep Graph Convolutional Neural Network .................... 28

5 Discussion .............................................................. 29

5.1 F1 Score Comparison ................................................ 29

5.2 Classwise Comparison ............................................... 30

5.3 Runtime Comparison ............................................... 32
5.4 Mapping Graph Embeddings for Neural Network-based Models . . . . 32

6 Conclusion and Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
  6.1 Future Works . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37

LIST OF REFERENCES . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

APPENDIX

A Figures Used for Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
B Tables Used for Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
CHAPTER 1

Introduction

Android malware is a common problem on mobile devices and poses a serious challenge due to the sheer volume and diversity of such malware. According to a report published by AV-TEST [1] in 2021, 9.09 million new malware samples were intercepted on mobile terminals, an average of about 24,000 new malware samples every day, out of which 3.5 million were Android-based malware. Since Android is based on an open-source model, it is flexible for mobile developers to create custom apps. However, this same flexibility is exploited by bad actors to create malicious apps that can access sensitive user information and sell it for a profit. Users can also install third-party applications that might be untrustworthy, giving bad actors another way to exploit them using malware. Therefore, solutions that counteract malware are crucial for online public safety.

Earlier studies [2] have illustrated that new malware apps tend to be a variant of pre-existing malware families and showcase similar behavioral traits. For example, GinMaster sends confidential information to a remote server, while DroidKungFu allows a hacker to control the device from a remote location and access confidential information. These are two different families of malware, that can be classified using their different behavior patterns. Researchers classify malware into families based on various characteristics [3].

Currently, signature-based malware detection methods are popular for creating endpoint protection systems because they are comparatively quick and do not use additional infrastructure to collect and analyze the data. However, they require expert knowledge to reverse engineer malware instances and produce the features that will be used for detection. Reverse engineering does not scale as fast as malware production. Static malware analysis is also vulnerable to code obfuscation techniques employed by polymorphic and metamorphic malware [4]. On the other hand, dynamic malware analysis or behavioral analysis is based on behavioral data such as API or system calls, which is harder to
obfuscate [5] but in order to collect this data, we need to run the program in a sandbox environment [6] which can require significant resources. These traditional techniques fail to identify zero-day malware, whereas machine learning-based methods have the potential to identify and classify zero-day malware based on static and dynamic features, such as permissions and API calls [7].

Studies such as DREBIN [2] perform broad static analysis to generate features like user permissions, suspicious API calls, and network addresses and then use machine learning algorithms to classify malware. However, these machine-learning algorithms generally do not consider interactions between different function calls. These can be taken into account by using a graph-like data structure. Graph-based methods do not assume that the features of a particular instance are independent of each other, and the models themselves can learn relationships between features. This provides us with an additional layer of information that is inherent in the input data and can be utilized with the correct model. Moreover, manual analysis of code is not required to generate feature sets for our graph-based models, giving them a significant advantage over traditional techniques, such as signature analysis. The goal of this research is to explore malware classification techniques using graph-based learning that relies solely on graphs generated from source code, without using typical features such as permissions and API calls.

The remainder of this paper is organized as follows. In Chapter 2, we discuss relevant work related to our research. In Chapter 3, we explore several graph learning techniques based on different principles: a traditional neural network, MLP as the first step in graph learning, then graph learning based on graph kernels, random walk, and spectral distance, followed by several graph neural networks. Chapter 4 includes implementation details related to the experiments that we perform. In Chapter 5, we analyze and discuss our results obtained from the previous chapter. In Chapter 6, we provide a summary of our main results and discuss potential avenues for future work.
CHAPTER 2
Related Works

In the previous chapter, we discussed some drawbacks of static and dynamic analysis methods, in this chapter, we highlight a few learning-based methods that were used for malware classification previously. We divide this survey chapter into two sections: traditional learning-based classification and graph learning-based classification.

2.1 Traditional Learning-Based Classification

In a paper from 2013 [8], we see that a Naive Bayes classifier was trained to identify malware with 58 previously defined attributes that were based on the source code of the malware, implying that these features were handcrafted. In 2014, DREBIN introduced [2] features that were extracted from .apk files, like permissions and Application Programming Interface (API) calls using broad static analysis and then classified using Support Vector Machine (SVM). This automated the process of feature selection to a certain extent. A later method, Significant Permission Identification for Android Malware Detection (SIGPID) [9], mined the permissions data of each malware app using three levels of pruning techniques to identify 22 significant permissions as features and used an SVM to distinguish between malicious and non-malicious apps. After the popularization of deep learning models, we see papers like [10] where permission sequences were extracted as word embeddings and fed into a Long Short-Term Memory Network (LSTM) model and used for malware detection. Similarly, [7] used API features that were selected based on their occurrence frequency and then mapped into a graph matrix as input for the Convolutional Neural Network (CNN) classifier. The results showed that the 20 best API calls performed better malware detection but this is dependent on the dataset used. All these methods use a combination of permissions and API calls to extract feature sets through different techniques.

In parallel, opcode-sequence \( n \)-gram features were also utilized in training a simple
SVM to detect malware [11]. Later, [12] used raw opcode sequences and encoded them as one-hot vectors, and then classified them using a CNN. Although, these approaches are prone to errors since opcode sequences can be directly altered with bytecode-level obfuscation. We also see some inspired techniques for malware classification, like [13], where patterns called inter-component communication (ICC) are extracted from the code, utilizing communication that is used to attack the system to detect malware. Another popular technique for malware classification is to encode malware app information into grayscale images and then use CNNs to perform malware detection and classification [14, 15].

2.2 Graph Learning-Based Classification

While traditional learning for malware classification relies on the type of features and the classifier, graph-based learning relies on the type of graphs and how the node features are embedded. For example, [16, 17, 18] use API call graphs. In [16] apps are represented in relation to APIs, and API relationships are mapped as a structured Heterogeneous Information Network (HIN). The semantic relatedness of apps and APIs is then described using a meta-path-based approach, aggregating similarities using multi-kernel learning. API call sequence graphs along with permission features learn multiple embedding representations for malware detection classification in [17]. They use a recurrent neural network to decode the deep semantic information, to independently extract features, and a version of Graph Convolutional Networks (GCNs) for modeling high-level graphical semantics. In [18], the skip-gram model is used to extract features based on API sequences for each for graph node. These features are based on both App-API relationships and API-API relationships to form a heterogeneous graph.

Another type of graph used is an opcode-level function call graphs [19, 20, 21]. While [19] uses opcode-sequences as text features along with traditional ML methods
like Random forest, SVM, etc. [21] does the using an LSTM-based neural network. On the other hand, [20] utilizes a new novel graph structure known as a co-opcode graph corresponding to each of the metamorphic families that extracts engine-specific patterns based on opcode from the graphs and methods similar to the Hidden Markov model (HMM) to classify malware. We see that opcode-level graphs are treated more or less like text features rather than graph data and hence are not zero-day ready. Using dynamically generated network flow graphs, [22] creates a new model called Network Flow Graph Neural Network (NF-GNN) with two variations, which extracts the flow as graphs and then classifies the graphs using a new edge feature-based GNN model. The model performs very well in all settings. Heterogeneous graphs use more than one type of relationship to make connections between nodes.

However, the methods mentioned above are transductive, which means that they cannot classify or recognize unknown applications or even zero-day malware, since every API call or permission cannot exist in the model embeddings. We aim to illustrate graph-based models that are inductive, so as to extend them to predicting zero-day malware and unknown applications like using control flow graphs (CFG). In a CFG, nodes represent program statements, including called subroutines and conditionals, while edges represent the flow of the program. They are intraprocedural mappings that are better suited to classifying zero-day malware [23] since they only rely on the source code of an apk. In [23], a deep graph convolutional neural network (DGCNN) embeds structural information that is inherent to CFGs for effective malware classification.

The interprocedural counterpart to CFGs, function call graphs or FCGs are graphs where nodes represent subroutines, while edges represent the relationship caller-called between two subroutines. In [21], opcode level function call graphs are obtained from static analysis of malware while [24] uses a Natural Language Processing (NLP)-inspired using graph embedding to convert the graph-like structure of an app into a vector. On
a similar note, [25] introduced MalNet, a large-scale Android malware Function Call Graph (FCG) dataset extracted from \texttt{apk} files and used state-of-the-art graph learning techniques like GraphSAGE [26] and GIN [27] to classify FCGs as different malware types and families. Among these methods, GIN and Feather [28] had the best classification performance. Using the same dataset, [29] employs jumping knowledge GNNs with node features set to page rank, degree nodes, etc., and concludes that GraphSAGE-JK performed the best.

2.3 Dataset

While deciding on the type of graph to use, we wanted to focus on one that did not require handcrafted features, could work for zero-day malware, and could be generated without malware forensics. Network flow graphs and control flow graphs are difficult to generate and require resources to store them that were not available to us for this project, while heterogeneous graphs are difficult to interpret and encode. A function call graph is easy to interpret since it has all the execution paths of a program that are called during its runtime. For example, if an app is making a sequence of function calls to gather sensitive information and send it to a server, all the possible calls are mapped in the FCG. Therefore, FCG dataset was the best way forward.

We explored several datasets for our project. A basic comparison between commonly used datasets for malware detection is provided in Table 1. We also provide a summary of the papers presented in the previous section and the datasets that they used along with their accuracies in B.10.

All these datasets consisted of malware \texttt{apk} files or hexadecimal representations of the file’s binary content along with manifests that contain metadata information like function calls, strings, etc. For initial experiments, we used the Android malware Genome dataset. We used the Androguard tool to generate Function Call Graphs from Android
Table 1: Common Datasets for Malware Classification

<table>
<thead>
<tr>
<th>Name</th>
<th>Families</th>
<th>Sample Size</th>
<th>Papers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Android Malware Dataset (AMD)</td>
<td>42</td>
<td>5,000</td>
<td>[17, 18, 24]</td>
</tr>
<tr>
<td>Android Malware Genome Project</td>
<td>72</td>
<td>1,260</td>
<td>[24]</td>
</tr>
<tr>
<td>Drebin Dataset</td>
<td>179</td>
<td>5,560</td>
<td>[17, 18, 24]</td>
</tr>
<tr>
<td>Microsoft Malware Challenge dataset</td>
<td>9</td>
<td>10,000</td>
<td>[20, 22, 23]</td>
</tr>
<tr>
<td>AndroZoo/Malnet-tiny Dataset</td>
<td>5</td>
<td>5,000</td>
<td>[25, 30]</td>
</tr>
</tbody>
</table>

apk files. An example of a FCG, constructed in Androguard and then converted into an image of a multi-directional graph is given in Figure 1(a). We can also filter the function call graph to show a subsection of methods and calls as highlighted in Figure 1(b).

![Function Call Graph](image1)

(a)

![Filtered View of FCG](image2)

(b)

Figure 1: (a) Visualizing a Function Call Graph and (b) Filtered View of FCG using Gephi

While the initial experiments were conducted using Android Malware Genome Project dataset, we quickly found out that it might not yield good results for graph-based models due to the small size of the dataset. Therefore, we used a subset of Malnet dataset called the Malnet-tiny dataset, which has become a benchmark for malware classification and detection since it is a balanced dataset consisting of 1000 malware Function Call Graphs for each of its five classes [25].
CHAPTER 3
Background

A graph is a data structure consisting of two components: nodes (also known as vertices) and edges. A graph $G$ is defined as $G = (V, E)$, where $V$ is the set of nodes, and $E$ is the set of edges. These edges can be directed or undirected. An example of a graph is given in Figure 2.

![Figure 2: Example of a Graph](image)

Graph data is more complex than traditional text and image data. Conventional Machine Learning and Deep Learning tools are technically specialized for simple data types. For example, images with the same structure and size can be thought of as fixed-size grid graphs while text and speech are sequential and can be line graphs. However, graphs can be varied and more complex, with any number of unordered nodes, where nodes can have different amounts of neighbors and without a fixed form.

Recently, analyzing graphs with machine learning has become more popular. Traditional machine learning algorithms such as MLPs and CNNs treat the features of a particular instance as if they are independent of each other. While Recurrent Neural Networks (RNN) can deal with sequential data, Graph Neural Networks (GNN) allow us to model arbitrary interactions that are far beyond the scope of RNNs.
Graph Neural Network [31] is any neural network that directly operates on a graph structure. They are based on CNNs and graph embedding and collect aggregate information from graph structure. Input and output made up of elements and their dependencies can therefore be modeled. The process of creating a GNN model is described in Figure 3.

Specifically, in malware, we see that it is easier to understand and classify behaviors if we can map relationships between the methods that are not implicit in traditional ML methods of classification.

3.1 Node Feature Generation

We extracted features like average degree nodes, node count, edge count, and page rank from the graphs for classification. We also used Local Degree Profile (LDP) [32] for generating node features. LDP is a representation technique that uses five-degree statistics to summarize each node and its one-hop neighborhood by computing min, max, average, and standard deviation. The resulting node features are then combined into feature vectors in a histogram by concatenating them. After initial experiments with Multi-layer Perceptron (MLP), the results using LDP [32] were far more consistent than
ones from our extracted features as given in Figure A.31 and therefore, LDP features became the standard for further analysis using neural networks including MLP, GNNs, JK-GNNs, UnetGraph, and DGCNN.

3.2 Graph Learning Methods

We explore several methods to learn graphs. Some methods like MLP learn representations based on the LDP-based node features that were discussed previously. On the other hand, graph kernels reduce the dimensionality of the graph data itself, while random walk and spectral distance-based methods learn a new representation of the graph data.

3.2.1 Multi-Layer Perceptron

Traditional ML models like MLP were used to draw up a comparison between the effectiveness of graph-based models. We also found that the traditional feature selection in these models takes time and a certain level of domain knowledge that is not required while using other methods. Therefore, we used LDP [32] features mentioned earlier.

3.2.2 Graph Kernel Methods

Graph kernels are a supervised classification method and use the kernel trick to reduce dimensionality in conjunction with Machine Learning models. They help process graph features, unlike traditional ML models that use domain-specific, hand-crafted features. In each step, the labels of the node are renamed with a set of labels formed by combining the immediate neighbors. This step is repeated until the two graph’s labels differ. A kernel matrix is computed upon applying a graph kernel. This matrix is passed to a kernel-based machine learning algorithm to perform classification. We used Python’s GraKel library to test Weisfeiler-Lehman (WL) kernels [33]. Weisfeiler-Lehman Subtree Kernel, based on the test of the same name, is traditionally used for finding isomorphism between two graphs [34].
3.2.3 Random Walk-Based Representations

Feather [28] is a sophisticated representation technique that combines random walk weights along with characteristic functions of node features to identify node neighborhoods and to create Euclidean node embeddings. These characteristic functions have probability weights that are defined by their tie strength. It employs one particular instance of these functions, namely, the $r$-scale random walk weighted characteristic function to generate the embeddings. Later, Random forest is used to classify said embeddings.

3.2.4 Spectral Distance-based Representations

We use Slaq-VNGE and Slaq-LSD, both of which approximate the spectral distances between graphs based on different functions. While Slaq-VNGE [35] uses a measure of information divergence and distance between graphs called Von Neumann Graph Entropy (VNGE), Slaq-LSD [36] uses a kernel to approximate the spectral distance between graphs, also called NetLSD.

3.3 Graph Neural Networks

Image classification has been effectively addressed using convolutional neural networks (CNNs) that are restricted to Euclidean data structures. Thus the basic idea of GNNs is to extend CNNs, to produce node embeddings that reduce the dimension of the embedding space for the graphs. The mean of these node embeddings is then taken to form the complete graph embedding, which encodes a particular graph into low-dimensional space and then uses this embedding for graph classification [37]. The architecture of a GNN model is highlighted in Figure 4.

3.3.1 Graph Convolutional Networks

Graph Convolutional Network or GCN generates node embeddings by aggregating the node features in a given node’s neighborhood. We consider graph $G = (N, E, A)$, where $N$ is the set of nodes and $E$ is the set of edges. Here, $|N|$ is the number of nodes in
Figure 4: Architecture of a GNN Model for Graph Classification

the graph, and $|E|$ is the number of edges, while $A$ is the $N \times N$ sparse adjacency matrix. A $k$-dimensional feature vector corresponds to each node, and the node feature matrix is $X \in \mathbb{R}^{N \times K}$. If the GCN is $L$-layered GCN then the number of graph convolutions represented as layers is $L$, where each layer generates embeddings for every node by combining the embeddings of the given node’s neighbors as specified in the previous layer [38].

The equation for the transformation on the $l + 1$ layer is given by

$$Z^{l+1} = \sigma(X^{(l)}W_0^{(l)} + \tilde{A}X^{(l)}W_1^{(l)})$$

where $X^{(l)} \in \mathbb{R}^{N \times K_l}$ is the embedding for $k$-dimensional feature vectors of all $N$ nodes at the $l$-th layer ($K_l$ being the dimension of each at $l$-th layer) and $X^{(l)} = X \cdot W^{(l)}$ represents the learned weight matrix that is used for subsequent tasks. The $\sigma$ is an activation function that is generally the ReLU function. For $L$ layered GCN, the output is represented as $Z(L)$, a matrix consisting of each of the node embeddings after they undergo $L$ layers of transformations.
3.3.2 GraphSAGE

Introduced in Hamilton et al., Graph Sample and aggregate (GraphSAGE) algorithm employs random sampling on a node neighbor subset of pre-determined size. Regardless of the graph topology or batch size, this enables the algorithm’s space and time complexity to be limited. Local information about a node’s neighborhood is gathered and utilized to compute the node embedding, much like the convolution operation in CNNs.

In every iteration, first, the node’s immediate neighborhood is sampled, and then the data from these sampled nodes are combined into one vector. Let \( u \) be a node in the neighborhood of node \( v \), that is, \( u \in N(v) \). Then, at the \( k \)-th layer, the aggregated information \( h_{N(v)}^k \) at a node \( v \), as computed for the sampled neighborhood \( N(v) \), is expressed as

\[
h_{N(v)}^k = AGG_k \cdot (h_{u}^{k-1}, \forall u \in N(v))
\]

where \( h_{u}^{k-1} \) represents the embedding of each node \( u \) that lies in the neighborhood of node \( v \), that is, \( u \) is at a one-hop distance from \( v \) in the previous layer. The embedding of node \( u \) in the neighborhood of \( v \) is combined into the embedding of node \( v \) at layer \( k \). The aggregators or \( AGG \) function is either pooling, mean or LSTM. We use Pytorch’s mean as our aggregator. This combined embedding or \( h_{N(v)}^k \) of the sampled neighborhood is then added to the previous layer’s node embeddings.

We then apply the model’s learning parameters, to obtain \( W_k \) or the trainable weight matrix, and pass it through the activation function \( \sigma \). The node embedding at layer \( k \) is then combined. The resulting node embedding of node \( v \) is expressed as \( z_v \), which is essentially the node embedding at the final layer \( K \).

3.3.3 Graph Isomorphism Network

For graph classification tasks, Graph Isomorphism Network (GIN) [26] is considered better than the aforementioned GNNs. It differs from them due to its message aggregation
function, given by

\[ h_v^{(k)} = \text{MLP}^{(k)} \cdot \left[ (1 + \epsilon^{(k)}) \cdot h_u^{k-1} \cdot \sum_{u \in N(v)} h_u^{k-1} \right] \]

where MLP\(^{(k)}\) stands for a Multi-layer Perceptron on \(k\)-th layer and \(\epsilon^{(k)}\) is a scalar parameter. Message aggregation methods are similar to the hash function of the Weisfeiler-Lehman algorithm mentioned earlier and come up short when compared to it. GIN, with an injective message aggregation function, was introduced to overcome this barrier.

### 3.3.4 Simple Graph Convolution

GCNs have a significant amount of complexity due to their connections to CNNs, which can be problematic for problems that require lower complexity. By continually eliminating the non-linearities between given convolution layers and condensing the output into a linear transformation, Simple Graph Convolution or SGC lowers this excess complexity that is present in a GCN. SGC streamlines the process into a feature propagation phase followed by conventional logistic regression, as opposed to a GCN that modifies the feature vectors continuously in each of the \(K\) layers and then uses a Adam optimizer as a classifier to generate the final embeddings. We compare GCN architecture to SGC in Figure 5.

### 3.4 Jumping Knowledge Networks

As in CNNs, GNNs with more depth exhibit inferior performance because, while the message-passing mechanism aids in the extraction of information from the graph structure, it may have certain drawbacks when combined with GNN depth. Therefore, our effort to increase the number of layers in order to provide nodes with more information results in a model that treats all nodes equally. This is called the over-smoothing problem [40]. To mitigate this, we apply the Jumping Knowledge [41] technique by using the concatenation layer. A Jumping Knowledge Network’s main principle calls for choosing from among all intermediate node representations before moving on to the final layer, where these
intermediate representations are combined to create the node embedding used in the classifier. For the purpose of computing the final node embeddings, layer aggregation concatenation was used to merge all intermediate node representations which then undergo global pooling, performing element-wise pooling over the final node embeddings. We apply Jumping Knowledge to GCN, GraphSAGE, and GIN models described previously. The architecture of these models is given in Figure 6. Jumping Knowledge networks were recently used in [29] but they use a different feature set that does not include the LDP-based features that we used or provide a comparison with earlier GNN models.

3.5 U-NetGNN

In [42], Unets are used to correlate graph data to images by considering images as a special case of graphs, where nodes are located on typical 2D lattices. We can apply convolution and pooling techniques on images thanks to this structure. Therefore, graph embedding and graph classification tasks can correspond to pixel-wise prediction tasks like image segmentation [43]. Recently, pixel-wise prediction tasks have achieved major advances. Originally introduced in [44], U-Nets [45], present a way to apply these advances to graph learning problems. Unets are based on encoder-decoder architecture, popular for 3D image segmentation tasks. To extend the operations to pooling and up-sampling as

Figure 5: Layout of GCN vs. SGC [39]
in image segmentation, [42] introduces graph pooling (gPool) and unpooling (gUnpool) operations. We use GraphUNet implementation of this architecture from the Torch Geometric library, and its architecture is given in Figure 7.

Figure 6: Graph Embedding Generation with Jumping Knowledge GNN Model

Figure 7: Architecture of a U-Net GNN Model for Graph Classification [42]

3.6 Deep Graph Convolutional Neural Network

Deep Graph Convolutional Neural Network or DGCNN [46] has three main stages. First is the graph convolution layers which extract the local substructure information from the vertices and provide a fixed vertex ordering. Second is the SortPooling layer
that sorts the given vertex features according to the previously provided vertex order and makes the input sizes uniform. Lastly, there are conventional dense layers that scan the ordered graph embeddings and produce predictions.

As shown in Figure 8, a number of graph convolution layers are first applied to an input graph with any structure, allowing node information to spread among neighbors. Then, to develop a predictive model, given the vertex features are passed to a conventional CNN structure after sorting and pooling them with a SortPooling layer.

Figure 8: Architecture of a DGCNN Model for Graph Classification [46]

Comparing DGCNN to current graph neural networks, there are several benefits. First, it allows end-to-end gradient-based training since it takes graph data as a direct input while others transform graphs into tensors first. By sorting vertex features rather than adding them up, it enables learning from global graph topology, which is enabled by an unique SortPooling layer. DGCNN is also the only sorting-based method that we explored and can open up new areas of discussion. DGCNN was previously used in [46] for malware family classification using API call graphs.
CHAPTER 4

Experiments and Results

In this chapter, we first discuss the dataset that we use in our experiments. Then, to provide a baseline comparison, we consider a standard MLP model, followed by extensive experiments with a wide range of GNN models.

4.1 Malnet-Tiny Dataset

We use the Malnet-Tiny dataset which consists of 5000 Function Call Graphs of a subset of apk files extracted from the AndroZoo dataset. This is a balanced dataset, with each malware type having 1000 samples. We also provide an example graph for each family in Figure A.32. We highlight some basic features of the graphs in Table 2 like the minimum, maximum, median, and standard deviations of both the number of vertices and the number of and edges and also their average degree. From the data, we can clearly see that Downloader family should be relatively easier to classify since the basic graph features differ a lot from others, although they might overlap in features with Trojans. We also see that the Trojan family is fairly distinct but we can’t be entirely certain since their standard deviations in the number of nodes and edges are similar to the others.

Table 2: Overview of Graph Features

<table>
<thead>
<tr>
<th>Malware Type</th>
<th>Number of Vertices</th>
<th>Number of Edges</th>
<th>Average Degree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min</td>
<td>Max</td>
<td>Median</td>
</tr>
<tr>
<td>AdDisplay</td>
<td>122</td>
<td>4923</td>
<td>1248</td>
</tr>
<tr>
<td>Adware</td>
<td>211</td>
<td>4983</td>
<td>2317</td>
</tr>
<tr>
<td>Benign</td>
<td>5</td>
<td>4994</td>
<td>1790.5</td>
</tr>
<tr>
<td>Downloader</td>
<td>40</td>
<td>117</td>
<td>51</td>
</tr>
<tr>
<td>Trojan</td>
<td>9</td>
<td>4993</td>
<td>144</td>
</tr>
</tbody>
</table>

4.2 Infrastructure Setup

For all graph learning-based analyses, we use Python’s pyTorch-geometric library. We also use Python’s sklearn library for methods that require a traditional classifier like
a random forest classifier. We parallelize our code using Python’s *joblib* package. Our GPU uses CUDA version 11.2. Lastly, we use Google Colab for our experiments.

### 4.3 Graph Learning Methods

Here we explore five methods: Multi-Layer Perceptron (MLP), WL, Feather, LSD-Slaq and VNGE-Slaq. Each model requires a different set of parameters. Their best parameters are highlighted in boldface in Table 3.

<table>
<thead>
<tr>
<th>Table 3: Accuracy and best parameters for each GNN model</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Model</strong></td>
</tr>
<tr>
<td>MLP</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>WL</td>
</tr>
<tr>
<td>Feather</td>
</tr>
<tr>
<td>LSD-Slaq</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>LSD-VNGE</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

#### 4.3.1 Multi Layer Perceptron

We use MLP as a benchmark for traditional ML models. We use LDP to generate node features for classification. We train a 5-layer MLP model for 50 epochs dropout set to 0.5. The loss curve, accuracy curve, and classwise F1-score are plotted in Figure 9. The model was tuned using the parameters mentioned in Table 3, with the best validation accuracy being 80.53%. The complete classification report is given in Figure A.33.

#### 4.3.2 Graph Kernel Method

We use node degree and page ranks and concatenated them into feature vectors for each node to generate features for kernel-based methods. We used Weisfeiler-Lehman Subtree Kernel to compute a kernel matrix and then applied Random Forest for classification. The model was tuned using the parameters mentioned in Table 3, and the best
Figure 9: (a) Loss Curve, (b) Accuracy Curve and (c) Classification Report for MLP

accuracy was 70.53%. The accuracy and runtime metrics for each parameter are given in Table B.7. The classwise F1-Scores for the best parameters are given in Figure 10 while the complete classification report is given in Figure A.34.

Figure 10: Classwise F1-Scores for WL Kernel-based model

4.3.3 Random Walk-Based Representations

Using Feather [28], we perform a grid search over the key order parameter, which controls how much information is seen from higher-order neighborhoods. The model was tuned using the parameters mentioned in Table 3, and the best accuracy was 84.88%. The accuracy and runtime metrics for each parameter are given in Table B.8. The classwise F1-Scores for the best parameters are given in Figure 11 while the complete classification report is given in Figure A.35.
4.3.4 Spectral Distance-based Representations

For Slaq-LSD and Slaq-VNGE, we perform a grid search over two key parameters: the number of random vectors and the number of Lanczos steps. The models were tuned using the parameters mentioned in Table 3. The accuracy and runtime metrics for each parameter are given in Table B.9. The classwise F1-scores for the best parameters of Slaq-LSD and Slaq-VNGE are given in Figure 12(a) and Figure 12(b) respectively. The complete classification reports for both models is given in Figure A.36.
4.4 Graph Neural Networks

In this section, we build the GNN models mentioned earlier. The input graph is a batched graph produced by GraphDataLoader. The readout functions provided by Torch Geometric can handle batched graphs so that they return one representation for each minibatch element.

Each model is trained for 200 epochs. Each of the convolutional layers computes new node representations using a convolutional operator from Pytorch’s Torch Geometric library and then optimized using Adam optimizer. Each model was tuned on the following hyperparameters:

- Number of Layers $\in [5, 6]$
- Hidden Dimensions $\in [64, 128]$
- Learning Rate $\in [0.001, 0.0001]$

Initial experiments were also conducted with a 5, 6, and 7-layered GCN but after seeing virtually no difference between 6 and 7-layered models and a considerable increase in run-time, the 7-layered model was scraped. We highlight the best parameters in boldface for each GNN model in Table 4.

<table>
<thead>
<tr>
<th>Model</th>
<th>Hyperparameters</th>
<th>Tested Values</th>
<th>Validation Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCN</td>
<td>Number of Layers</td>
<td>[5, 6]</td>
<td>95.82%</td>
</tr>
<tr>
<td></td>
<td>Hidden Dimensions</td>
<td>[64, 128]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Learning Rate</td>
<td>[0.001, 0.0001]</td>
<td></td>
</tr>
<tr>
<td>GraphSAGE</td>
<td>Number of Layers</td>
<td>[5, 6]</td>
<td>79.13%</td>
</tr>
<tr>
<td></td>
<td>Hidden Dimensions</td>
<td>[64, 128]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Learning Rate</td>
<td>[0.001, 0.0001]</td>
<td></td>
</tr>
<tr>
<td>GIN</td>
<td>Number of Layers</td>
<td>[5, 6]</td>
<td>94.07%</td>
</tr>
<tr>
<td></td>
<td>Hidden Dimensions</td>
<td>[64, 128]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Learning Rate</td>
<td>[0.001, 0.0001]</td>
<td></td>
</tr>
<tr>
<td>SGC</td>
<td>Number of Layers</td>
<td>[5, 6]</td>
<td>90.79%</td>
</tr>
<tr>
<td></td>
<td>Hidden Dimensions</td>
<td>[64, 128]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Learning Rate</td>
<td>[0.001, 0.0001]</td>
<td></td>
</tr>
</tbody>
</table>
4.4.1 Graph Convolutional Network

GCN uses the convolutional operator GCNConv and a global pooling layer to generate embeddings. The model was tuned using the parameters mentioned in Table 4, and the best accuracy was 95.82%. The accuracy curves for each hyperparameter are plotted in Figure A.37(a). The loss curve, accuracy curve, and classwise F1-score are plotted in Figure 13 while the complete classification report is given in Figure A.37(b). While the accuracy achieved is fairly high, we can see from the curves provided in Figure 13 that it suffers from the over-smoothing problem.

![Figure 13: (a) Accuracy Curve, (b) Loss Curve, and (c) Classwise F1 Scores for GCN](image)

4.4.2 GraphSAGE

GraphSAGE uses the convolutional operator SAGEConv, a batch normalization layer, and a global pooling layer to generate embeddings. The model was tuned using the parameters mentioned in Table 4, and the best accuracy was 79.13%. The accuracy curves for each hyperparameter are plotted in Figure A.38(a). The loss curve, accuracy curve, and classwise F1-score are plotted in Figure 14 while the complete classification report is given in Figure A.38(b).

4.4.3 Graph Isomorphism Network

GIN uses the convolutional operator GINConv and a global pooling layer to generate embeddings. The model was tuned using the parameters mentioned in Table 4, and the
Figure 14: (a) Accuracy Curve, (b) Loss Curve, and (c) Classwise F1 Scores for GraphSAGE

The best accuracy was 94.07%. The accuracy curves for each hyperparameter are plotted in Figure A.39(a). The loss curve, accuracy curve, and classwise F1-score are plotted in Figure 15 while the complete classification report is given in Figure A.39(b).

Figure 15: (a) Accuracy Curve, (b) Loss Curve, and (c) Classwise F1 Scores for GIN

4.4.4 Simple Graph Convolution

SGC uses the convolutional operator SGConv and a global pooling layer to generate embeddings. The model was tuned using the parameters mentioned in Table 4, and the best accuracy was 90.79% The accuracy curves for each hyperparameter are plotted in Figure A.40(a). The loss curve, accuracy curve, and classwise F1-score are plotted in Figure 16 while the complete classification report is given in Figure A.40(b).
4.5 Jumping Knowledge Networks

We use an extension of the torch.nn.Sequential container in order to define a sequential GNN model. Since GNN operators take in multiple input arguments, torch_geometric.nn.Sequential expects both global input arguments, and function header definitions of individual operators. If omitted, an intermediate module operates on the output of its preceding module. This allows us to create more sophisticated models, such as Jumping Knowledge models. We highlight the best parameters for each GNN model in boldface in Table 5.

4.5.1 Jumping Knowledge Graph Convolutional Network

Similar to the GCN model, JK-GCN also uses GCNConv layers and implements them on a Jumping Knowledge model. The model was tuned using the parameters mentioned in Table 5, and the best accuracy was 89.41%. The accuracy curves for each hyperparameter are plotted in Figure A.41(a). The loss curve, accuracy curve, and classwise F1-score are plotted in Figure 17 while the complete classification report is given in Figure A.41(b).

4.5.2 Jumping Knowledge GraphSAGE

Similar to the GCN model, JK-GraphSAGE also uses SAGEConv layers and implements them on a Jumping Knowledge model. The model was tuned using the parameters mentioned in Table 5, and the best accuracy was 92.91%. The accuracy curves for each
Table 5: Accuracy and best parameters for each Jumping Knowledge GNNs, UnetGraph, and DGCNN models

<table>
<thead>
<tr>
<th>Model</th>
<th>Hyperparameters</th>
<th>Tested Values</th>
<th>Validation Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>JK-GCN</td>
<td>Number of Layers</td>
<td>[5, 6]</td>
<td>92.91%</td>
</tr>
<tr>
<td></td>
<td>Hidden Dimensions</td>
<td>[64, 128]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Learning Rate</td>
<td>[0.001, 0.0001]</td>
<td></td>
</tr>
<tr>
<td>JK-GraphSAGE</td>
<td>Number of Layers</td>
<td>[5, 6]</td>
<td>97.69%</td>
</tr>
<tr>
<td></td>
<td>Hidden Dimensions</td>
<td>[64, 128]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Learning Rate</td>
<td>[0.001, 0.0001]</td>
<td></td>
</tr>
<tr>
<td>JK-GIN</td>
<td>Number of Layers</td>
<td>[5, 6]</td>
<td>92.18%</td>
</tr>
<tr>
<td></td>
<td>Hidden Dimensions</td>
<td>[64, 128]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Learning Rate</td>
<td>[0.001, 0.0001]</td>
<td></td>
</tr>
<tr>
<td>UnetGraph</td>
<td>Weight Decay</td>
<td>[0.001, 0.0001]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Hidden Dimensions</td>
<td>[64, 128]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Learning Rate</td>
<td>[0.001, 0.0001]</td>
<td></td>
</tr>
<tr>
<td>DGCNN</td>
<td>Weight Decay</td>
<td>[0.001, 0.0001]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Learning Rate</td>
<td>[0.001, 0.0001]</td>
<td></td>
</tr>
</tbody>
</table>

Figure 17: (a) Accuracy Curve, (b) Loss Curve, and (c) Classwise F1 Scores for JK-GCN

hyperparameter are plotted in Figure A.42(a). The loss curve, accuracy curve, and classwise F1-score are plotted in Figure 18 while the complete classification report is given in Figure A.42(b).

### 4.5.3 Jumping Knowledge Graph Isomorphism Network

Similar to the GIN model, JK-GIN also uses GINConv layers and implements them on a Jumping Knowledge model. The model was tuned using the parameters mentioned in Table 5, and the best accuracy was 97.69% The accuracy curves for each hyperparameter
4.6 UnetGraph

We create a UnetGraph using Pytorch’s UnetGraph model, with depth 3 and pooling ratios set to 1 and 0.5. We also set dropout to 0.5. The model was tuned using the parameters mentioned in Table 5, and the best accuracy was 95.81%. This model was run for 100 epochs instead of 200 due to over-fitting at 200 epochs. The loss curve, accuracy curve, and classwise F1-score are plotted in Figure 20. The complete classification report for best parameters is given in Figure A.44.
4.7 Deep Graph Convolutional Neural Network

We create a DGCNN with a depth of four layers of GCNConv, two 1-D convolutional layers, and a max pooling layer. Two linear classifiers are used to concatenate these layers, with a dropout of 0.5. The model was tuned using the parameters mentioned in Table 5, and the best accuracy was 92.18%. With some more hyperparameter tuning, we believe that this can be improved upon since we only varied the learning rate and weight decay. The loss curve, accuracy curve, and classwise F1-score are plotted in Figure 21. The complete classification report for best parameters is given in Figure A.45.

![Figure 20](image1.png)
Figure 20: (a) Accuracy Curve, (b) Loss Curve, and (c) Classwise F1 Scores for UnetGraph

![Figure 21](image2.png)
Figure 21: (a) Accuracy Curve, (b) Loss Curve, and (c) Classwise F1 Scores for DGCNN
Chapter 5

Discussion

In this chapter, we try to analyze the results obtained from each of the models that we implemented. For each comparison, we split the models into broad categories, to make the analysis easier and more intuitive. We analyze overall F1 scores, classwise F1 scores, and runtimes. We also map the embeddings from the neural network-based models and compare them.

5.1 F1 Score Comparison

For ease of use and clarity, we split the analysis into two categories:

1. **GNN-based models** as given in Figure 22. We observe that JK-GIN has the highest F1 Score at 97.69, with UnetGraph being a close second at is 95.81%, while others like GIN (94.07%), JK-GraphSAGE (92.91%), DGCNN (92.18%) and SGC (90.79%) are also fairly competitive and can be used for further analysis. While GCN has a high F1 Score of 95.82%, we believe it is due to over-smoothing and therefore, not included in the best models. We especially highlight DGCNN, where the number of convolutional layers was not tuned due to a resource crunch.

![Figure 22: Accuracy Comparison for GNN-based models](image-url)
While these models expected high performance, they vastly outperformed other graph-based models that use handcrafted features like page rank and degree node that were current state-of-art [29].

2. **Non-GNN-based models** as given in Figure 23. Among the non-GNN-based models, Feather works the best at 84.88% with MLP being a close second at 80.54%. This is also expected as these models were highlighted to work the best along with GIN in the original Malnet paper [25].

![Figure 23: Accuracy Comparison for Non-GNN-based models](image)

### 5.2 Classwise Comparison

For the analysis, we generated heat maps of the F1 scores of each class for every model corresponding to its best parameters. We analyze them in two categories:

1. **GNN-based models** as given in Figure 24. We observe that GIN, GraphSAGE, and JKGCN perform the worst and mainly because they classify benign malware incorrectly. The other types of malware that badly classified is Addisplay and Adware respectively. This could be because of multiple reasons - the chief being that the data points for Trojan, Adware and Downloader classes were more diverse than benign or Addisplay types. We also notice that Downloader malware is the easiest to classify followed closely by Trojans.
2. **Non-GNN-based models** as given in Figure 25 include MLP, WL Kernel, Feather, Slaq-LSD, and Slaq-VNGE. Here again we see that benign class is the most difficult to classify for any model.

Overall, we see that downloader is the easiest class to classify followed by trojans
while benign is the hardest. This is also supported by our initial analysis of the node and edge features in Table 2. Even our worst model, VNGE-Slaq, identifies downloader class with a 93% accuracy which leads us to believe that we need a more diverse dataset. For most models, adisplay, adware and trojans seem to be equally hard to classify, although GNN-based models classify trojans better than the other two classes. However, our best models classify benign malware very well. We also provide confusion matrices for each model in Figures A.46 to A.52.

5.3 Runtime Comparison

In this section, we plot the total time taken to train, test, and validate a model using its corresponding best parameters. We call this the runtime. We analyze them in two categories:

1. **GNN-based models** are given in Figure 26. We observe that both GraphSAGE and JK-GraphSAGE run for longer durations than other models. The convolution operator SAGEConv is the most complex operator used in any of the GNNs, so this is expected. Surprisingly, both DGCNN and UnetGraph models take a similar time as other competitive models. GIN and GCN models require the least time for classification, as they use the least complex convolutional operators.

2. **Non-GNN-based models** are given in Figure 27. These models require significantly less time than GNN-based models. The best performing models, Feather and MLP, require the most amount of time.

5.4 Mapping Graph Embeddings for Neural Network-based Models

Lastly, we visualize the whole graph embeddings of nine of the 14 models after applying a non-linear, dense layer transformation with UMAP dimensionality reduction [47]. The classes are labeled alphabetically: Addisplay (0), Adware (1), Benign (2), Downloader (3), and Trojan (4). We categorize the analysis into the following categories:
1. **Classic GNNs** are given in Figure 28 include GCN, GraphSAGE, and GIN. There is some class separability between Downloader, and other classes, which is supported by our observation that Downloader was the easiest class to classify. Most class separability is achieved by GCN, which was our best-performing model among these three.

2. **JK-GNNs** are given in Figure 29 include JK-GCN, JK-GraphSAGE, and JK-GIN. There is a significant difference in class separation between Jumping Knowledge-based models given in Figure 29 as compared to the other models in Figure 28 and
Figure 28: UMAP Embeddings for (a) GCN, (b) GraphSAGE, and (c) GIN

Figure 30. It is clear that JK-GCN and JK-GIN learn similar representations. While Adware and Downloader classes achieve higher class separability, other classes are a mixed bag. Moreover, in terms of JK-GraphSAGE, comparing two other pairs of models, we can see that a higher class separability has been achieved, even though each class forms multiple smaller clusters, which could be why we don’t achieve a higher accuracy from this model.

Figure 29: UMAP Embeddings for (a) JK-GCN, (b) JK-GraphSAGE, and (c) JK-GIN

3. Other GNNs are given in Figure 30 include SGC, UnetGraph, and DGCNN. While SGC creates fewer clusters for each class, they are not very well separated. While both UnetGraph and DGCNN create more clusters that are smaller in size, DGCNN seems to achieve the best class separability.
Overall, UMAP embeddings support our major observations. They also show us that our models form smaller clusters, leading us to believe that they might be identifying smaller classes within each type of malware, but this needs to be supported by further research.
CHAPTER 6

Conclusion and Future Work

In this extensive study, we classified 5 different types of malware from Malnet-tiny dataset: addisplay, adware, benign, downloader, and trojan. Instead of static or dynamic analysis-based features, we decided to learn using graph-based learning methods and therefore used a dataset made of function control graphs. This dataset is publicly available and not memory intensive. We provide a comparison of the best models implemented on this dataset from previous papers in table 6.

Table 6: Accuracy comparison with relevant papers

<table>
<thead>
<tr>
<th>Paper</th>
<th>Node Features</th>
<th>Best Model</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Freitas et. al [25]</td>
<td>Node degree</td>
<td>GIN</td>
<td>90.00%</td>
</tr>
<tr>
<td>Lo et. al [29]</td>
<td>Degree, centrality, and page rank</td>
<td>GraphSAGE-JK</td>
<td>94.40%</td>
</tr>
<tr>
<td>Our Paper</td>
<td>LocalDegreeProfile</td>
<td>JK-GIN</td>
<td>97.69%</td>
</tr>
</tbody>
</table>

We explore several graph representation schemes including Feather, WL-kernel, Slaq-VNGE, and Slaq-LSD, and use them with a random forest classifier to classify malware. We see that Feather works best among these methods. We used LDP to encode node features and then classify malware using a simple MLP and then using several different graph neural networks including GCN, GrahSAGE, GIN, SGC, JK-GCN, JK-GraphSAGE, JK-GIN, UnetGraph and DGCNN. GNNs especially perform well, with JK-GIN, JK-GraphSAGE, UnetGraph, and DGCNN performing the best. They overcome the over-smoothing problem that plagues a lot of GNNs. Among the malware classes, we see that benign class was the hardest to classify while downloader was the easiest. This is also supported by their UMAP embeddings.

In conclusion, graph-based learning methods perform malware classification and have a lot of scope for research given that these methods can be automated since they do not require handcrafted features or any malware forensics. Two of our models beat the
current state-of-art model that used degree, betweenness centrality, and page rank as node features along with a JK-GraphSAGE model (94.40%), which include, JK-GIN at 97.69%, and UnetGraph at is 95.81%, while others like GIN (94.07%), JK-GraphSAGE (92.91%), DGCNN (92.18%) and SGC (90.79%) are also competitive and can be used for further analysis. We believe that these methods should be explored with a more diverse dataset so as to further prove this point.

6.1 Future Works

In future work, we could extend the dataset and use the complete Malnet dataset instead of the Malnet-tiny subset or at least extend the Malnet-tiny dataset to include other types of malware for a diverse analysis. We could also use Feather embeddings as node features and classify using the models that we have created as a way to compare the effectiveness of Feathers versus LDP. It would also be worthwhile to explore graph classification architectures that were used with different node features in previous papers and provided good results, like DeeperGCN [48], EdgePool [49], and Kernel Graph-based CNN (KG-CNN) [50]. Additionally, edge-based graph neural methods like E-GraphSAGE [51] should be investigated since they can be used to detect malicious network flows and when combined with the GNN-Based FCGs technique. Another area of research might be checking the effectiveness of GNNs on zero-day malware. We explored GNNExplainer [52], an algorithm designed to derive insights from hidden layers of GNNs but were not able to since the current functionality is only designed for node classification problems and not graph classification. In the same vein, to understand how and why a GNN classifies malware the way it does, explainable GNN algorithms, such as SubgraphX [53] can be used. Lastly, end-to-end software for malware classification based on graph learning through source code would be a good addition to the current frameworks that exist for malware classification.
LIST OF REFERENCES


APPENDIX A

Figures Used for Analysis

In this section, we illustrate the accuracy curves for each parameter of the models. We plot them on the basis of the model. We also provide heat maps of the classification reports for the best parameters of each model. Some models do not have accuracy curves, and so for them, only classification reports are given.

Figure A.31: Comparison of accuracy curves extracted features vs. LDP features

Figure A.32: Example graphs each type: (a) AdDisplay, (b) Adware, (c) Benign, (d) Downloader, and (e) Trojan

Figure A.33: Classification report for MLP
Figure A.34: Classification report for WL Kernel Method

Figure A.35: Classification report for Feather

Figure A.36: Classification report for (a) LSD-Slaq (b) VNGE-Slaq
Figure A.37: (a) Comparison of accuracy curves of each parameter and (b) Classification report for GCN

Figure A.38: (a) Comparison of accuracy curves of each parameter and (b) Classification report for GraphSAGE

Figure A.39: (a) Comparison of accuracy curves of each parameter and (b) Classification report for GIN
Figure A.40: (a) Comparison of accuracy curves of each parameter and (b) Classification report for SGC

Figure A.41: (a) Comparison of accuracy curves of each parameter and (b) Classification report for JK-GCN

Figure A.42: (a) Comparison of accuracy curves of each parameter and (b) Classification report for JK-GraphSAGE
Figure A.43: (a) Comparison of accuracy curves of each parameter and (b) Classification report for JK-GIN

Figure A.44: Classification report for UnetGraph

Figure A.45: Classification report for DGCNN
Figure A.46: Confusion Matrix for (a) Feather and (b) WL-Kernel

Figure A.47: Confusion Matrix for (a) LSD-Slaq and (b) VNGE-Slaq

Figure A.48: Confusion Matrix for (a) MLP and (b) GCN
Figure A.49: Confusion Matrix for (a) GraphSAGE and (b) GIN

Figure A.50: Confusion Matrix for (a) SGC and (b) JK-GCN

Figure A.51: Confusion Matrix for (a) JK-GraphSAGE and (b) JK-GIN
Figure A.52: Confusion Matrix for (a) UnetGraph and (b) DGCNN
APPENDIX B
Tables Used for Analysis

Table B.7: Accuracy and Runtime Metrics for different number of iterations with WL Kernel

<table>
<thead>
<tr>
<th>Number of Iterations</th>
<th>Accuracy</th>
<th>Macro-F1 Score</th>
<th>Runtime</th>
<th>Number of CPU Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.714</td>
<td>0.705</td>
<td>138.15s</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>0.682</td>
<td>0.674</td>
<td>182.01s</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>0.621</td>
<td>0.606</td>
<td>262.5s</td>
<td>2</td>
</tr>
</tbody>
</table>

Table B.8: Accuracy and Runtime Metrics for different order parameters with Feather model

<table>
<thead>
<tr>
<th>Order</th>
<th>Accuracy</th>
<th>Runtime</th>
<th>Number of CPU Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0.839</td>
<td>672.78s</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>0.849</td>
<td>667.18s</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>0.847</td>
<td>680.34s</td>
<td>2</td>
</tr>
</tbody>
</table>

Table B.9: Accuracy and Runtime Metrics for different order parameters with Slaq-LSD model

<table>
<thead>
<tr>
<th>Number of vectors</th>
<th>Accuracy</th>
<th>Runtime</th>
<th>Number of CPU Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.775</td>
<td>314.40s</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>0.776</td>
<td>352.35s</td>
<td>2</td>
</tr>
<tr>
<td>20</td>
<td>0.780</td>
<td>401.01s</td>
<td>2</td>
</tr>
</tbody>
</table>

Table B.10: Salient Features of Papers Explored

<table>
<thead>
<tr>
<th>Paper</th>
<th>Method</th>
<th>Classification or Detection</th>
<th>Dataset</th>
<th>Accuracy/ F1 Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>[8]</td>
<td>Bayesian classification</td>
<td>Classification</td>
<td>Original dataset with 49 families and 2000 samples</td>
<td>84.50%</td>
</tr>
<tr>
<td>Reference</td>
<td>Methodology</td>
<td>Task</td>
<td>Dataset</td>
<td>Accuracy</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>------</td>
<td>---------</td>
<td>----------</td>
</tr>
<tr>
<td>[2]</td>
<td>Broad static analysis and SVMs</td>
<td>Classification</td>
<td>DREBIN</td>
<td>94.00%</td>
</tr>
<tr>
<td>[9]</td>
<td>Detection using significant permissions</td>
<td>Detection</td>
<td>Small and Large dataset with 5494 and 54694 samples each</td>
<td>93.62%</td>
</tr>
<tr>
<td>[10]</td>
<td>API features as word embeddings with LSTM</td>
<td>Classification</td>
<td>Android Genome and McAfee Labs dataset</td>
<td>89.70%</td>
</tr>
<tr>
<td>[7]</td>
<td>API features based on occurrence frequency and CNN</td>
<td>Detection</td>
<td>Original dataset with 3697 malware samples and 3312 benign samples</td>
<td>94.30%</td>
</tr>
<tr>
<td>[11]</td>
<td>Opcode-based n-gram features</td>
<td>Classification</td>
<td>Android Genome dataset and original dataset from Google play store with 40000 samples</td>
<td>89.31%</td>
</tr>
<tr>
<td>[12]</td>
<td>Opcode sequences encoded as one-hot vectors and CNN</td>
<td>Classification</td>
<td>Android Genome and two McAfee Labs dataset with 18000 samples</td>
<td>86.00%</td>
</tr>
<tr>
<td></td>
<td>Methodology</td>
<td>Detection</td>
<td>Dataset Description</td>
<td>Accuracy</td>
</tr>
<tr>
<td>---</td>
<td>------------------------------------------------------------------------------</td>
<td>-----------</td>
<td>-------------------------------------------------------------------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>13</td>
<td>Patterns based on ICC</td>
<td>Detection</td>
<td>Original dataset with 5264 malware and 12026 benign apps</td>
<td>97.40%</td>
</tr>
<tr>
<td>14</td>
<td>Resnet and GoogleNet</td>
<td>Detection</td>
<td>Microsoft Malware Challenge dataset</td>
<td>88.36%</td>
</tr>
<tr>
<td>15</td>
<td>Grey-scale vs. RGB images using Resnet</td>
<td>Detection</td>
<td>Andro-dumpsys dataset with 906 malware files across 13 families and 1776 benign apps</td>
<td>95.00%</td>
</tr>
<tr>
<td>16</td>
<td>API call-based HIN</td>
<td>Detection</td>
<td>Two Comodo Cloud Security Center datasets: 1800 and 20000 samples respective, about half are benign</td>
<td>99.67%</td>
</tr>
<tr>
<td>17</td>
<td>API call sequence graphs with permission features</td>
<td>Detection</td>
<td>DREBIN, AMD datasets</td>
<td>99.67%</td>
</tr>
<tr>
<td>18</td>
<td>API call based graph using GCN</td>
<td>Classification and Detection</td>
<td>Android Malware Genome Project, DREBIN and AMD</td>
<td>95.04%</td>
</tr>
<tr>
<td>Reference</td>
<td>Methodology</td>
<td>Type</td>
<td>Dataset Specifications</td>
<td>Accuracy</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>------</td>
<td>------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>[19]</td>
<td>Opcode-sequence as text features with Random forest and SVM</td>
<td>Detection</td>
<td>7500 malware and 7500 benign samples</td>
<td>97.00%</td>
</tr>
<tr>
<td>[20]</td>
<td>Co-opcode graph</td>
<td>Detection</td>
<td>Imbalanced dataset with 4000 samples from four families and 7000 benign samples</td>
<td>99.30%</td>
</tr>
<tr>
<td>[21]</td>
<td>Opcode-sequences as text features with LSTM</td>
<td>Classification</td>
<td>1796 malware and 1000 benign apps</td>
<td>97.00%</td>
</tr>
<tr>
<td>[22]</td>
<td>NF-GNN based on network flow features</td>
<td>Classification</td>
<td>2126 samples from 36 families</td>
<td>96.14%</td>
</tr>
<tr>
<td>[23]</td>
<td>CFGs using DGCNN</td>
<td>Classification</td>
<td>Microsoft Malware Classification Challenge dataset</td>
<td>99.42%</td>
</tr>
<tr>
<td>[24]</td>
<td>Graph embeddings with FCGs</td>
<td>Classification</td>
<td>AndroZoot, DREBIN and AMD</td>
<td>99.33%</td>
</tr>
<tr>
<td>[25]</td>
<td>FCGs using GIN</td>
<td>Classification</td>
<td>Malnet-tiny</td>
<td>90.00%</td>
</tr>
<tr>
<td>[29]</td>
<td>FCGs using Jumping Knowledge networks</td>
<td>Classification</td>
<td>Malnet-tiny</td>
<td>94.40%</td>
</tr>
</tbody>
</table>