Graph-Oriented Modelling of Process Event Activity for the Detection of Malware

*Regular Research Paper

1st Kenneth Brezinski†
dept. Electrical and Computer Engineering
University of Manitoba
Winnipeg, Canada
brezinkk@myumanitoba.ca

2nd Ken Ferens
dept. Electrical and Computer Engineering
University of Manitoba
Winnipeg, Canada
ken.ferens@umanitoba.ca

***Abstract*—This paper presents an approach to malware detection using Graph Neural Networks (GNN) to capture the complex relationships and dependencies between different components of an operating system (OS). Traditional methods for malware detection rely on known signatures of malware and may fail to detect new or modified malware variants. GNNs offer a promising solution by analyzing graph-structured data and identifying malicious behavior patterns. Specifically, this paper investigates the use of GNNs for malware detection based on the API call sequences of different event types, including File System, Registry, and File and Thread activity. The paper presents a representative dataset of host process activity of malware collected in a custom sandbox environment, comprising over 239 malware executions with randomly executed benignware samples. The paper then describes the GNN model trained on the dynamic process behavior generated from process execution graphs, with independent models developed based on each category of API events. Finally, the paper presents a trained model that maximizes the generalization performance of the model, demonstrating the applicability of GNNs for malware detection. This paper presents one of the first applications of GNN classification based on process hierarchy during malware execution that includes interaction with benignware as well .**

***Index Terms*—Malware Detection, Graph Neural Networks, Operating Systems, Application Programming Interface, Artificial Intelligence, Security**

I. Introduction

Malware detection is important due to the devastating consequences that malware can have on individuals, businesses, and governments. For instance, malware attacks can compromise sensitive data, steal financial information, and disrupt critical infrastructure. According to Malwarebytes 71% of companies worldwide were targeted by some for of ransomware attack in 2022, and on average Malware variants such as Emotet has cost state, local, tribal, and territorial governments up to 1 million USD per incident to remediate [1].

Some of the shortcomings of traditional signature-based and heuristic-based malware detection methods are that they rely on known signatures of malware in order to detect future variants. For instance, signature-based methods may fail to detect new or modified malware variants, while heuristic-based methods may generate many false positives. This emphasizes the need for a more effective malware detection approach as malware attacks have become more sophisticated and evasive, leading credence to a need for a more robust and accurate malware detection approach.

Graph Neural Networks (GNN) offer a promising solution to this problem by combining the ability to process large datasets with ease with discriminative power - all the while generalizing well to samples. GNNs are a type of neural network architecture that have gained popularity in recent years due to their ability to process and analyze graph-structured data, which are increasingly prevalent in various domains, including malware detection [2]. GNNs can be used to leverage the graph structure of an Operating System (OS) to capture the interactions between different system components and identify malicious behavior patterns. OS's are constantly under threat from malware, and traditional methods for malware detection often fail to keep up with the increasing sophistication of malware attacks. GNNs offer a promising approach for detecting and classifying malware based on their behaviors, as they can capture the complex relationships and dependencies between different components of the system.

One of such components is the event activity of a process, which can include the networking activity, edits of the registry, or the creation of a new file on disk or a new thread or Mutex. The motivation for investigating event activity is that the behavior of malware can be best characterized by the independent steps that it carries out, and this is best captured through investigation of the Application Protocol Interface (API) sequence [3, 4]. More specifically, this study can reveal behavioral-interaction-fingerprints that interacting processes have with the graph structure, which can be used to differentiate and classify different processes. In this paper we explore the use of GNNs for malware detection based on the API call sequences of different event types. Our work makes the following overarching contributions:

- a representative dataset is developed of host OS process activity collected in a custom sandbox environment that

†Corresponding Author; kbrezinski.github.io

comprises over 239 malware executions with randomly executed benignware samples with malware;
- a GNN model with attention is trained on the dynamic process behavior generated from process execution graphs, and independent models are developed based on File System, Registry and File and Thread activity;
- a trained model is developed that maximizes the generalization performance of the model, thereby demonstrating the applicability of GNNs for malware detection based on different categories of API events;

We will first provide an overview of GNNs in Section II and their application in various domains including malware detection. In Section III we introduce the sandbox used to collected malware and benignware, along with the API vectorization used and the architecture and design considerations for the GNN. This is followed by Section IV where the performance of a GNN-based malware detection systems on a real-world dataset representing some of the most recent malware variants of concern will be evaluated.

II. Related Works

Malware attacks continue to pose a significant threat to individuals, businesses, and governments, leading to devastating consequences such as data breaches, financial loss, and disruption of critical infrastructure [5]. Traditional signature-based and heuristic-based malware detection methods have limitations in detecting new and modified malware variants and generating false positives, highlighting the need for more effective approaches to malware detection [6].

GNNs have emerged as a promising solution for detecting malware by leveraging the graph structure of an operating system to capture the interactions between different system components and identify malicious behavior patterns [7]. GNNs can process and analyze graph-structured data with ease, making them well-suited for malware detection tasks. For instance, in [8], the authors proposed a GNN-based approach for malware detection that captures the structural and semantic features of the system call graph to identify malware samples based on use of Markov chains and Principal Component Analysis for feature extraction. The results demonstrated that their approach outperformed traditional machine learning-based methods in F1-Score. In total their work included 13,624 samples executed in a Cuckoo sandbox, which only includes the isolated samples executed without other processes running - which severely limits applicability as processes do not not run in isolation on host OSs. Similarly, in [9], the authors proposed a GNN-based approach that leverages the attention mechanism to capture the important features of the API call graph from Android APKs. The results showed that their approach achieved high accuracy and outperformed other machine learning-based methods, including a simple Graph Convolution Network (GCN) without attention. This use of attention will be explored in this work as well, as it has been shown that the use of attention in a GNN can dynamically learn the importance of nearby neighbors in a graph framework [10]

Recent studies have shown that event activity of a process, such as the API sequence, is a critical component for detecting and characterizing malware behavior [11, 12]. In particular, analyzing the API call sequence of different event types can reveal the independent steps carried out by malware and enable more accurate detection and classification. GNNs have been used to effectively capture the behavior of malware based on the API call sequences [8]. The authors used Cuckoo as well to generate the directed cyclic graph and weighted according to the proportion of API i called between nodes n and m. In this fashion a merged graph which satisfies the Markov property was generated and combined with the adjacency matrix of the API calls. In our previous work [13] API call sequences were fed into a transformer whereby the sequences of API calls were learned using self-attention. Self-attention is not to be confused with the attention mechanism described thus far, as self-attention relates to the learning of the sequences by propagating the vectors of the APIs with each other to learn contextual information [14, 13]. In [4], the authors grouped suspicious behavior into 9 categories, ranging from searching for file to infect to distributing virtual memory. All in all, the API calls associated with a particular behavior were modeled using finite automaton to trace suspicious behavior. An API tracer was used on 914 samples and following classification a final precision of $\approx$94% was obtained. Similarly to [8] the work focused on sequences provided by a single sample, as opposed to a robust host environment populated with benignware and malware. It is also the case that the work did not report Recall or F1 score, meaning the model may have very poor coverage of the totality of the malicious class.

Overall, the use of GNNs for malware detection based on the API call sequences of event types has shown great promise in recent years. While the applications of GNNs have been a profound benefit to the field of anomaly detection for malware detection, the datasets provided do not mimic real life host environments. This is due to the simple fact that tools such as Androguard or Cuckoo are already established frameworks by which API call graphs can be extracted with little setup. This does not provide the dynamism of a host OS which includes both benignware, OS processes as well as malware all running simultaneously [13]. For this reason, this work bridges that gap by introducing a representative dataset that incorporates the API call sequences of all processes simultaneously in a single model. This improves on existing work by expanding beyond a single process call graph and investigating the entirety of the API call sequences of the process hierarchy. The next section will discuss some of the innovations made in this respect on the generation of this dataset and the proposed GNN architecture for classification.

III. Methodology

This section will provide the theoretical basis for the development of a GNN architecture for the process classification of malware, as well as the representative dataset used to model a dynamic host environment complete with benignware and malware process activity. First, discussion of the sandbox

environment in Section III-A and how the malware samples were collected; followed by information on the sampling procedure used to generate a representative dataset in Section III-B. This follows into discussions on the feature vectorization of the API sequence in Section III-C followed by the GNN theory and architecture design in Section III-D and III-D, respectively.

A. Sandbox Process Execution Collection

Malware and benignware samples are collected in a custom sandbox environment as to simulate a real host environment. Careful attention is taken to ensure the dynamic behavior of malware is captured and cached for downstream tasks. To this end, we capture process event activity belonging to File System, Process and Thread Activity, as well as Registry events. Networking was not included as Networking activity uses similar networking sockets as benignware, and many malware, as a part of their anti-emulation procedure, fail to reach out to the network at all. In virtual environments many malware variants become Virtual Machine aware and fail to execute their viral payloads; therefore, the signatures of the malware which probe the current environment for signs of virtualization are as part of the malicious payload as the payload itself. Additionally, we wish to capture and detect anomalous behavior before secondary payloads are fetched from the internet - which further complicates the process of labelling processes. For the sake of brevity, we refer readers to our previous work in [13] which provides a complete overview of the sandbox environment and configuration.

B. malware Sampling and Run-time Configuration

The malware samples used for execution were drawn from a repository of recent malware samples obtained from VirusTotal[1]. VirusTotal provides researchers a repository of 10's of thousands of malware samples identified and bundled in the last quarter year to represent new and unique infections submitted to VirusTotal through an Academic License. In this work 239 malware samples were retrieved from Q4 2022. This dataset of malware samples was filtered to remove non-Windows malware, and includes both 32-bit and 64-bit executables. A full list of malicious executables, complete with file sizes and MD5 hashes, can be found in the Github repository for this work. Alongside malware, benignware is executed in tandem as to simulate a real host environment. During malware execution, 3 - 5 processes are randomly selected from the benignware list and run sequentially. This is to populate the execution graph with noise and negative training samples, and ensures the dataset mirrors the OS environment of a real host as closely as possible. In total, 300 benignware samples are collected from the `cnet.com` Apps for Windows category representing popular windows applications. All benignware was confirmed to have an AV score of 0 according to VirusTotal.

A systematic approach was used to stochastically sample negative training examples from the benignware category of executables. First, consider the set of malware samples M and benignware samples C. If each malware executable were to be executed once, then M total trials or executable graphs are being populated with process activity. Let c_m represent the subset of benignware executables $c_m \subset C$ for execution m. At each trial m, the chance of drawing a particular benignware sample is $(1/|C|)^{E[p_c]}$ with replacement; where $E[p_c]$ is the expectation of drawing samples with probability p_c. Therefore for M trials the chances of not drawing a particular benign sample is governed by Eq. 1; where the $|C| - i$ term takes into account the fact that replacement can not occur as no executed can be executed twice on a clean snapshot. This exercise is simply to demonstrate the case that some benignware samples are not used to train the model based on the sampling technique used. This provides a good generalization of potential host environments without introducing bias into the dataset if the same processes were chosen manually for each malware execution graph.

$$M \prod_{i=1}^{E[p_c]} \left(1 - \frac{1}{|C| - i}\right) \tag{1}$$

In the sandbox environment the samples were automatically run through the use of a batch script which helps to automate the process and provide consistent executions between malware samples in terms of time window. A short script is used to run *Procmon* and load relevant configuration files and filters to begin collection. The filter files (`.pmc` files) are used to exclude some Procmon specific events from appearing in the list of captured events, but all other events, including windows operating system behaviour, is captured. This ensures there is no bias introduced in the collected event activity by the author. A list of the rules used in the *Procmon* filter can be found in Table IV Appendix V-0c.

C. API Call Sequence Vectorization

In this work process APIs were vectorized according to a simple scheme combining both N-*grams* with *tf-idf*. To begin, N-grams looks at all the unique n number sequences of APIs, and creates a feature vector with the numeration of the unique sequence. N-grams improves on Bag of Words by accounting for pairs - in the case of bigrams - or trigrams in the case of 3, of unique API sequences of length n. These sequences are taken from the stack traces, where they appear in order from low-memory space to high-memory space (`0xfffff`) in the stack. In Fig. 1 we see an illustration for a stack trace that has been resolved using the Windows Symbol Table to produce the names of the API calls [3]. In this example `LoadLibraryExa` and `LoadLibraryA` appear after one-another, therefore the tuple `(LoadLibraryExa, LoadLibraryA)` is added to the corpus. Then the next element in the sequence would include `BaseThreadInitThunk` which would be added to the corpus with the previous element as `(LoadLibraryA, BaseThreadInitThunk)`. This captures all unique combinations of Windows APIs, and maintains some of the order in the sequences. When considering trigrams we would

[1]www.virustotal.com/

incorporate more information of the sequence by considering all combination of three ($n = 3$) APIs in succession.

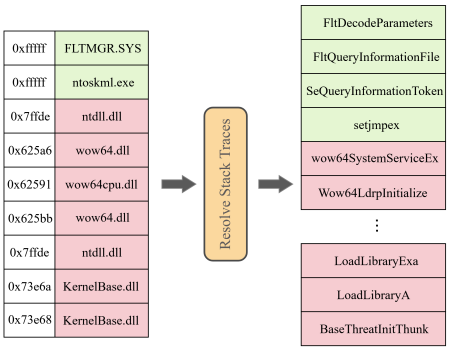

Fig. 1. Translation from memory locations of imported DLLs to Windows API function calls. Retrieved from [3].

The decision to select larger n comes down to data sparsity. In a previous work the number of Windows APIs used by event processes were noted to be in the range of 1500-1700 [3]; which is a relatively low number compared to typical Natural Language Processing (NLP) problems. Larger values of n mean larger and larger sequences which will occur less frequently, meaning the model may overfit on the training data as it will generalize poorly. Conversely, a smaller n will lose the ability to trace longer sequences and will be unable to learn the importance of classifying maliciousness, thereby causing the model to underfit.

Term Frequency–Inverse Document Frequency (tf-idf) takes the vectorization one step further by attempting to learn the importance of words by considering their inverse-frequency of occurrence. The motivation for this technique is that more frequent APIs are less important in distinguishing maliciousness, while greater attention should be paid towards less frequent APIs. Theoretically this technique would be more powerful in attending to rarely used APIs that malware will use but benignware does not, while at the same time, disregarding commonly used APIs that are routine to the functionality of any running process. Calculating tf-idf is a straightforward process and can be done according to Eq. 2.

$$idf_{td} = tf_{td} \times \log\left(\frac{N}{df_t + 1}\right) \tag{2}$$

Where tf_{td} accounts for the proportion of the API, t, used by the process d by dividing the count of t by the total number of APIs used by d. df_t keeps track of the number of occurrences of t by all N processes. The logarithm has the effect of dampening the explosion of the numerator for high values for N. When df_t gets larger for more frequent APIs, the term in the logarithm approaches 1 and the idf_{td} is small. When df_t is small for rarely used APIs the idf_{td} becomes larger and is given a larger weight. Through this preprocessing step APIs can be vectorized in such a way where malicious API usage can have a greater impact for process classification.

D. Introduction to Graph Neural Networks

Node feature vectors are prepared from a directional flow graph G. We define V as the set of vertices on the graph which represent spawned processes on a host OS. Each vertex has a set of node features, $\mathbf{h} = \{\bar{h}_1, \bar{h}_2, \ldots, \bar{h}_N\}, \bar{h}_i \in \mathbb{R}^F$ where N is the number of nodes in the graph and F is the number of features for each node. In Fig. 2 we observe four nodes that are neighbors. For example, if $\bar{h}_2$ is the node under consideration it shares a neighbour with $\bar{h}_2$ and $\bar{h}_3$ but not with $\bar{h}_4$. Therefore, for any ith node we want to consider the influence of its neighboring nodes $j \in N_i$.

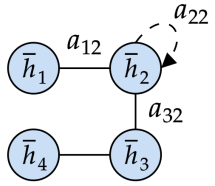

Fig. 2. Graph structure of a 4 node cluster, complete with attention coefficients α_{ij} between any given nodes i and j.

In Fig. 2 we also observe coefficient a_{ij}, where i, j correspond to the indices of neighboring vertices, which acts as an attention mechanism, known as the *attention coefficient*, which dictates the amount of influence to be paid between neighboring nodes. In Section III-F we will discuss the role these coefficients play and how they are computed. For now we will explore how we can make use of our feature vectors through the the use of linear projections.

E. Projecting Higher-Order Features

To create a higher-order representation of our features we apply a simple linear transformation. In Fig. 3 the feature matrix, $\mathbf{h}$, is created using a simple *Bag-of-Words* (BoW) feature representation by numerating whether or not a node used a particular Windows API or not. This representation is used just for demonstrative purposes. If a process uses the particular API call it is incremented by 1, otherwise it is given the default value of 0 meaning it was not used. Using a *weight matrix*, $\mathbf{W} \in \mathbb{R}^{n \times E}$ where E is the embedding dimension and n is the number of features, we can compute the embeddings for node j using the dot product between $\mathbf{W}$ and $\bar{h}_j$ - producing $\mathbf{W}\bar{h}_j$. These embeddings are learned and provide the model the ability to learn which features - in this case Windows APIs - are important in considering the maliciousness of nodes.

F. Attention Coefficient

The *attention coefficient* combines the influence of nearby nodes and allows every node to attend to every other node on the graph framework. The expression for calculating a_{ij} between nodes i and j is shown in Eq. 3; where $\bar{w}^T$ is a trainable weight vector that is transposed, $||$ signifies the double pipe operator which stacks the transformed feature vectors $\mathbf{W}\bar{h}$

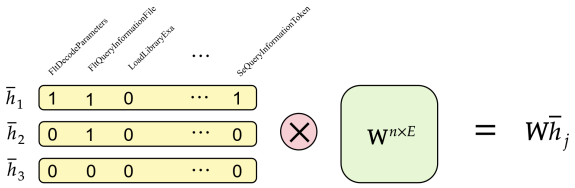

Fig. 3. Creation of the node embeddings via the dot product between the BoW API matrix and the weight matrix **W**.

along the second axis, and $LeakyReLU()$ applies the leaky $ReLU$ activation function.

$$a_{ij} = \frac{\exp\left(LeakyReLU\left(\bar{w}^T\left[\mathbf{W}\bar{h}_i||\mathbf{W}\bar{h}_j\right]\right)\right)}{\sum_{k\in N_i}\exp\left(LeakyReLU\left(\bar{w}^T\left[\mathbf{W}\bar{h}_i||\mathbf{W}\bar{h}_k\right]\right)\right)} \quad (3)$$

The use of $leakyReLU$ has the effect of only considering the positive influences of the node feature vectors, while allowing the weight to possibly recover it were driven to less than 0 where the derivative evaluates to 0 in a normal $ReLU$. Additionally, the use of $\exp()$ in the numerator and denominator implements a $softmax$ which ensures the *attention* coefficients for node $a_{ij} \in [0, 1]$ are normalized; as the denominator will always evaluate to 1 when considering all possible neighboring nodes N_i for node i. When we compute a_{ij} for all neighboring nodes, we can then proceed to calculate the transformed feature representation according to Eq. 4. Therefore all neighbors to i, j, where $j \in N_i$ are considered through their respective a_{ij} values; with a final $sigmoid$ being used to introduce non-linearity. A less verbose explanation and original derivation of GATs can be found in the original work [10].

$$\bar{h}_i' = \sigma\left(\sum_{j\in N_i} a_{ij}\mathbf{W}\bar{h}_j\right) \quad (4)$$

G. Graph Neural Network Architecture

Understanding the topology of the process execution is important as a preliminary step for several reasons. First, process graph topology is fundamentally different than that of a social network or a citation network, and knowledge of the topology leads to better decision making in the architecture design. For example, if processes on average spawn 1 to 2 daughter processes, then creating a deeper GNN based on 3-hop neighbours (a 3-layer GNN) would be redundant and lead to over saturation - or a smoothing effect over the embedding space. As an example of this impact, lets denote any ith process p_i^j where j refers to the hierarchy of the process, or depth if one were to consider some form of n-ary tree. If a user uses `explorer.exe`, denoted as p_i^1, to execute a malware executable p_i^2, then we can use GNNs to share the event behaviour of any process $p_i^2 \rightarrow p_i^1$ via message passing. However, the OS parent process that spawned `explorer.exe`, lets call it p^0 which acts as a root node, would not be helpful to understand the behaviour of p_i^2 since there are many processes that are Benign which share an ancestor with p_i^2. As a result, all p_i^2 processes that exist would share the same embedding information from p^0, leading to all the process nodes of p_n^2 becoming saturated with similar information. This would impact performance negatively, and is best resolved through domain knowledge of the dataset used and carefully considering the amount of capacity of your model. A visualization of this effect is shown in Fig. 4. We can see the difference in a 1-hop aggregation (Fig. 4(left)) and a 2-hop aggregation (Fig. 4(right)) whereby the malicious red node p_1^2 is smoothed over with information from p^0, leading to a node embedding that is more similar to that of its benign counterpart p_2^2. This is illustrated by the proportion of red, orange and green in its node embedding, which geometrically would coincide with the vectors p_1^2 and p_2^2 being closer or farther apart in d-dimensional space.

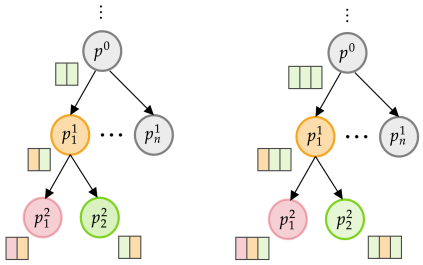

Fig. 4. Process hierarchy for a directed graph, where the red, orange and green coloured nodes refer to malicious, suspicious and benign processes, respectively. Crude illustration of node embeddings after message passing and aggregation for (left) 1-hop neighbourhood and (right) 2-hop neighbourhood.

Additionally, for each feature set belonging to each event type, a corpus is created for the API calls based on the discussion in Section Section III-C. In Table I we can view the size of each corpus, which is the size of the node feature vector which is being fed into the model. Additionally, Table I provides the corpus sizes for 1 2 and 3-grams - representing all unique combinations of n APIs in sequence. What is noteworthy is that the corpus sizes are much larger that previous related work in [3] which recorded corpus sizes in the 1500-1700 range. The previous work only looked at the stack traces of 9 malware executables, while this work looks at over 200; meaning we are covering a more recent swathe of potential threats and behaviours in our dataset. The previous work additionally did not look at benignware executables - which further exemplifies the novelty and the necessity of this work. Looking at longer sequences has the benefit of encoding more useful information of the behaviour of the process via long-range dependency, at the cost of introducing noise and added dimensionality and complexity. The corpus sizes in Table I directly relate to the size of the initial input dimensions in the GNN architecture.

In a GNN architecture the model needs to learn from various topologies in order to learn the contexts of malicious and benign node frameworks. For this reason the model will be trained in a *deductive setting*, where nodes are batched into training and validation sets (80/20 split) as shown in Fig. 5. For a given topology the model will be fed with a list of edges, $\mathbf{E}^{2\times|E|}$,

TABLE I
SUMMARY TABLE OF THE NUMBER OF API CALLS BELONG TO THE CORPUS FOR DIFFERENT n GRAMS. THE **EVENTTYPE** *All* REFERS TO A COMBINATION OF REGISTRY, FILE AND THREAD EVENT APIS INTO A SINGLE CORPUS.

Event	1-gram	2-gram	3-gram
Registry	1917	6517	10985
File	2017	6018	9669
Thread	612	1506	2200
All	2921	10656	18267

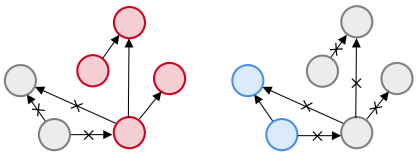

Fig. 5. Training (left) and Validation (right) sets for nodes randomly selected at each iteration.

derived from an adjacency list connecting source nodes to destination nodes using PIDs and PPIDs, respectively. Based on the aforementioned considerations for GNN design discussed in this section, a summary of GNN model parameters can be found in code snippet 1 in Appendix V with additional details for training configuration.

IV. RESULTS AND DISCUSSION

Based on the aforementioned design considerations outlined in the previous section, an overview of the malware topologies is briefly covered in Section IV-A. This is followed by the main performance metrics compared for various datasets and model architectures in Section IV-B; followed by some final discussions on model variance in Section IV-C.

A. Node Topology and Feature Characteristics

This section briefly summarizes the structure and attributes of the Sandbox execution graphs, first through through a discussion of network metrics as it pertains to the topology of the sandbox Malware execution graphs. A table summarizing the network metrics is shown in Table II. A summary of these network statistics and their meaning follows. Many of these network statistics play a role in the architecture design of the GNN as understanding the network is a precursor to establishing effective message passing layers.

a) Avg. Node Degree: This is the average degree of a node in the network. The degree of a node is the number of edges connected to it. In this case, the average node has a degree of 6.70, meaning the process topology is higher interconnected.

b) Min Node Degree: This is the minimum degree of a node in the network. In this case, the minimum degree is 2.33.

c) Max. Node Degree: This is the maximum degree of a node in the network. In this case, the maximum degree is 11. This node would belong to a process such as `cmd.exe` or `explorer.exe` in the case of a execution initiated by the user using the OS Graphical User Interface (GUI).

TABLE II
SUMMARY NETWORK STATISTICS FOR THE SANDBOX MALWARE EXECUTION GRAPHS. VALUES ARE AVERAGED OVER ALL MALWARE EXECUTIONS ($n = 239$).

Metric	Value
# Nodes	40
# Edges	112
Avg. Node Degree	6.70
Min Node Degree	2.33
Max. Node Degree	11
Avg. Degree Connectivity	4.75
Degree Centrality	0.14
Degree Assortativity	-0.20
Density	0.07
Square Clustering	0.32

d) Avg. Degree Connectivity: This is the average degree connectivity of a node in the network. Degree connectivity is a measure of how well connected a node is to its neighbors. In this case, the average degree connectivity is 4.75. The degree connectivity of a node is calculated as the number of neighbors of the node with degree greater than or equal to the degree of the node, divided by the total number of neighbors of the node. For example, if a node has four neighbors with degree 2, two neighbors with degree 3, and one neighbor with degree 4, its degree connectivity would be 7/7 = 1. This is in contrast to Avg. Node Degree, which is simply the sum of the degrees of all nodes in the network, divided by the number of nodes.

e) Degree Centrality: This is a measure of the importance of a node in the network based on its degree. In this case, the degree centrality is 0.14. Degree centrality can be calculated as the degree of a node divided by the maximum possible degree in the network. For example, in a network with 10 nodes, the maximum possible degree is 9 (since a node cannot be connected to itself). If a node has a degree of 6, its degree centrality would be 6/9 = 0.67. Given the highly inter-connectivity of the graph, a value of 0.14 is fairly high.

f) Degree Assortativity: This is a measure of the degree homophily in the network. Degree homophily is the tendency of nodes to connect to other nodes with similar degree. In this case, the degree assortativity is -0.20, which means that nodes with high degree are more likely to connect to nodes with low degree, and vice versa. This value would be larger, or even positive, in the case of a larger network topology where servers are communication with each other to form of the backbone of the internet. In the case of process execution, there is no such behaviour to be observed as core OS behaviour tends to spawn daughter processes, but not other core OS processes.

g) Density: This is a measure of the number of edges in the network relative to the maximum number of edges that could exist. In this case, the density is 0.07, which means that there are relatively few edges in the network. Once process execution is carried out, then it does not make intuitive sense for the daughter process to spawn a process already running. It is even the case that through the use of mutexes processes are not allowed to execute once already running.

h) Square Clustering: This is a measure of the clustering coefficient of the network. The clustering coefficient is a measure of the number of triangles in the network. In this case, the square clustering is 0.32, which means that there is a relatively high level of clustering in the network. This follows from the Degree Connectivity metric that noted the high inter-connectivity of the network.

B. Graph Neural Network Performance on Sandbox Execution Graphs

Table III summarizes the validation loss, as well as balanced accuracy and F1 Score for several GNN architectures and datasets. For a description of these performance metrics, the authors refer the reader to Section V. First and foremost, the comparative performance for the different event types is shown in the upper section of Table III. The File System event type was the best performing feature dataset across the board (93.64 vs 91.81 and 92.72 **F1**). This coincides with the fact that File System usage by malware is characteristically different than benignware. This can be due to several reasons, but mainly the creation and deletion of files on disk occurs with an irregularity that isn't the case for benignware as malware attempts to cover it tracks on the OS. Its also the case that for Registry events certain querying and alteration of registry keys is indicative of persistence which occurs for both benignware and malware as many processes look to set registry keys upon startup. It is also the case that other event type usage are too similar between malware vs benignware, albeit by a small margin as the results all show comparative performance within error. This is the case with the sole exception of **GAT-All** which combines all the APIs from all event types. The GNN model loses the ability to learn the relationships between processes when all event types are used as it attempts to learn a general representation of behavior as opposed to one focused on a single event type. It is also the case that a larger feature representations are harder to learn, and may require running for more epochs and model fine-tuning.

Additionally, unigram was tested and compared with both a bigram and trigram model. The unigram feature representative was optimal by a large margin (96.39 vs 86.96 and 84.91 in Accuracy), and this can mainly be attributed to over-fitting. While longer sequences do encode more information about the behavior of the malware through a sequence of APIs, it has the side effect of relying too much on certain sequences and generalizing poorly to the validation set. Better regularization is one technique which can minimize the effect of over-fitting, which in this work was already implemented with L_2 regularization. Also, a larger feature representation is also more difficult to train and has similar shortfalls as mentioned previously with the **GAT-All** performance. Aside from the issue with over-fitting, a unigram model may be optimal in cases where the single use of APIs is salient enough such that the model learns malicious behavior based on their usage and frequency of usage. Since *tf-idf* was used, rarely used APIs are given a greater weight and for bigram and trigram the rarity of certain sequences penalizes the model more than it benefits based on these results.

TABLE III
SUMMARY MODEL PERFORMANCE METRICS FOR THE VALIDATION SET. METRICS ARE COMPUTED OVER 20 ITERATIONS. ELEMENTS **BOLDED** SIGNIFY THE BEST PERFORMING MODEL FOR EACH DATASET. NAMING SCHEME $model$-l-d REFERS TO THE $model$ (GAT - GRAPH ATTENTION; GCN - GRAPH CONVOLUTION W/O ATTENTION; ANN - LINEAR LAYER) WITH l LAYERS AND d HIDDEN NEURONS IN EACH LAYER. UNLESS OTHERWISE NOTED, THE MODELS USE THE FILE SYSTEM EVENT TYPE WITH A *unigram* VECTORIZATION WITH AN $l = 2$ AND $d = 64$.

Model	Loss ↓	Accuracy (%) ↑	F1 (10^{-2}) ↑
GAT-File	**1.05 ± 0.4**	**95.87 ± 1.9**	**93.64 ± 2.2**
GAT-Registry	1.19 ± 0.7	94.85 ± 2.2	91.81 ± 2.6
GAT-Thread	1.84 ± 0.3	95.22 ± 2.2	92.72 ± 2.0
GAT-All	5.02 ± 0.3	75.23 ± 2.0	93.29 ± 2.2
unigram	**1.14 ± 0.5**	**96.13 ± 1.0**	**96.39 ± 0.1**
biggram	1.31 ± 0.3	86.96 ± 1.0	95.15 ± 1.1
trigram	1.39 ± 0.2	84.91 ± 1.3	95.55 ± 1.2
GAT-2-32	1.27 ± 0.3	94.62 ± 1.0	94.91 ± 0.4
GAT-2-64	1.14 ± 0.3	**96.13 ± 1.6**	**96.39 ± 1.6**
GAT-2-128	**1.05 ± 0.3**	96.00 ± 2.5	90.92 ± 2.2
GAT-3-64	1.40 ± 0.4	94.53 ± 1.1	95.15 ± 1.2
GCN-2-64	1.60 ± 0.4	94.76 ± 1.0	89.98 ± 1.0
ANN-2-64	1.88 ± 0.3	90.44 ± 0.8	82.98 ± 1.0

Finally, an investigation on different model depths and capacities was carried out. This included investigating 2-hop and 3-hop neighborhoods (noted using the value of l for the number of hops), as well as different hidden neurons in each layer, d. Table III demonstrates 128 hidden neurons does lead to better loss but not necessarily an improvement in metric score that is statistically significant. This is because the model converges during training and the model does not learn anything new with additional capacity. In addition, 2-hop and 3-hop neighborhoods were tested (GAT-2-64 and GAT-3-64), and a 3 layer GNN ($l = 3$) noted worse performance in loss and accuracy (1.40/94.53 vs. 1.14/96.13, respectively). As previously discussed in Section III-D, deeper GNNs leads to over-saturation of the embedding space as malicious processes incorporate information of their benign counterparts. For comparison purposes and as a baseline, we compared results for a simple artificial neural network (ANN-2-64) which contains no message passing aggregation along with a GNN without attention (GCN-2-64). For both types the model had diminishing performances as the message passing is pertinent in the training of embeddings for a GNN which the ANN does not have; and the attention mechanism can learn the importance of nearby neighbors which a simple graph convolution does not have.

C. Discussion on Model Variance

One takeaway from the model performances is that models have larger variances between model runs. This is evident in the high standard deviations in model loss, accuracies and F1 Scores shown thus far in this section. The reason for this is due to the sampling procedure used and some of the considerations touched on in Section III-B. Some of the process execution

graphs are populated with benignware activity that is either executed twice in two different graphs and some samples which are not present at all. This has the effect of the model learning very different execution graphs which adds to the robustness of the model but also leads to large deviations between iterations. Secondly, the largest source of model variance is the selection of the training and validation set. To overcome effects due to sampling bias (i.e. the manual selection of samples to validate on which introduces bias), the positive and negative training examples that fall into the validation set are randomly selected for each of the iterations that a new model is trained. These iterations are different than epochs: where at each iteration a new model is initialized with random weights, a randomized training and validation set, and a new random seed; whereas during each epoch the model is trained and validated on the same model and training and validation set. So for each iteration the model experience is very different and this can lead to significant model variance. This is because depending on the iteration, if a hard to classify malicious executable is presented in the validation set, it would not have been trained on in the training set, and thus be hard to classify. In another iteration it would be present in the training set, in which case it would not belong to the validation set and the performance scores would improve. Repeated iterations tend to smooth over this effect as the effect averages out, but this presents a persistent effect due to the nature of sampling over 200 independent process execution graphs with heterogeneous topologies. This work already applies Regularization and Early Stopping to ensure the validation performance is as close as possible to the training performance to avoid over-fitting [15, 16]. One other remedy is to increase the size of the dataset which is always a solution to problems with over-fitting leading to poor generalization in deep learning research. But largely due to the time commitment and in other cases the cost of manually labelling training examples, this is infeasible and impractical in practice [17, 18, 19].

V. Acknowledgements

This research has been financially supported by Mitacs Accelerate (IT15018) in partnership with Canadian Tire Corporation,and is supported by the University of Manitoba.

References

[1] Malwarebytes. 2023 State of Malware. Technical report, Santa Clara, CA, 2023.

[2] Yakang Hua, Yuanzheng Du, and Dongzhi He. Classifying Packed Malware Represented as Control Flow Graphs using Deep Graph Convolutional Neural Network. In *2020 International Conference on Computer Engineering and Application (ICCEA)*, pages 254–258, March 2020.

[3] Kenneth Brezinski and Ken Ferens. Transformers - Malware in Disguise. In *Transactions on Computational Science & Computational Intelligence*. Springer Nature, ACC4507; Accepted, In Press, 2021.

[4] Cheng Wang, Jianmin Pang, Rongcai Zhao, Wen Fu, and Xiaoxian Liu. Malware Detection Based on Suspicious Behavior Identification. In *Proceedings of the 2009 First International Workshop on Education Technology and Computer Science - Volume 02*, ETCS '09, pages 198–202, USA, March 2009. IEEE Computer Society.

[5] Kenneth Brezinski and Ken Ferens. Metamorphic Malware and Obfuscation -A Survey of Techniques, Variants and Generation Kits. *Security and Communication Networks*, 2023.

[6] Daniel Gibert, Carles Mateu, and Jordi Planes. The rise of machine learning for detection and classification of malware: Research developments, trends and challenges. *Journal of Network and Computer Applications*, 153:102526, March 2020.

[7] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, January 2020.

[8] Shanxi Li, Qingguo Zhou, Rui Zhou, and Qingquan Lv. Intelligent malware detection based on graph convolutional network. *J Supercomput*, 78(3):4182–4198, 2022.

[9] Cagatay Catal, Hakan Gunduz, and Alper Ozcan. Malware Detection Based on Graph Attention Networks for Intelligent Transportation Systems. *Electronics*, 10(20):2534, January 2021. Number: 20 Publisher: Multidisciplinary Digital Publishing Institute.

[10] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. *arXiv:1710.10903 [cs, stat]*, February 2018. arXiv: 1710.10903.

[11] Tristan Bilot, Nour El Madhoun, Khaldoun Al Agha, and Anis Zouaoui. A Survey on Malware Detection with Graph Representation Learning, March 2023. arXiv:2303.16004 [cs].

[12] Youngjoon Ki, Eunjin Kim, and Huy Kang Kim. A Novel Approach to Detect Malware Based on API Call Sequence Analysis. *International Journal of Distributed Sensor Networks*, 11(6):659101, June 2015. Publisher: SAGE Publications.

[13] Kenneth Brezinski and Ken Ferens. Sandy Toolbox: A Framework for Dynamic Malware Analysis and Model Development. In *Transactions on Computational Science & Computational Intelligence*. Springer Nature, SAM4213; Accepted, In Press, 2021.

[14] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[15] Hwanjun Song, Minseok Kim, Dongmin Park, and Jae-Gil Lee. How does Early Stopping Help Generalization against Label Noise?, September 2020. arXiv:1911.08059 [cs, stat].

[16] Rich Caruana, Steve Lawrence, and C. Giles. Overfitting in Neural Nets: Backpropagation, Conjugate Gradient, and Early Stopping. In *Advances in Neural Information*

Processing Systems, volume 13. MIT Press, 2000.
[17] Xue-Wen Chen and Xiaotong Lin. Big Data Deep Learning: Challenges and Perspectives. *IEEE Access*, 2:514–525, 2014. Conference Name: IEEE Access.
[18] Douglas Heaven. Why deep-learning AIs are so easy to fool. *Nature*, 574(7777):163–166, October 2019. Bandiera_abtest: a Cg_type: News Feature Number: 7777 Publisher: Nature Publishing Group Subject_term: Computer science, Information technology.
[19] John D. Kelleher. *Deep Learning*. MIT Press, September 2019. Google-Books-ID: b06qDwAAQBAJ.
[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *arXiv:1512.03385 [cs]*, December 2015. arXiv: 1512.03385.
[21] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, March 2010. ISSN: 1938-7228.
[22] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.
[23] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv:1412.6980 [cs]*, January 2017. arXiv: 1412.6980.
[24] Ilya Loshchilov and Frank Hutter. Decoupled Weight Decay Regularization. *arXiv:1711.05101 [cs, math]*, January 2019. arXiv: 1711.05101.

APPENDIX A - GRAPH NEURAL NETWORK ARCHITECTURE AND TRAINING CONFIGURATION

Model Weights and Initialization

Weight matrices were all uniformly distributed using He initialization [20, 21]. He initialization uses a uniform distribution ($W^l_{i,j} \approx N(\mu = 0, \sigma^2 = 0.01)$) to randomly initialize the weights in a range that is determined by the number of input and output units in the layer. Additionally, *Dropout* is used to regulate the network to prevent the model from over-fitting [22]. Unlike in [10] where the authors implemented *dropout* of the original feature vector $\bar{h}_j$, in this work the sparsity of the vectorized API calls makes it so that *dropout* would have a minimal effect. *Dropout* is however applied after the linear projection and after determining the attentions coefficients, as this is typical in a deep neural network to regulate over-fitting and prevent the network from relying on any particular set of weights. So for each forward propagation and at each layer, a binary mask is set for each input unit j, which is drawn with probability p where $r^l_j \sim Bernoulli(p)$ at each layer l. Each input layer is then masked with r^l_j where $\tilde{\mathbf{x}}^l := \mathbf{x}^l \odot r^l_j$ to produce an output with fraction p of layers set to 0. Dropout probability for this work was tested at $p \in \{0.2, 0.5\}$. Since the model learns with a proportion of layers p set to 0, during inference the model output needs to be scaled by a proportional amount equal to $\tilde{\mathbf{x}}^l := \mathbf{x}^l \odot (1 - p)$ to account for the scaled back activations which the model is accustomed with during training. The other considerations for model capacity are the depth of the GNN n (illustrated in Fig. 4) and the number of hidden neurons in each layer d. A sample sequential layer for a GNN which encodes information for the 2-hop neighborhood ($n = 2$) is shown in code listing 1. The code listing represents a sample architecture in which all of the models outlines in Table III is based upon.

```
GNN(
  (layers): ModuleList(
    (1): GATConv(in_features=input_dims, out_features=d)
    (2): LeakyReLU(alpha=0.2)
    (3): Dropout(p=0.2)
    (4): GATConv(in_features=d, out_features=d)
    (5): LeakyReLU(alpha=0.2)
    (6): Linear(in_features=d, out_features=2)
  )
)
```

Code Listing 1. Sequential GNN architecture for evaluating Malware detection performance for experimental results. $input\_dims$ is the size of the input corpus, which is tabulated in Table I; d is the number of hidden neurons, p is the dropout probability and alpha is the negative slope in the $LeakyReLU$ activation.

Additionally, the *LeakyReLU* activation function which was set with a negative slope $\alpha = 0.2$; where the function evaluates to αx when the input value $x < 0$, otherwise it evaluates to x in the piece-wise function. This function has the advantage of producing large gradients when needed similar to *ReLU*; with the added advantage that gradients do not die out as the gradient can still recover when $x < 0$.

Performance Metrics

a) Loss: is defined in this work as the Cross-Entropy loss which is used to measure the difference between the predicted probability distribution and the true probability distribution of the target variables. The equation to calculate Cross-Entropy is shown in Eq. 5 for C classes for the true and predicted probability y_i and $p(y_i)$, respectively, for class i. In this case y_i is the one hot encoded label vector. Crossentropy and Eq. 5 is a generalized form of the Binary Cross-Entropy, which only applies to predictions with 2 classes.

$$L_{CE} = \sum_{i=1}^{C} c_i * y_i \log(p(y_j)) \tag{5}$$

b) Accuracy: in this work is calculated as the macro-average of the accuracies whereby the arithmetic mean of each individual class is taken into account. Therefore, for C classes, the macro-average accuracy is calculated according to Eq. 6. The expression for Eq. 6 has the advantage of calculating the accuracy for each class independently, meaning class-imbalance is accounted for in the metric according the number of training examples for each class N_c.

$$Average_{macro} = N_1 * Acc_1 + N_2 * Acc_2, \ldots + N_C * Acc_C \tag{6}$$

c) F1-Score: is a performance metric, where $F1\text{–}Score \in (0, 1)$, that is used to evaluate the accuracy of a classifier. It is defined as the harmonic mean of precision and recall, where precision is the fraction of True Positive predictions among all positive predictions, and recall is the fraction of True Positive (TP) predictions among all actual positive instances. F1-score can be calculated according to Eq. 7; where $precision = TP/(TP+FP)$ and $recall = TP/(TP+FN)$; where FN is False Negative and FP is False Positive. This metric accounts for class imbalance, providing a meaningful indicator of model performance on both the malicious and benign training examples.

$$F1\text{–}Score = 2 \times \frac{precision \times recall}{precision + recall} \tag{7}$$

Training Configuration

The *Adam* optimizer was used as the training optimizer which combines stochastic gradient descent with momentum with RMSProp [23]. Readers can refer to the original paper for the procedure, or the Pytorch documentation for an overview of the algorithm. The optimizer's β parameters were set to $(0.9, 0.999)$, with a learning rate $\eta \in \{5 \times 10^{-2}, 1 \times 10^{-2}, 5 \times 10^{-3}\}$. L_2 regularization was also implemented through weight decay λ at 10% of the learning rate [24]. So for a η of 5×10^{-3} the decay rate would be 5×10^{-4}. In the Adam optimizer, the decay rate is multiplied by the weights w_t at iteration t which penalizes the magnitude of the weights at each epoch. Models were run for 500 epochs each, where one epoch is equivalent to one pass through of each of the execution graphs. For statistical analysis models were trained for 20 iterations, where one iteration corresponds to re-initialization of the model parameters and optimizer and randomization of the training/validation split.

APPENDIX B - PROCESS MONITOR FILTER CONFIGURATION

In Table IV a summary of the allow/disallow list for the Procmon configuration is presented. The filters remove processes belonging to the Procmon application itself, as well as some auxiliary files and formats.

TABLE IV
PROCESS MONITOR CONFIGURATION FOR FILTERING OUT RELEVANT EVENTS. WITH THE EXCEPTION OF THE FINAL ENTRY, ALL OTHER ENTRIES ARE DISALLOW FILTERS.

Entity	Relation	Value
Process Name	is	`Procmon.exe`
Process Name	is	`Procexp.exe`
Process Name	is	`Autoruns.exe`
Process Name	is	`Procmon64.exe`
Process Name	is	`Procexp64.exe`
Operation	begins with	`IRP_MJ_`
Operation	begins with	`FASTIO_`
Result	begins with	`FAST IO`
Path	ends with	`pagefile.sys`
Path	ends with	`$Mft`
Path	ends with	`$MftMirr`
Path	ends with	`$LogFile`
Path	ends with	`$Volume`
Path	ends with	`$AttrDef`
Path	ends with	`$Root`
Path	ends with	`$Bitmap`
Path	ends with	`$Boot`
Path	ends with	`$BadClus`
Path	ends with	`$Secure`
Path	ends with	`$UpCase`
Path	ends with	`$Extend`
Event Class	is	`Profiling`[a]

[a] alternates between Process, Network, File System and Registry depending on which events are to be collected.