Guided Malware Sample Analysis Based on Graph Neural Networks

Chen, Yi Hsien; Lin, Si Chen; Huang, Szu Chun; Lei, Chin Laung; Huang, Chun Ying

DOI
10.1109/TIFS.2023.3283913

Publication date
2023

Document Version
Final published version

Published in
IEEE Transactions on Information Forensics and Security

Citation (APA)
Chen, Y. H., Lin, S. C., Huang, S. C., Lei, C. L., & Huang, C. Y. (2023). Guided Malware Sample Analysis Based on Graph Neural Networks. *IEEE Transactions on Information Forensics and Security*, *18*, 4128-4143. https://doi.org/10.1109/TIFS.2023.3283913

Guided Malware Sample Analysis Based on Graph Neural Networks

Yi-Hsien Chen, Si-Chen Lin, Szu-Chun Huang, Chin-Laung Lei, *Member, IEEE*, and Chun-Ying Huang, *Senior Member, IEEE*

***Abstract*— Malicious binaries have caused data and monetary loss to people, and these binaries keep evolving rapidly nowadays. With tons of new unknown attack binaries, one essential daily task for security analysts and researchers is to analyze and effectively identify malicious parts and report the critical behaviors within the binaries. While manual analysis is slow and ineffective, automated malware report generation is a long-term goal for malware analysts and researchers. This study moves one step toward the goal by identifying essential functions in malicious binaries to accelerate and even automate the analyzing process. We design and implement an expert system based on our proposed graph neural network called MalwareExpert. The system pinpoints the essential functions of an analyzed sample and visualizes the relationships between involved parts. We evaluate our proposed approach using executable binaries in the Windows operating system. The evaluation results show that our approach has a competitive detection performance (97.3% accuracy and 96.5% recall rate) compared to existing malware detection models. Moreover, it gives an intuitive and easy-to-understand explanation of the model predictions by visualizing and correlating essential functions. We compare the identified essential functions reported by our system against several expert-made malware analysis reports from multiple sources. Our qualitative and quantitative analyses show that the pinpointed functions indicate accurate directions. In the best case, the top 2% of functions reported from the system can cover all expert-annotated functions in three steps. We believe that the MalwareExpert system has shed light on automated program behavior analysis.**

***Index Terms*— Graph neural network, machine learning for security, malware analysis, reverse engineering.**

Manuscript received 30 October 2022; revised 11 April 2023 and 22 May 2023; accepted 22 May 2023. Date of publication 7 June 2023; date of current version 18 July 2023. This work was supported in part by the National Science and Technology Council (NSTC) of Taiwan under Grant 111-2628-E-A49-006-MY2, Grant 112-2634-F-A49-001-MBK, and Grant 112-2218-E-A49-023; and in part by the Taiwan Academic Cybersecurity Center (TACC) Project at the National Yang Ming Chiao Tung University. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Ghassan Karame. *(Corresponding author: Chun-Ying Huang.)*

Yi-Hsien Chen and Si-Chen Lin are with the Department of Electrical Engineering, National Taiwan University, Taipei 106319, Taiwan, and also with the Department of Computer Science, National Yang Ming Chiao Tung University, Hsinchu 300093, Taiwan.

Szu-Chun Huang is with the Department of Computer Science, National Yang Ming Chiao Tung University, Hsinchu 30010, Taiwan, and also with the Faculty of Technology, Policy and Management, Delft University of Technology, 2628 BX Delft, The Netherlands.

Chin-Laung Lei is with the Department of Electrical Engineering, National Taiwan University, Taipei 106319, Taiwan.

Chun-Ying Huang is with the Department of Computer Science, National Yang Ming Chiao Tung University, Hsinchu 300093, Taiwan (e-mail: chuang@cs.nycu.edu.tw).

Digital Object Identifier 10.1109/TIFS.2023.3283913

I. Introduction

MALICIOUS software is one of the most critical threats people and enterprises face nowadays. It has brought many data and monetary losses, and the even worse news is that new malware still grows much faster than humans can handle. The report shows that millions of new malware samples were observed monthly, accumulating billions of malware samples since 2013 [1]. Also, the number of observed ransomware attacks in the first half of 2021 has surpassed the total number of records observed in 2020 [2]. It is a foreseen future that cyberattacks will be more frequent and severe, and we need to figure out a more effective way to defend against them.

The vast number of malware samples has incurred much workload for security analysts. Although detecting malicious binaries is an age-old problem, malware detection is still challenging [3], [4], and it still requires a lot of human resources to analyze identified samples for further clarification. Even with the help of artificial intelligence models, there are still difficulties in various areas, including data labeling, feature selection, model selection, and evaluation. Suppose a sample can be accurately classified into either benign or malicious first. Analyzing a sample in-depth and understanding its behavior is still a time-consuming task. No matter how accurate a model is, a security analyst needs to know *why and how* the model makes a decision — especially when dealing with an unknown binary in real-world cases.

This study attempts to develop an expert system to analyze a malware sample and report essential functions identified in the sample. The system is composed of two components. One is to detect samples as malicious or benign accurately, and the other is to identify essential functions in the samples that lead to the detection. We design our approach based on static binary analysis techniques. To our knowledge, most existing machine learning-based detection approaches target improving detection performance. Although they can achieve high accuracy and low false-positive rates, it is challenging to explain *why* a model determines a malicious binary and *how* we can leverage the detection result to simplify the malicious sample analyzing process.

We tackle this problem by addressing the limitations and challenges observed in the current research works. The objective of our approach is to construct an *explainable* detection model. In addition to having a competitive detection accuracy, the retrieved binary semantic representations can be further used to explain the detection result and guide security analysts for better in-depth analysis. Our contribution is three-fold. First, we propose a graph neural network-based (GNN-based) malware detector which can achieve competitive performance in accuracy, precision, recall, and false-positive rates. Second, the GNN-based model produces explainable results by indicating the most critical subgraphs that lead to malicious detections. The identified subgraphs can be easily mapped to the evaluated samples' corresponding functions and function calls. Last, we further use our approach to analyze real-world samples and compare them against expert-made analysis reports to show the effectiveness of our approach. Specifically, we answer the following research questions to validate our research.

- **RQ1:** How many samples are required to train a good detector? (Section V-B)
- **RQ2:** Does our approach have a competitive performance to the state-of-the-art static-based detection approaches? (Section V-C)
- **RQ3:** Does our approach work well for unknown samples? (Section V-D)
- **RQ4:** What are the impacts of having function embeddings? (Section V-E)
- **RQ5:** What is the performance of the explainers? (Section V-F)
- **RQ6:** How to quantify the quality of clues provided by our proposed system? (Section V-G)
- **RQ7:** What behavior is explained from real-world samples using our proposed system? (Section V-H)
- **RQ8:** Does our approach work with customized packers? (Section V-I)
- **RQ9:** How the reverse engineering tools are selected for performance evaluation? (Section V-J).

The rest of this paper is organized as follows. We overview several research works that employ machine learning and neural network approaches to detect malware in Section II. In Section III, we review past research on Graph Neural Networks, feature representation, and graph-related model explanation, which are utilized in our proposed methods. We introduce our approach in Section IV and evaluate its performance in Section V. Finally, a concluding remark and future research directions are given in Section VI.

II. Related Work

Many research works have employed machine learning techniques for malware detection and classification. These works generally take static, dynamic, or raw binary features as input and output binary (e.g., benign or malicious) or multi-class (e.g., malware families) results. This section discusses machine learning-based malware detection research works using static analysis features. We classify these works into three categories: 1) using typical file content-based features, 2) employing modern embedding-based features, and 3) making explainable predictions.

A. File Content-Based Feature

Binary analysis techniques can be used to extract program information and runtime behavior from binary executable files. Before employing machine learning-based approaches, extracted program information and runtime behaviors must be transformed into features represented in fixed-length vectors. This section introduces several handcrafted features and how these features are used in machine learning models.

Raff et al. [5] propose two approaches to detect malware based on features extracted from the executable file header information. Their work focuses on Windows portable executable (PE) files. In the first approach, they parse the PE header information into feature vectors based on experts' domain knowledge and build a random forest model to perform the detection. In the second approach, PE header bytes are transformed into features using the N-gram algorithm instead of relying on domain knowledge. A long short-term memory (LSTM) model is then used to build a detection model. The results show that the two models are both effective for prediction. More in-depth experiments [6], [7] are also conducted to understand better what the model learned from N-gram features. They conclude that the model learned entropy and string features from N-gram features.

Researchers [8], [9] also attempt to convert an executable binary into a corresponding gray-scale image and use the images to train a CNN-based malware detector. The detection performance looks good on malware classification compared to models built from other features. Vasan et al. [10] further use image-based CNN models to create an ensemble architecture to improve the overall detection performance. Le et al. [11] propose deep learning-based malware classification without feature engineering. They feed raw binary byte sequences into a convolutional and recurrent neural network to classify malware families. Training a model with either gray images or raw binary byte sequences looks interesting and compelling. No domain knowledge is required in the preprocessing and training process. However, the rationale behind the models could be not intuitive and difficult to explain.

Anderson and Roth [12] propose the EMBER dataset, which defines several feature extraction methods for static malware analysis models, including header/section information, printable strings, entropy, import/export functions, and the N-gram of the most frequently used instructions. They further leverage a gradient-boosted decision tree model (LightGBM) to build a malware detection model, showing that it outperforms several featureless deep learning models. Pham et al. [13] use a gradient-enhanced decision tree algorithm to detect malware with a high detection rate and low false alarm rate. Also, they perform feature reduction on the features of the EMBER dataset to reduce the training time. The results show that the feature extraction from PE files using static analysis and domain knowledge would improve the performance.

B. Embedding-Based Feature

Working with typical file content-based features is straightforward. However, it does not provide sufficient insights for analysts to understand the criteria for detection and the semantics behind the models. One step toward understanding how a model works is using embeddings as features, which are frequently used in natural language processing. There are several benefits to using embeddings as features. Embeddings can be used to preserve contextual information. They also preserve geometric properties so typical distance measurement matrixes can measure the similarity between embeddings.

Many research works have proposed approaches to generate embeddings from sequential or structured data [14], [15], [16], [17], [18], [19], [20]. There are also embeddings explicitly designed for source codes and program binaries, such as FCG2vec [21], code2vec [22], and SAFE [23]. With embeddings, security researchers further build machine learning models to detect and classify malicious samples. FireEye proposes a model [24] that detects malware from raw byte sequences. It generates embeddings and feeds them into a one-dimensional convolutional layer and temporal max-pooling layers. Then, they use fully connected layers and a softmax layer to determine whether a sample is malicious. Hashemi et al. [25] propose a new approach for unknown malware detection using opcode graph embedding. They translate each opcode into a vertex. The weight of each directed edge represents the transition probability from one opcode to another. They use Power Iteration [26] to generate graph embedding for detecting malware. Hassen et al. [21] propose a function-call-graph representation vector for malware classification, and it could combine graph and non-graph features within the representation vector. Broder [27] calculate graph similarity using the MinHash signature of functions and classify the malware groups. Gascon et al. [28] efficiently condense structural information of function call graphs by calculating the neighborhood hash value for each node and evaluating the graph kernel in linear time for each call graph. They generate feature space embeddings fed into a linear SVM for training and detecting malicious functionality in samples and also attempt to perform interpretation of the model decisions. Xu et al. [29] compute the control flow graph embedding for each binary function and calculate the distance between embeddings to decide the similarity of functions. They achieve faster embedding time and better vulnerable detection ability than prior works. Fan et al. [30] depict the complex relationships among various entities (i.e., file, archive, machine, API, DLL) in PE files by constructing heterogeneous information networks (HIN). Based on different views of the HIN relations, meta-graphs representing the relatedness among files are generated and embedded into a lower-dimension vector through metagraph2vec. Moreover, embeddings are combined via a fusion algorithm and put into a Support Vector Machine (SVM) for further malware classification. Zhang et al. [31] extract behavior information of programs into word sequences. A semantic-captured word embedding model (GloVe) vectorized each word and formed a feature map for each file. Furthermore, a convolutional neural network (CNN) utilizes these vector maps for analyzing benign and malicious binaries.

C. Explainable Prediction

Well-tuned neural network models can give us high accuracy rates and good performance. Sometimes, we only need a little insight and domain knowledge of malware analysis. However, such a black box could bring uncertainties when applied to real-world cases. From developers' perspectives, they have to figure out how a model works and the critical points of a sample to make sure everything makes sense, not just find particular magic rules in the training dataset. We also expect a model to learn interesting characteristics analysts may not notice from samples. An explainable model lets developers and users realize how a prediction is made and thus is essential for building a trustworthy and accurate model.

Krčál et al. [32] investigate and evaluate the malware classification model learned from raw sequences of bytes and labels to figure out what it learned. They find that the model would take the header's context, sections of various types, resources, or relocation tables to classify the malware. Coull et al. [33] evaluate the raw-bytes model proposed by FireEye [24] with activation analysis. They conclude that import-related features occupy an essential place, and the model could identify ASCII strings and specific behaviors, such as calling functions from the bytes sequence. Moreover, it learns complicated features like checksum and Rich header information [34]. Demetrio et al. [35] use adversarial attack techniques to find out the critical points of MalConv. They assume that the DOS header, which is useless in modern PE binaries, could play an essential role in the decision. Bose et al. [36] evaluate MalConv and make a different conclusion from Demetrio's work. They find the header information important, while other binary pieces could also be accountable for the final result. Not only analyzing what a model learned, but some works also focus on building an explainable model. Korine et al. propose DAEMON [37], which generates a malware classification model through multi-stage feature mining, including entropy threshold computation, family representative N-grams extraction, and pairwise-separating feature selection. After these stages, a random forest model with high accuracy and explainability would be generated based on the selected features.

In summary, researchers have developed multiple ways to represent the characteristics of a binary, from handcrafted features to context-preserving embeddings, to boost the detection performance of the classifiers. Generally, neural network embedding-based approaches achieve better performance than classical machine learning-based approaches. Nevertheless, methods based on classical machine learning approaches are more likely to be used in the real world than neural network models because models based on statistics or mathematics would be easier to understand. Some researchers have attempted to inspect the content learned by the neural network models or even tried to build a human-understandable model. However, the answers to explaining a model's predictions or knowledge captured by models still need to be clarified. On the other hand, we can easily apply a classifier to predict the labels if the embeddings we used are distinguishable. However, even if using some attention mechanism to show the highly focused parts of the embedding, we still cannot understand the relation between embeddings and prediction labels.

In this study, we proposed a new direction for building a behavior-explainable malware classification model leveraging multiple types of program embeddings. We generate contextual embeddings from functions and function call relationships within a program and use the embeddings to train graph neural networks to perform malicious software detection. Using various approaches, we then identify critical structures in the program and explain its behaviors recognized by our model. By pointing out crucial components inside the program, we aim to reduce the security analyst's efforts when analyzing an unknown malicious sample.

III. Background

A. Representation Vector and Embedding

While developers or attackers generate binary files using a compiler, security analysts usually have to reverse the process by reconstructing assembly codes from machine codes. Furthermore, reconstructed assembly codes can be used to build control flow graphs, function call graphs, and even an abstract syntax tree for more in-depth analysis.

However, employing machine learning approaches for program analysis requires one further step because most machine learning models only handle inputs in the form of representation vectors. A representation vector (also called an embedding vector or simply embedding) is a form that can be used to represent complex or high-dimension inputs in a simplified form. It is usually in the form of low-dimension vectors. Ideally, an embedding can capture some of the input semantics and be used for training a machine learning model. Therefore, before feeding program samples to a model, it is essential to have a good approach for transforming reconstructed assembly codes into adequate embeddings.

Several works have been devoted to developing methods for obtaining embeddings from complex structures. Dai et al. proposed structure2vec [19], which generates embeddings of structure data with discriminative information. It extracts features by graph inference, which is similar to mean-field and belief propagation. Grohe et al. investigate the vector embedding theory [20] for structured data. In general, embeddings can be retrieved by using various approaches that can handle the input structures. Examples include DeepWalk [15], node2vec [16], the Weisfeiler-Leman algorithm [17], and Homomorphism vectors [18]. Creating embeddings using a graph neural network is one of the most generic and straightforward approaches for graph-based input structures.

There are also embedding approaches explicitly developed for program analysis. Program structures required for creating embeddings can be obtained from static analysis or dynamic analysis results. Choosing between static analysis or dynamic analysis approaches is a trade-off between program analysis time and the realness of the obtained information. Working with a static analysis approach would be more efficient, and it can analyze a program thoroughly. However, it may not be able to get the most precise program information if a processed program is protected. Several approaches can obtain embeddings from program analysis results. Asm2vec [38] attempts to generate embeddings from low-level assembly codes and basic blocks. It leverages natural language processing techniques like word2vec [14] to handle each assembly instruction as a word and pieces of assembly codes as a sentence to learn the embeddings. Moreover, structured data such as function call graphs and abstract syntax trees contain the relationships between components in a program. To get more semantic information, we can take instructions to construct a control flow graph and generate function embeddings by using algorithms such as FCG2vec [21], code2vec [22], and SAFE [23].

B. Graph Neural Network and Model Explanation

A graph neural network (GNN) is a neural network that processes graph structure data. While neural networks process inputs in the form of vectors and matrices, the most important part of a graph neural network is obtaining the most appropriate form to represent a graph in vectors, often called embeddings. Most models deal with graph structure data based on message-passing schemes, which are considered the generalization versions of the convolution operator. Using the message passing scheme, we can pass messages such as some features or hidden vectors to our neighbors and aggregate the information of the messages to produce embeddings. Graph Convolutional Network [39] (GCN) applies convolutions on graph structure data and encodes both local graph structures and features of nodes to generate embedding. Graph Attention Network [40] (GAT) leverages masked self-attentional layers that let the model knows the importance and focuses on the relevant features of neighbor nodes to learn the representation vector. Gated Graph Sequence Neural Network [41] (GGNN) uses gated recurrent units (GRUs) to memorize the sequential relationship of the node features at each update and generate graph semantic vectors.

In recent years, extensive research works have applied GNN techniques to antagonize various malicious activities. Researchers [42], [43], [44] have explored different graph extraction and context embedding techniques to classify PE malware with GNN models. From extracting control flow graph or function call graph to retain the structural information of a program, using multiple semantic embedding methods, such as word2vec, to capture semantic features, to applying GNN model structures, like Graph Isomorphism Network (GIN) or Graph Attention Network (GAT), for generating graph embedding from a program, they aim to provide input with as much information as possible to their final classifiers. Zhang et al. [45] propose adversarial attacks to evade the detections from GNN malware detection models. Interestingly, many GNN-based detectors are designed specifically for Android binaries [46], [47], [48]. The Android decompilers can precisely decompile bytecodes into (minified) source codes.

The nature of graph-based models is that they are easier to be explained. There are several ways to explain a graph model's prediction. A straightforward way is to mutate the input and calculate the loss gradient to see what graph structure or features affect it. Furthermore, if a model leverages the attention mechanism to generate embeddings, it can calculate attention scores and see the highest attention part of the graph and features. Ying et al. [49] propose GNN explainer, a model-agnostic approach, to identify impactful subgraphs and node features that influence the model prediction. The explainer focuses on maximizing mutual information between the GNN's prediction probability and the prediction distribution of possible crucial subgraphs. It supports explanations for various machine learning tasks, including node classification, graph classification, and link prediction.

Researchers also leverage GNN-based detection models to recognize relevant activities in malicious samples. He et al. [48] propose MsDroid, which extracts sub-graphs from a program based on the neighboring nodes of selected sensitive APIs. They then train a GNN-based detection model to classify whether a sub-graph (code snippets) is malicious or benign. Besides showing sensitive APIs correlated with malicious behaviors in the input call graph snippets and retrieving similarly implemented snippets from known malware, they try to maximize the mutual information between prediction and distribution of crucial edge dependencies, as GNN explainer, to identify the importance of edges in API call graphs.

Our approach differs from previous works as follows. First, we do not depend on any predefined API list, allowing for greater flexibility and adaptability to changing threat landscapes. Second, we consider the whole graph structure instead of API-segmented sub-graphs, allowing for a more comprehensive understanding of its behavior. Last, we examine the influence of various embedding methods on classification performance, explore different explaining possibilities, and further validate the effectiveness of our explanations in improving the malware analysis process. Readers can refer to Section V for more details.

IV. Approach

A. Problem Statement

The proposed expert system aims to build an explainable malware classification model that detects malicious samples and explains further the critical part that leads to the detection. We give a formal statement of the problem, clarify the scope of this study, and list assumptions of the proposed approach in this section. Suppose we have a set of binary programs X containing $|X|$ programs. Each program $x_i \in X$ $(1 \leq i \leq |X|)$ is analyzed first to generate its call graph g_{x_i}. Suppose there are n_{x_i} functions in x_i. The call graph g_{x_i} is built based on each recognized function $f^j_{x_i}$ $(1 \leq j \leq n_{x_i})$ in x_i and their calling relationships. The graph g_{x_i} can be transformed to its corresponding embedding $\vec{g_{x_i}}$, where each node $f^j_{x_i}$ in the graph can be represented as either a null embedding, e.g., $[0 \ldots 0]$, or a preferred function embedding $\vec{f^j_{x_i}}$. By collecting the graph embeddings of all programs in X and form G_X, our system trains the graph neural network model Φ_X to perform the recognition of malicious samples. Given an arbitrary program sample x' (known or unknown), our system first detects x' as either benign or malicious. Furthermore, a model explanation is performed by feeding $g_{x'}$ and Φ_X to our proposed model explainer to identify essential parts in program x'. Table I summarizes the notations used in this study.

TABLE I

Notations Used in This Study

Notation	Meanings
X	A set of samples used to train the detection model.
x_i	A sample program in X, where $1 \leq i \leq \lvert X \rvert$.
n_{x_i}	Number of functions in program x_i.
$f^j_{x_i}$	A function in x_i, where $1 \leq j \leq n_{x_i}$.
g_{x_i}	The call graph constructed based on functions in x_i.
$\vec{f^j_{x_i}}$	The function embedding generated for function $f^j_{x_i}$.
$\vec{g_{x_i}}$	The graph embedding generated for program x_i.
G_X	The set of graph embeddings for all programs in X.
Φ_X	The graph neural network model trained from G_X.
x'	An arbitrary program for detection and explanation.

The usage scenario of our proposed approach is as follows. A security analyst may use a pre-trained model from others to perform guided sample analysis. However, to build a self-trained model, a security analyst collects a bunch of samples from known sources and has total control of these samples. To capture the essence of binaries, the analyst may conduct preprocessing procedures for these samples, such as removing packer-related parts, to avoid noise information. Based on this scenario, this study follows three assumptions.

- **Most samples are deobfuscated or unpacked.** Malicious programs often adopt polymorphic or metamorphic techniques to evade the malware detection systems using packer or obfuscated code. Several studies [50], [51], [52], [53] have discussed packer detection, unpacking, and deobfuscation and aimed to solve these problems systematically. Because training a model for recognizing crucial behaviors in a malicious program interests us the most, we ignore samples packed in well-known packers or obfuscators. There still exist some programs packed by customized or unknown packers or obfuscators in our dataset. Nevertheless, our experiment results show that our system can capture these routines when handling packed or obfuscated samples since identifying function parts that perform unpacking and deobfuscating is also a crucial process in reverse engineering.
- **Use only static analysis features.** We consider only features that can be retrieved from static analysis for the following reasons. First, the cost of retrieving static analysis is manageable. We do not have to set up a runtime environment, and the analysis time is usually proportional to the size and complexity of a sample program. Second, static features are deterministic. It is not affected by runtime conditions and exceptions. The results depend only on the algorithm we used to extract features. Third, static features are scalable. Due to its simplicity and deterministic property, it is also easier to scale out and scale up the feature extraction process to accelerate the analyzing process.
- **Do not consider adversarial samples.** As using GNN-based models on malware detection become more popular, researchers have explored the possibilities of adversarial attacks [45] on these detection models. However, we ignore adversarial attacks in this work because we focus more on training a model that can recognize interesting parts in a (malicious) program, not proposing a strong detector. In our proposed scenario, a user can choose to filter out possibly adversarial samples to prevent his model from being polluted. Adversarial samples mainly mislead a detector into classifying the samples into benign samples. In order to mitigate the possible impacts caused by adversarial samples, benign samples can be collected only from trusted sources. For malicious samples, we can only use samples that are considered malicious by most of the detectors.

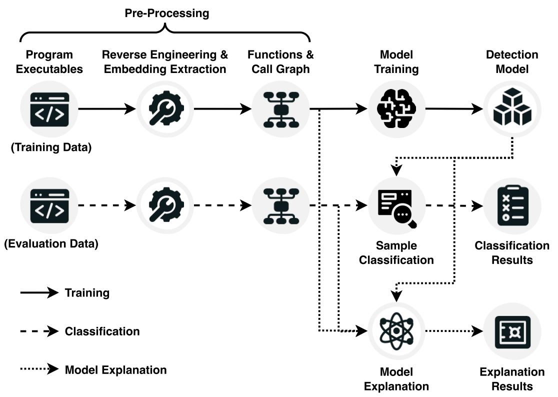


Fig. 1. The workflow of the proposed MalwareExpert system.

B. Workflow Overview

The workflow of our proposed detection and explanation system is depicted in Figure 1. There are three phases: the sample preprocessing phase, the training/classification phase, and the explanation phase. The sample preprocessing is mandatory for the rest of the phases. In the sample processing phase, each program executable is reverse-engineered and statically analyzed for identifying functions and caller-callee relationships in the executable. A corresponding call graph is then built based on the analyzed information. Note that we need additional models to perform embedding extractions, which are used in the preprocessing phase in Figure 1. The models can be publicly available pre-trained or self-trained models. The selected embeddings are introduced later in Section IV-C

The training and classification phase is similar to typical machine learning applications. They share the same path in the workflow. With a well-trained model, an unknown program executable can follow the same procedures used in the training phase to obtain the embeddings of the program and then feed the embeddings to perform the classification.

The most challenging phase in our workflow is the explanation phase. This phase aims to report critical parts identified in a classified program executable. The program can be either a known (available in the training set) or an unknown sample. Since the input to the classification model contains function embeddings and graph structure, the reported critical parts that impact a sample as malicious or benign can be shown as functions and function calls. We expect those reported functions and function calls to be clues for security analysts to analyze the input program's behaviors better. The details of the involved phases are discussed in Sections IV-C, IV-D, and IV-E, respectively.

C. The Sample Preprocessing Phase

Procedures involved in the sample preprocessing phase include reverse engineering, static analysis, and embedding extraction. The preprocessing phase is required in both the training phase and the detection phase. There are a lot of reverse engineering tools and static analysis tools available, including IDA Pro [54], Ghidra [55], Radare2 [56], and many other alternatives. Our study uses the open-sourced Radare2 as the default reverse engineering tool. It is used to identify functions and build the corresponding function call graph. A function call graph is a directed graph containing nodes and edges, where each node represents a function, and each directed edge represents a function call relationship. If function f_a (caller) calls function f_b (callee), two nodes, a and b, are added to the graph with a directed edge e_{ab} linked from node A to node B.

Embedding for each function is extracted and placed in the generated call graph, serving as the node feature. Although node information, i.e., function embeddings in this study, is not a must in graph-based classification, working with embeddings would be better for the classification. The structure of call graphs is relatively monotonic compared to other graph-based applications such as social networks. Therefore, considering both the node information and the graph structure simultaneously to obtain the graph embeddings can enrich the information in the output. To validate our assumption, we consider three different function embeddings: the null embedding, the Asm2Vec [38] embedding, and the self-attentive function embeddings (SAFE) [23]. We use a vector containing only zero values as the null embedding for each node. The sizes of SAFE and Asm2Vec embeddings follow the settings recommended by the authors, which are 100 and 200 dimensions, respectively. The size of the null embedding follows the smaller one, which is 100 dimensions.

The Asm2Vec model handles assembly codes of a function as a document and learns the representation vector based on the operand and operator tokens inside a function. Asm2Vec updates the prediction of current instruction each time based on the predicted function representation at the moment and the neighboring context to generate the final function representation. With only assembly codes, Asm2Vec can capture semantic relationships between tokens in the assembly code. The architecture of the SAFE model is depicted in Figure 3. The model generates a function embedding from a disassembled function. Similar to word2vec [20], it first uses an instruction-to-vector (i2v) block to convert each instruction to a corresponding vector. It then leverages a self-attentive network [57] to generate function embeddings from the instruction vectors. The SAFE model should be trained with sufficiently diverse programs to ensure that it can produce different embeddings for various executables.

D. The Training/Classification Phase

Once the graph embeddings are ready, the next step is to train a model that classifies a program as malicious or benign. We propose a MalwareExpert model to perform graph-based malware classification, as shown in Figure 2. The model is designed by extending the graph convolutional network [39] (GCN). The model takes the call graph and function embeddings as input and generates the binary semantic embedding for prediction based on the graph structure. While typical graph neural networks have few layers (usually 2–4), it is still challenging to effectively handle a massive number of inputs retrieved from a program. Three typical issues are over-fitting, over-squashing [58], and over-smoothing [59], [60] issues. Although adding dropout layers can mitigate the problem of over-fitting, the performance in terms of accuracy could

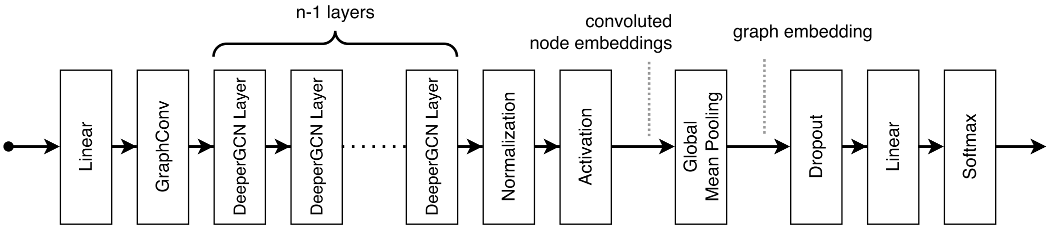
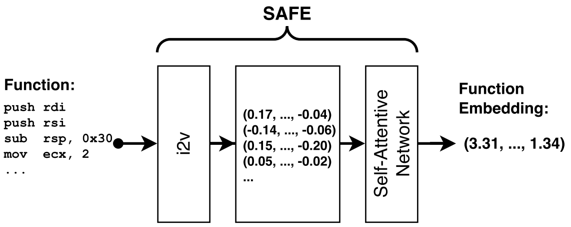


Fig. 2. The architecture of the MalwareExpert model.



```
push rdi
push rsi
sub  rsp, 0x30
mov  ecx, 2
...
```

Fig. 3. The architecture of the SAFE model.

be better due to limited model capacity, which leads to over-squashing and over-smoothing. Therefore, we integrate the DeeperGCN [59] concept in our design to further solve the over-squashing problem by increasing the number of layers in the network.

Our MalwareExpert model first takes the node feature vectors as input and uses a Linear layer to embed them into hidden vectors to ensure the inputs can fit the memory limitation of the hardware. For performing exact n times of graph convolutions, the hidden vectors and the corresponding edge relations are fed to a GraphConv layer and the following $n - 1$ DeeperGCN layers. The output is the convoluted node embeddings. After passing through the normalization and the activation layer, the convoluted node embeddings are forwarded to the global mean polling layer to produce the graph embedding for each sample in the batch. Our current model design predicts scores for the corresponding classes (benign and malicious) instead of predicting a single malicious probability. Passing the graph embedding to the following dropout, linear, and softmax layers, the model finally makes the prediction based on the class having the highest score. The primary benefit of having prediction scores for each class is that we do not need to define a threshold for malicious probability, which is not a trivial task.

Existing research works mitigate the over-smoothing problem mainly by designing specific GCN-based models or adjusting the convolution process and normalization function to preserve the diversity of features [59], [61]. Figure 4 shows the architecture of each DeeperGCN layer. It uses the skip connection operation, the pre-activation residual connection, to keep meaningful information not fading out and prevent gradient vanishing. The implementation setup of our MalwareExpert model is as follows. We set n to 8, use GENeralized Graph Convolution [59] to implement GraphConv, use LayerNorm [62] to perform normalization, and use the ReLu function for activation.

The process of the classification phase has the same preprocessing steps as the training phase. Instead of training a detection model, the classification phase uses the trained model to perform sample classification. With only the trained model and the input data required for the explanation phase, which aims to explain the reason behind the prediction result, the stages of classification and explanation are essentially independent.

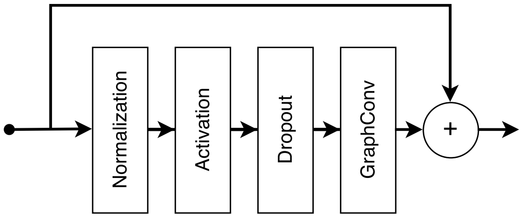


Fig. 4. The architecture of each DeeperGCN layer.

E. The Explanation Phase

When a malicious sample is reported, a security analyst needs to understand why the sample is classified as malicious and what is the underlying behavior behind the classification. We believe that if a graph-based model can report which part of the graph leads to the detection, it would benefit security analysts and accelerate analyzing malicious programs.

We propose two approaches to identify the graph's critical parts that lead to a corresponding detection. A straightforward approach is designed based on graph pruning, and the other leverages the GNN explainer [49]. The rationale behind the graph pruning approach is straightforward. When the graph-based model could not recognize the critical structure pruned from a graph, it would decrease the prediction score of the involved class. Therefore, we use two pruning strategies for edges derived from a target sample's function call graph. One is edge-pruning, which removes only a selected edge from a graph. The other is node-pruning, which removes an edge's two endpoint nodes and edges connecting to the two nodes from a graph. We apply the selected pruning strategy for each edge from the original graph, perform the detection against the pruned graph using our proposed model, and measure the predicted benign score for the pruned edge. The impact of pruning a single edge is determined by measuring the benign score difference between the pruned and the original graph. After measuring the impacts for an edge, we roll back the graph to its original state. The rank of each edge is sorted based on their impact on the prediction.

Alternatively, the GNN explainer attempts to identify critical subgraph structures that impact the GNN model's decision. It can apply to various GNN architectures, including Graph Networks, Graph Convolutional Networks, and many other GNN-based models.

Each GNN prediction is evaluated based on a computation subgraph containing the critical information for a sample's classification result. The hidden computation subgraph must have enough GNN's neural messages to flow through and arrive at the related nodes to generate a proper prediction. Thus, the GNN explainer trains another neural network to maximize the mutual information between approximate computation subgraphs and the model prediction, which aims to find the core graph structure that affects the model's behavior.

The output model from the GNN explainer takes the trained GNN model and its prediction as input and assigns weight to each edge in the corresponding input graph as the edge's importance score. We can use these GNN model explanations as clues to examine the caller-callee relationships in the original function call graph and infer which subgraph parts are crucial to this sample. Based on these clues, experts can then mine the call chain of these crucial functions.

V. Evaluation

This section answers the following research questions (RQs) by designing experiments to evaluate our proposed approach and discuss several interesting findings.

We implement our proposed approach using PyTorch and run the codes on a server with an NVIDIA Quadro RTX 6000 GPU. We follow the TESSERACT [63] practice to perform the evaluations to reduce the potential bias in malware detection experiments. The TESSERACT practice suggests that the spatial and temporal distribution of samples selected for evaluations should be well-controlled. In the case of spatial distributions, the proportion of benign and malicious samples should be fair in most test cases. Furthermore, the number of programs developed in different periods should be balanced when dealing with temporal distributions. Otherwise, it may mislead a model to classify samples based on timing-relevant features instead of the nature of the samples. In short, the samples used for evaluations should be sufficiently diverse, or it could lead to biased models and results.

A. Dataset

We perform the evaluations with three different datasets: benign samples, viruses, and APT samples. While Windows is still the primary operating system that dominates the desktop OS market [64], we mainly use samples available on the Windows platform in our evaluations.

The datasets are collected from various sources. For benign samples, we collect 44,953 benign samples from Windows 10 system binaries and libraries, popular packages from the Chocolatey[1] repository, and the Cygwin[2] utilities. For viruses, we collect 75,257 portable executable (PE) files from the VirusShare[3] dataset. For APT samples, we collect them from publicly available sources[4] and use 2,865 real-world samples from 12 state-sponsored APT groups. All of them are also executables in the PE format.

TABLE II

Summary of the Datasets Collected in This Study

Dataset	Arch.	Packed by Well-known Packers	The Rest	Total
Benign	32-bit	7	30,793	30,800
	64-bit	1	14,152	14,153
Virus	32-bit	5,386	64,072	69,458
	64-bit	517	5,282	5,799
APT	32-bit	94	2,656	2,750
	64-bit	0	115	115

Table II shows the summary of the datasets collected in this study. Before we use the samples to evaluate our proposed model, we perform simple preprocessing against the samples and drop samples that could mislead our model. We first drop all the binaries packed by well-known packers to reduce the possible dataset pollution from well-known packers, as suggested by Aghakhani et al. [65]. The identified packers from the virus dataset are shown in Figure 6. The detection is performed by Detect-It-Easy [66] tool. Although we do not train our model with binaries packed with well-known packers, our virus dataset contains more than 100 unique types of customized packed samples reported from virus scanners. We also notice that the numbers of 32-bit binaries are generally more than that of 64-bit binaries. When sampling binaries from the datasets, we maintain the ratio of samples selected from the pools equivalent to the ratio we observed in the field.

We plot the number of nodes and edges retrieved from samples of each dataset in Figure 5. Each point in the figures indicates the number of nodes and edges obtained from a sample. The figures show some interesting observations in the samples. First, we can see that some samples are plotted on the bottom line of the figures. These samples are exceptional because they are composed of only nodes (functions) without edges (function calls inside the sample). We remove them from our datasets because no call graph can be constructed from these exceptional samples. After the removal, the benign, virus, and APT datasets left 16,847, 55,886, and 2,665 samples, respectively. Note that many files were removed from benign samples because they were library files. Second, we observe that some samples have a vast number of nodes. These samples are large-scale applications such as browsers and office software components. It is also interesting that some malicious samples also have many nodes. Most cases are because a malicious code snippet is injected into a benign target with many nodes.

B. RQ1: Is the Model Well-Trained?

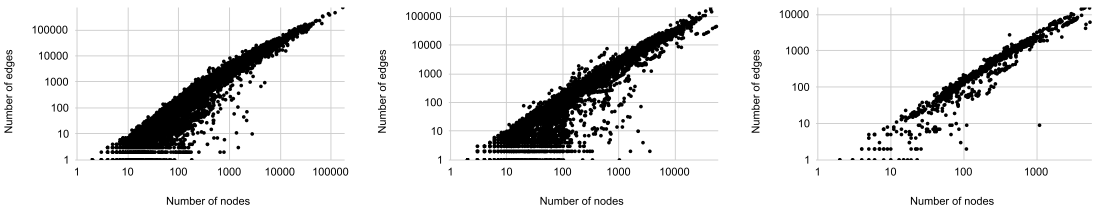
This section evaluates how many samples are required to train a good detector and what is the impact of multiple GCN layers. We control the number of samples used to train the model for training time and space considerations. First, extracting the call graphs and embeddings from the sample executables costs a lot of time. Second, the obtained call graphs and the function embeddings from the samples also consume many spaces. For example, given 30,000 samples from our dataset, the total size of the raw features in JSON format is about 40GB. Finally, the time required for training a model is highly relevant to the number of samples fed to a model. While the performance of the model and the number

[1] https://chocolatey.org/

[2] https://www.cygwin.com/

[3] https://virusshare.com/

[4] https://github.com/cyber-research/APTMalware



(a) Samples in benign dataset.

(b) Samples in virus dataset.

(c) Samples in APT dataset.

Fig. 5. The number of nodes and edges observed in the collected datasets.

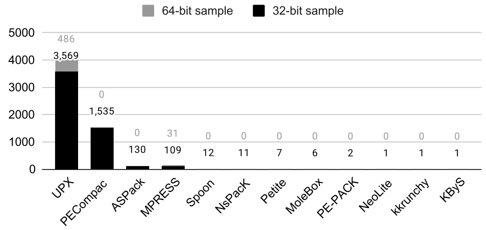


Fig. 6. Identified packers from the virus dataset.

TABLE III

EVALUATION WITH DIFFERENT NUMBERS OF SAMPLES

# of Benign	# of Virus	Accuracy	Precision	Recall	F1-score
1,500	1,500	0.885481	0.893238	0.906137	0.899642
4,500	4,500	0.926988	0.941606	0.923628	0.932530
7,500	7,500	0.962000	0.970297	0.950139	0.960112
12,000	12,000	0.965333	0.959766	0.972387	0.966035
15,000	15,000	0.970000	0.964912	0.976331	0.970588

of used samples is often a trade-off, we have to ensure that we use a sufficient number of samples in the evaluations to have good quality evaluations.

To answer this research question, we use different numbers of samples ranging from 3,000 to 30,000 to train our proposed model and evaluate its performance. Note that the ratio between benign and virus samples is 1:1 in all settings. The evaluation settings and the results are presented in Table III. To determine if the performance change is caused by a change in quantity and not by some of the newly added samples, we perform experiments ten times to obtain the average results. In each experiment, we randomly select samples from a fixed dataset, train the model, and conduct the experiments multiple times to eliminate distribution bias.

The results show that more training samples would improve detection performance. However, the improvement increases more slowly. It becomes relatively stable when more than 15,000 samples are in the training dataset. We use a total of 30,000 samples from our benign and virus datasets, if not otherwise mentioned, based on the evaluation results.

We further validate the GNN's sample classification capability by visualizing the graph embeddings produced in our model. We train the model with 30,000 samples from the benign and virus datasets and plot the graph embeddings using the t-SNE [67] dimension reduction approach, which projects high-dimension embeddings to two-dimension spaces. The results are depicted in Figure 7. The figure shows that the malicious samples can be separated from benign ones in the projected space in most cases. It also concludes that the model is well-trained to perform the detection.

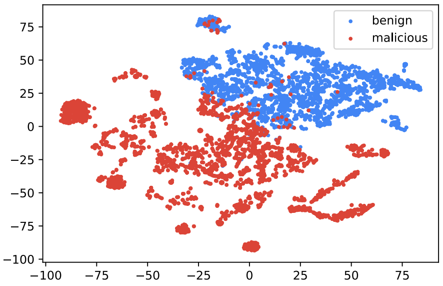
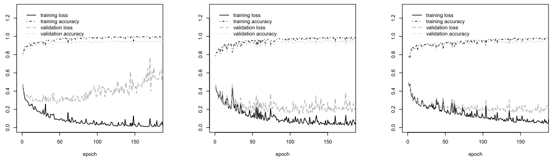


Fig. 7. Graph embeddings of samples in the benign and the virus datasets produced by our proposed model.



(a) Layers = 1

(b) Layers = 4

(c) Layers = 8

Fig. 8. Performance results for different layers of GCN implemented in our proposed approach.

TABLE IV

EVALUATION RESULTS FOR THE BENIGN AND VIRUS DATASETS

Model	Accuracy	Precision	Recall	F1-score
EMBER (LightGBM)	0.998166	0.998332	0.998000	0.998166
MalConv (CNN-based)	0.944500	0.941118	0.948333	0.944711
MalwareExpert (GNN-based)	0.973787	0.981752	0.965715	0.973632

Readers may notice that a few red and blue points are mixed in Figure 7. It does not indicate that our proposed model cannot distinguish them well. While the t-SNE algorithm reduces high-dimension points to a 2D plane, it is unavoidable that separable points in the original space could collide in the 2D space. Therefore, the linear layer is employed in our model to help find the correct boundary to classify high-dimensional data.

To verify that our proposed approach does not have over-squashing and over-smoothing issues, we conduct performance evaluations using different layers of GCNs and observe whether the performance is stable. In the experiments, we set the number of GCN layers to 1, 4, and 8 and plot the measured loss and accuracy for the training and validation phases in Figure 8. The results show that the losses and accuracy numbers get stabler and better as the number of layers increases.

C. RQ2: Detection Performance

This section evaluates whether our proposed approach has a competitive performance to the state-of-the-art static malware detection approach. We compare our proposed approach against two different flavors of detection approaches. One is based on typical machine learning algorithms, and the other is based on neural networks. For the case of typical machine learning algorithms, we use the LightGBM model to evaluate the EMBER dataset [12] but train it with our datasets. For the case of neural networks, we consider the MalConv model [68], which constructed a neural network model trained by raw bytes from samples. The MalConv model is trained using the same datasets. Because the two models output a probability of being malicious, we use a threshold of 0.5 as the decision boundary for the models to predict benign or malicious.

We train the two selected approaches and our proposed approach using samples half from the benign dataset and half from the virus dataset. The model is evaluated with 30,000 samples splitting into a ratio of 8:2 for training and validation. The evaluation result is presented in Table IV. Our proposed GNN-based model has a competitive performance close to the LightGBM model and outperforms the MalConv model. Furthermore, our proposed approach is much better than the LightGBM model in explainability. The LightGBM model is a tree-based model containing decision boundaries for selected feature vectors. Although a user can inspect and verify how a decision is made, two challenges make it difficult to explain the classification. First, the meanings of these decision boundaries could be magic numbers to a user. Second, a decision often depends on several diverse inputs before making it. However, the involved inputs may not have any relationships, making it difficult to explain. Our approach tackles these challenges by identifying the critical structures in call graphs, which have direct mappings to functions and function calls. More details on the explainability are further discussed in Section V-G.

D. RQ3: Handling Unknown Samples

We further evaluate whether the proposed approach can detect unknown types of samples that the model has never seen. We train our model and then compare models with only benign and virus dataset samples to perform the evaluation. In addition to training all three approaches used in RQ2 with our evaluation dataset, we also use the pre-trained models available to the public to perform the evaluations. We perform the detection against samples from the APT dataset and the same amount of model-unknown samples from the benign dataset, using the self-trained and the pre-trained models for the evaluation.

The specifications of the pre-trained models are as follows. The LightGBM model is pre-trained with the EMBER dataset containing 1.1 million binaries. The features include PE structure, byte entropy, strings, and many static analysis features. The pre-trained LightGBM model is publicly available under the GitHub repository of the Elastic project [69]. Researchers [12] also provide the pre-trained MalConv model to compare the performance of the LightGBM and MalConv models. The pre-trained MalConv model is also available under the GitHub repository of the Elastic project [69]. Note that we can not train our approach with EMBER dataset because the dataset does not contain the original executables to produce the features required by our model.

TABLE V
EVALUATION RESULTS FOR THE APT DATASET AND UNKNOWN BENIGN SAMPLES

Model	Approach	Accuracy	Recall
LightGBM	EMBER (pre-trained)	0.819382	0.639043
	EMBER (self-trained)	0.938820	0.879310
CNN-based	MalConv (pre-trained)	0.746384	0.493604
	MalConv (self-trained)	0.883870	0.843159
GNN-based	MalwareExpert	0.965021	0.950094

The evaluation result is presented in Table V. We observe that the models trained with our prepared datasets (labeled as self-trained) have better detection performance than the pre-trained models in the LightGBM and MalConv models. The result shows that all the self-trained LightGBM, MalConv, and our GNN-based model can effectively recognize unknown samples. Nevertheless, our model outperforms the other two, indicating that, with the same training dataset, our model has better detection ability when facing unknown samples.

E. RQ4: Impacts of Function Embeddings

This section discusses the impacts of having function embeddings in our design. We believe that simultaneously considering both the node information and the graph structure to derive graph embeddings can enrich the information in the output and benefit the prediction model. Therefore, we validate our assumption by considering three different function embeddings: the null embedding, the Asm2Vec embedding, and the SAFE embedding. We train our model using the same samples selected from the benign and virus datasets. We use 15,000 samples from benign and virus datasets, respectively, and split the collected samples with an 8:2 ratio for training and testing purposes.

The evaluated function embedding models are derived as follows. We consider the pre-trained SAFE[5] model and the self-trained Asm2Vec and SAFE embedding models in the evaluation. The Asm2Vec embedding model is trained using the same training data to train the SAFE embedding model for the fair performance comparison of the two embeddings. The i2v blocks used in the pre-trained and self-trained SAFE model are the same. It is derived from about 1.3 million functions from UNIX executables and libraries compiled for different CPU architectures, including ARM and AMD64. The self-attentive network pre-trained by the authors is then trained with 548 thousand of functions from UNIX libraries compiled for ARM and AMD64 CPU architectures. All the binaries are compiled using three compilers and four optimization levels. For our self-trained Asm2Vec and SAFE embeddings, we compile 209 thousand functions from three open-source UNIX tools (binutils, curl, and openssl) using GCC with four optimization levels.

Table VI shows the model performance of using different function embeddings. With null embeddings, the average accuracy, precision, recall, and F1 score are 67%, 77%, 59%, and 63%, respectively. Compared to the results against graphs with modern function embeddings, the null embeddings lead to a worse result than the model trained with Asm2Vec and SAFE embeddings under the same setting. The monotonic structure of function call graphs leads to the dropped performance,

[5] https://github.com/gadiluna/SAFE

TABLE VI

Evaluation Results for Different Function Embeddings

Selected Function Embedding	Accuracy	Precision	Recall	F1-score
Null	0.679617	0.773491	0.593621	0.633797
Asm2Vec (self-trained)	0.897784	0.926837	0.863340	0.893472
SAFE (self-trained)	0.973412	0.976921	0.969835	0.973338
SAFE (pre-trained)	0.973787	0.981752	0.965715	0.973632

but having embeddings from nodes (functions) would help compensate for the weakness.

F. RQ5: Performance of Explainers

We evaluate the performance of the proposed model explainers (node pruning, edge pruning, and GNN explainer) from two aspects in this section: efficiency and stability. We measure the required time for different model explanation approaches. Moreover, we confirm that the last method's explanation result can obtain stable results.

For this experiment, we randomly select different numbers (ranging from 100 to 600) of samples from the sample pool and redo the experiment several times for each sample set. All of the experiments get consistent results. The node and edge pruning always take predictable and identical running times, and the GNN explainer consumes varying calculation times for each run. The difference between the former two methods and the last method is that node and edge pruning use step-by-step commands to rebuild the new graph, and the GNN explainer trains a new neural network every time to approximate the critical subgraph structure in a sample and thus receives various and unpredictable running time. Due to the giant time difference between node/edge pruning and GNN explaining, we select a small scale of 100 random samples for plotting. Also, this is to avoid the result dots of the previous two methods condensing in an unrecognizable cluster.

Figure 9(a) shows the scatter plot of how the scale of samples impacts the required time to perform an explanation. The Y-axis indicates the scale of the samples in terms of the number of edges. The X-axis shows the required time to process a sample. The figure shows that the more edges in a sample, the longer the time required to perform the explanation. The edge pruning approach is generally faster than the node pruning approach. Nevertheless, the two approaches are way faster than the GNN explainer-based approach.

Since the GNN explainer trains a new neural network model each time to explain a sample, we further investigate the explanation stability of the GNN explainer. We use the AZORult malware, which contains 1645 edges, to perform the evaluation. We use the GNN explainer to explain the model prediction 100 times for the same malware and observe the weights of edges assigned by the explainer to the sample. For each edge in the sample, we present the scatter plot of its average weight and the variance in Figure 9(b). The cumulative distribution function (CDF) of the variances is plotted in Figure 9(c). The two figures show that the explanation results from the GNN explainer are stable. Most edges are assigned with the same or similar weights, indicating that the explanations are consistent and stable between different runs.

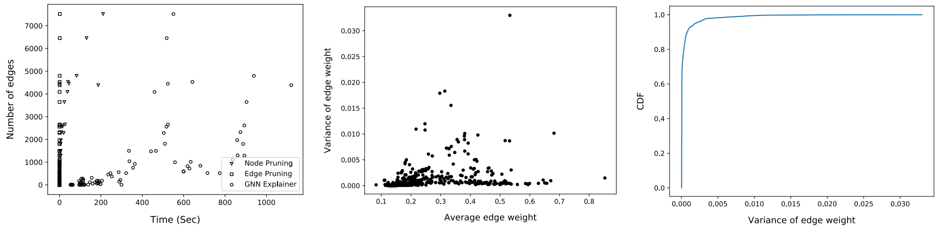
G. RQ6: Quantitative Analysis of Explainability

In this section, we attempt to quantify the quality of our generated explanations. We use the functions annotated in Lumina as expert-annotated explanations and measure the distance from the top explainer-ranked edges toward these targets. Lumina is a service hosted by IDA Pro (available since version 7.2 was released in 2018) to collect and share metadata from reverse engineers. It holds metadata (function names, prototypes, comments, operand types, and other info) about investigated functions. Any IDA users can send or receive metadata from the Lumina service if they are willing to share. For the annotated functions retrieved from Lumina, we remove the standard library APIs and drop functions that have no relationship with other functions in a sample. A reverse engineer would prefer to annotate functions they think are engaging in the reversing process. Therefore, to validate the explanation quality of our proposed explainers, we are interested in how many steps are required to trace from the explainer-ranked edges to the expert-annotated functions.

We use the three ransomware samples, Lockbit, Phobos, and WannaCry, for evaluation due to the samples and annotations in the Lumina server. We only use a limited number of samples for the following reasons. First, having quantitative measurement means we need ground truth from experts. The Lumina service is a good source of ground truth based on the assumption that users would annotate their interested function in a reverse engineering process. Second, although there are many samples, only a limited number of them have been annotated in the Lumina service. Since the Lumina service has been available since 2018 (IDA Pro 7.2), we choose samples available near 2018 or after 2018. Third, we choose ransomware because they are popular, and its availability in the Lumina database would be better than other types of samples in the Lumina service.

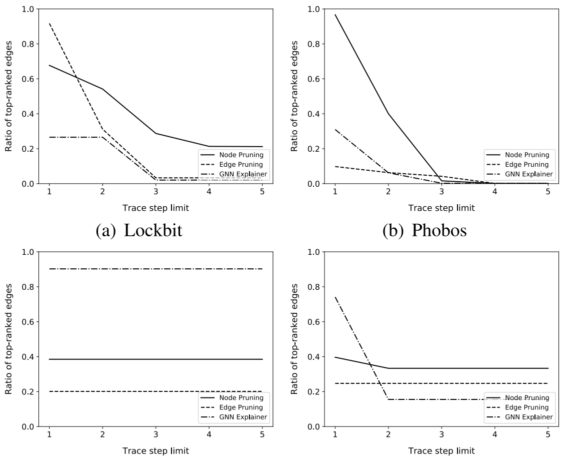
Note that the GNN model used to perform the detection is trained *without* any ransomware samples. Lockbit, Phobos, and WannaCry have 468, 291, and 132 nodes and 783, 807, and 174 edges, respectively. We retrieved 43, 30, and 16 annotated functions for the three samples, respectively, from the Lumina server as of the time of conducting the experiments. Since the explainers output the rank of essential edges, we expect the explainer-selected top-ranked edges would "cover" *all* expert-annotated functions. The definition of coverage is that tracing from either one of the two nodes associated with a selected edge can reach the expert-selected function within t steps (forward or backward). Figure 10 shows the explanation quality evaluation results based on annotations from Lumina. The X-axis is the trace step limit t, and the Y-axis indicates the ratio of top-ranked edges required (lower is better) to cover all expert-annotated functions. The experiment results show that all the explainers perform well with a trace step limit of three. The GNN explainer is generally better than the edge-pruning and node-pruning explainers. In the best case, we only need 2% of selected edges to cover all expert-annotated functions in Phobos.

Readers may notice that the explainers perform poorly for the WannaCry sample presented in Figure 10(c). This is because the model may have learned some malicious features from typical malicious software and then recognizes ransomware samples based on these features. However, due to the limited amount of ransomware samples in the training data, the model could only partially discover some features held by



(a) Explanation time for different scales of samples. (b) Average edge weights and their variances. (c) CDF of weight variances explained by the GNN explainer.

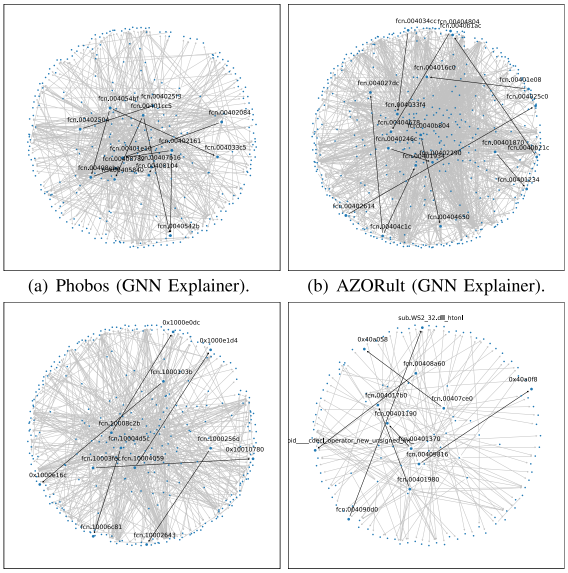
Fig. 9. Performance evaluations for explainer efficiency and stability.



(a) Lockbit (b) Phobos

(c) WannaCry (d) WannaCry (updated model)

Fig. 10. Quantitative analysis results for explanation quality based on the annotated metadata from Lumina.



(a) Phobos (GNN Explainer). (b) AZORult (GNN Explainer).

(c) Equation (GNN Explainer). (d) WannaCry (GNN Explainer).

Fig. 11. Case studies for selected sample predictions explained by discussed model explainer.

them, thus creating unsatisfied explanations for unknown or unique ransomware.

Training a "good detector" and a "good explainer" are two different stories. We use a simple example to illustrate the difference. Typical ransomware often contains several features, including (1) compromising a user, (2) making itself persistent, (3) scanning for interesting files, and (4) performing encryption. While features relevant to (1) and (2) can be observed in regular malicious software, they are not unique to ransomware and are not interesting to a reverse engineer. Since the unique behavior of ransomware (such as (3) and (4) in the example) are not learned in the model, the model may still detect that the sample is malicious based on features learned from (1) and (2). However, we would require features like (3) and (4) to explain the sample well. To validate our assumption, we add a small number of ransomware samples (200 samples) from VirusTotal and retrain the GNN detection model using the sample settings in RQ V-C. We then evaluate the explanation quality between the models trained with and without ransomware samples.

With the updated model, Phobos and Lockbit achieve similar explanation performance compared to previous results, but the explanation output quality of WannaCry encounters massive growth, especially on the GNN explainer. The improvements in the explanation quality can be clearly observed by comparing Figures 10(c) and 10(d). Though the two detection models obtain similar detection performance, the GNN explainer receives a significant drop in the ratio of required top-ranked edges (from 90% down to 15%) when the trace step limit is three. To conclude, the evaluation results show that all three explanation approaches can capture essential subgraph structures in the call graphs of unknown samples. Moreover, training models with target-similar samples may boost the explanation quality for targeted samples.

H. RQ7: Explainable Behavior

This section performs qualitative analyses of the explained results by discussing the clues reported from the model. We select four real-world samples, Phobos, AZORult, Equation (APT), and WannaCry, and compare the essential parts recognized by our model against our reversing engineering analysis results and external experts' analysis reports [70], [71], [72], [73]. In conclusion, the explainer provides highly relevant hints for program behavior analysis. Note that the recognized edges and subgraphs can be connected or disconnected, depending on the implementation of critical functions and the selection ratio for critical edges. We briefly summarize the results of the GNN explainer as follows.

1) The Phobos Sample: Figure 11(a) shows the recognized edges for Phobos ransomware. The explainer approach reports the functions used for file enumeration and data encryption, which is closely relevant to the major functionalities reported for the sample.

2) The AZORult Sample: Figure 11(b) shows the recognized edges for AZORult info stealer. The explainer successfully finds the entry functions that collect Internet Explorer's

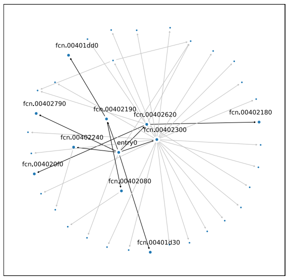


Fig. 12. GNN explainer result for WastedLocker (stage 1).

sensitive data. By tracing two steps from the entry functions, it reaches info-stealing functions, including collecting cookies and passwords from browsers and stealing crypto wallet data.

3) The Equation Sample: Figure 11(c) shows the recognized edges for Equation APT. The explainer identifies the DLL installer functions, which are highly relevant to the dropper implementation. The observations align with multiple security vendors reported on the VirusTotal website, which indicates the sample is a dropper for APT installation.

4) The WannaCry Sample: Figure 11(d) shows the recognized edges for WannaCry ransomware. We can see that the explainer identifies several critical edges highly relevant to the attacks presented in CVE-2017-0143. It indicates the function used to scan internal networks, recognize SMB services, and create socket connections to victims.

I. RQ8: Deal With Packed Samples

Although we removed samples packed by well-known packers from the training set, we want to know if our model can still find critical parts from packed binaries. There are indeed samples packed by customized or unknown packers or obfuscators. Nevertheless, it works well with our proposed solution because reverse engineering is a progressive process, and identifying essential parts that perform unpacking and deobfuscating is also one important process to realize the implementation of a malicious sample.

In the case of analyzing a packed sample, the objective of our approach is to identify the routines used to perform unpacking. This section uses real-world ransomware called WastedLocker to perform the evaluation. We choose it because it is two-stage ransomware, where the first stage is a customized self-made encryption-based packer. We explain the prediction using the GNN explainer and plot the results in Figure 12. The figure marks the top 10 essential edges, including the decryption and payload loading functions. Moreover, it also discovers the function to check the runtime environment and points out the APIs that the function uses. The result indicates that our proposed approach works well on packed binaries.

J. RQ9: Observation of Reverse Engineering Tools

Readers may wonder why we choose the open-sourced Radare2 (r2) to perform the experiments. We choose Radare2 for the following reasons. First, a tool must support automated analysis. Although IDA supports Python script integration, it is not available in the free version but only in the Pro version. Second, a tool must be efficient. Our experiment results show that Radare2 is about one point five to two times faster than IDA Pro in most cases. Third, it would benefit the community if the experiments were easier to reproduce, and working with an open-sourced tool would make it easier for the community.

```
# Python script for Radare2
for s in samples:
    t0 = time.time()
    r = r2pipe.open(s)
    r.cmd('aaa')
    aflj = r.cmd('aflj')
    r.quit()
    t1 = time.time()
    # store results into database
```

Fig. 13. Automated analysis script for Radare2.

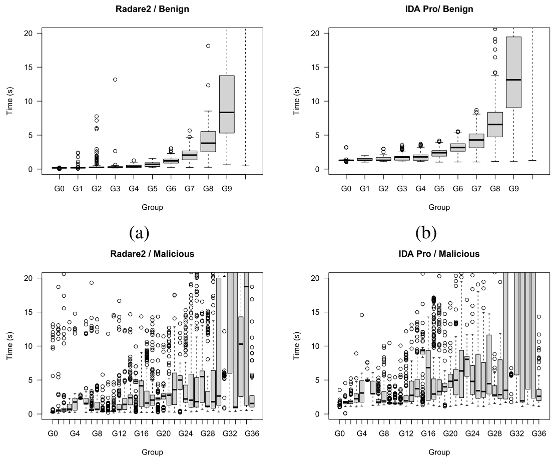
```
# shell script for invoking IDA Pro
for s in $samples; do
    $IDA -A -Sscript.py "-$s"
done
```

```
# script.py for IDA Pro
funcs = []
t0 = time.time()
auto_wait()
for seg in Segments():
    for func in Functions(seg, get_segm_end(seg)):
        funcname = get_func_name(func)
        for (begin, end) in Chunks(func):
            funcs.append({
                'offset': begin,
                'name': funcname,
                'size': end-begin})
t1 = time.time()
# store results into database
```

Fig. 14. Automated analysis script for IDA Pro.

We conduct performance measurements against our choice (Radare2) against the well-known commercial reverse engineering tool IDA Pro to validate our choice. We use samples from our collected datasets containing 45,611 benign samples and 75,276 malicious samples to measure the performance of the two tools. For Windows portable executable (PE) files, we skip files implemented in .Net frameworks because they are bytecode-based executables and cannot be completely decoded by the selected tools. The final datasets contain 20,778 benign samples and 72,894 malicious samples, which are 45.55% and 96.94% of the benign and malicious samples, respectively. We sort the files in the benign and malicious datasets independently in ascendant order based on the file sizes and group every 2000 files into subgroups based on the sorted result. As a result, there are 11 groups (from G0 to G10) for benign samples and 37 groups (from G0 to G36) for malicious samples. We finally randomly select 10% of samples from each group for performance evaluation.

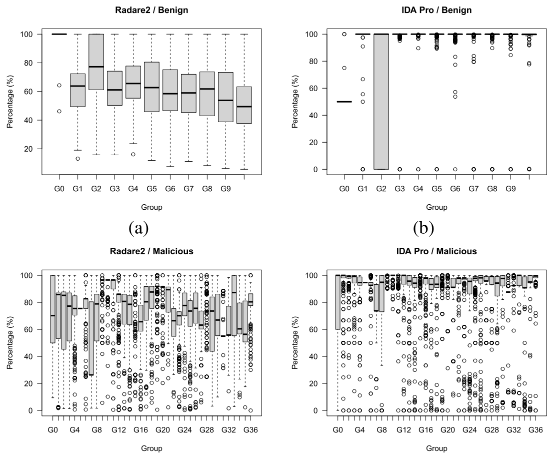
We use Radare2 version 5.4 and IDA Pro version 8.2 to conduct the experiments. We implement scripts to automate the analysis process with default settings and record the required processing time and the functions they discovered. The scripts implemented for the compared two tools are illustrated in Figures 13 and 14, respectively. For Radare2, the Python script iterates through all sample files. For each sample file, it invokes Radare2 through the pipeline interface and then uses the "aaa" command and the "aflj" command to analyze the files and store analyzed information in JSON format. For IDA Pro, we use a batch script to iteratively command the IDA Pro to analyze a sample and invoke a given Python script. The script waits until IDA Pro finishes analyzing a sample and retrieves the required function information, including offsets, names,



(a) (b)

(c) (d)

Fig. 15. Sample processing time for Radare2 and IDA Pro.



(a) (b)

(c) (d)

Fig. 16. Percentage of recognized functions for Radare2 and IDA Pro.

and sizes. The results are also stored in a JSON-compatible format.

We use box plots to present the measured processing time in Figure 15. The following observations can be made in the presented result. First, both Radare2 and IDA Pro have several outliers having much longer analysis time than others. Because of these outliers, we set a Y-axis limit at 20s to avoid squashed plots. Second, we see that both Radare2 and IDA Pro have similar processing time trends in analyzing the samples. The observation is based on the median values presented in the figures. Third, based on the measured running time, Radare2 runs one point five to two times faster than IDA Pro.

We further evaluate the identified functions reported from the two compared tools. We only compare the functions reported from the tools because once we have recognized the functions, we can use the same disassembly tools to recognize `call` instructions and build the call graphs. Figure 16 shows the percentage of functions discovered by Radare2 and IDA Pro. The percentage is measured by the equation

$$p = \frac{\text{\# of functions recognized by Radare2 (or IDA Pro)}}{\text{\# of all distinct functions recognized by both tools}}.$$

IDA Pro generally reports much more functions than Radare2. However, if we take a closer look at the differences, we find that the heuristics employed by the tools cause the differences. We summarize several findings as follows. First, we notice that the two tools interpret exception handlers differently. If a program invokes an exception handler using the `call` instruction, Radare2 considers the exception handler as a function, but IDA Pro does not. Second, IDA Pro sometimes considers a jumping target as a function, but Radare2 does not. However, it is interesting that this phenomenon is only visible in the automation script invoked within IDA Pro but is invisible in the GUI. Third, IDA Pro employs an intelligent heuristic to *guess* possible indirect call targets in a program. The heuristic detects whether addresses within the text segment are stored in the data segment. Although there could be false positives, i.e., incorrectly considering a constant value as a text segment address, detected addresses are reported as functions.

Based on our experiment results presented in this section, we see that the compared tools have their strengths and weaknesses. Reverse engineering tools could have different interpretations for the same sample. However, it might be challenging to tell which one is better than the other due to the design choices employed by the tools. Although different interpretations may lead to different results, the differences are not because of *inaccurate* but *inconsistent* interpretations of different tools. The more important thing is that we should use the same tool to perform the interpretation and generate the datasets instead of mixing the usage of tools to ensure that the system has a consistent view of program structures.

VI. Conclusion and Future Work

In this study, we develop the MalwareExpert system, which is composed of our proposed graph neural network-based malware classification model (the MalwareExpert model) and several model explainers. The system converts input samples into graph embeddings for detection and model explanation. In addition to achieving a high detection performance (97.3% accuracy and 96.5% recall rate), the model explainers recognize critical graph structures for classified samples. Our qualitative and quantitative analyses have shown that the identified functions provide accurate directions for accelerating malware binary analyses. In the best case, the top 2% of functions reported from the system can cover all expert-annotated functions in three steps. We believe that the MalwareExpert system has shed light on automated program behavior analysis. Our proposed architecture that combines a high-performance graph-based model and well-designed explainers is sufficiently generic to perform various automated program analysis purposes.

References

[1] *Malware Statistics & Trends Report*. Accessed: Oct. 22, 2022. [Online]. Available: https://www.av-test.org/en/statistics/malware/

[2] SCT Report. (2021). *Cyber Threat Intelligence for Navigating Today's Business Reality*. [Online]. Available: https://www.cerdant.com/wp-content/uploads/2021/09/2-2021-threat-report-midyear-summary.pdf

[3] D. Gibert, C. Mateu, and J. Planes, "The rise of machine learning for detection and classification of malware: Research developments, trends and challenges," *J. Netw. Comput. Appl.*, vol. 153, Mar. 2020, Art. no. 102526.

[4] E. Raff and C. Nicholas, "A survey of machine learning methods and challenges for windows malware classification," 2020, *arXiv:2006.09271*.

[5] E. Raff, J. Sylvester, and C. Nicholas, "Learning the pe header, malware detection with minimal domain knowledge," in *Proc. 10th ACM Workshop Artif. Intell. Secur.*, Nov. 2017, pp. 121–132.

[6] E. Raff et al., "An investigation of byte n-gram features for malware classification," *J. Comput. Virol. Hacking Techn.*, vol. 14, no. 1, pp. 1–20, Feb. 2018.

[7] R. Zak, E. Raff, and C. Nicholas, "What can N-grams learn for malware detection?" in *Proc. 12th Int. Conf. Malicious Unwanted Softw. (MALWARE)*, Oct. 2017, pp. 109–118.

[8] M. Kalash, M. Rochan, N. Mohammed, N. D. B. Bruce, Y. Wang, and F. Iqbal, "Malware classification with deep convolutional neural networks," in *Proc. 9th IFIP Int. Conf. New Technol., Mobility Secur. (NTMS)*, Feb. 2018, pp. 1–5.

[9] D. Gibert, C. Mateu, J. Planes, and R. Vicens, "Using convolutional neural networks for classification of malware represented as images," *J. Comput. Virol. Hacking Techn.*, vol. 15, no. 1, pp. 15–28, Mar. 2019.

[10] D. Vasan, M. Alazab, S. Wassan, B. Safaei, and Q. Zheng, "Image-based malware classification using ensemble of CNN architectures (IMCEC)," *Comput., Secur.*, vol. 92, May 2020, Art. no. 101748.

[11] Q. Le, O. Boydell, B. M. Namee, and M. Scanlon, "Deep learning at the shallow end: Malware classification for non-domain experts," *Digit. Invest.*, vol. 26, pp. S118–S126, Jul. 2018.

[12] H. S. Anderson and P. Roth, "EMBER: An open dataset for training static PE malware machine learning models," 2018, *arXiv:1804.04637*.

[13] H.-D. Pham, T. D. Le, and T. N. Vu, "Static PE malware detection using gradient boosting decision trees algorithm," in *Proc. Int. Conf. Future Data Secur. Eng.*, 2018, pp. 228–236.

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

[15] B. Perozzi, R. Al-Rfou, and S. Skiena, "DeepWalk: Online learning of social representations," in *Proc. 20th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2014, pp. 701–710.

[16] A. Grover and J. Leskovec, "Node2vec: Scalable feature learning for networks," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2016, pp. 855–864.

[17] N. Shervashidze, P. Schweitzer, E. J. Van Leeuwen, K. Mehlhorn, and K. M. Borgwardt, "Weisfeiler-Lehman graph kernels," *J. Mach. Learn. Res.*, vol. 12, no. 9, pp. 2539–2561, Nov. 2011.

[18] P. Hell and J. Nesetril, *Graphs and Homomorphisms*, vol. 28. London, U.K.: Oxford Univ. Press, 2004.

[19] H. Dai, B. Dai, and L. Song, "Discriminative embeddings of latent variable models for structured data," in *Proc. 33rd Int. Conf. Mach. Learn.*, vol. 48, 2016, pp. 2702–2711.

[20] M. Grohe, "Word2vec, node2vec, graph2vec, X2vec: Towards a theory of vector embeddings of structured data," in *Proc. 39th ACM SIGMOD-SIGACT-SIGAI Symp. Princ. Database Syst.*, Jun. 2020, pp. 1–16.

[21] M. Hassen and P. K. Chan, "Scalable function call graph-based malware classification," in *Proc. 7th ACM Conf. Data Appl. Secur. Privacy*, Mar. 2017, pp. 239–248.

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

[23] L. Massarelli, G. A. Di Luna, F. Petroni, R. Baldoni, and L. Querzoni, "SAFE: Self-attentive function embeddings for binary similarity," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*, 2019, pp. 309–329.

[24] J. Johns. (2017). *Representation Learning for Malware Classification*. [Online]. Available: https://www.fireeye.com/content/dam/fireeye-www/blog/pdfs/malware-classification-slides.pdf

[25] H. Hashemi, A. Azmoodeh, A. Hamzeh, and S. Hashemi, "Graph embedding as a new approach for unknown malware detection," *J. Comput. Virol. Hacking Techn.*, vol. 13, no. 3, pp. 153–166, Aug. 2017.

[26] R. V. Mises and H. Pollaczek-Geiringe, "Praktische Verfahren der Gleichungsauflösung," *ZAMM-J. Appl. Math. Mech. Zeitschrift Für Angew. Mathematik und Mech.*, vol. 9, no. 1, pp. 58–77, 1929.

[27] A. Z. Broder, "On the resemblance and containment of documents," in *Proc. Compress. Complex. SEQUENCES*, 1997, pp. 21–29.

[28] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck, "Structural detection of Android malware using embedded call graphs," in *Proc. ACM Workshop Artif. Intell. Secur.*, Nov. 2013, pp. 45–54.

[29] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 363–376.

[30] Y. Fan, S. Hou, Y. Zhang, Y. Ye, and M. Abdulhayoglu, "Gotcha–Sly malware: Scorpion a metagraph2vec based malware detection system," in *Proc. 24th ACM SIGKDD Int. Conf. Knowl. Discovery, Data Mining*, Jul. 2018, pp. 253–262.

[31] J. Zhang, L. Yan, R. Wang, C. Tian, and Z. Duan, "Malware detection using CNN via word embedding," in *Proc. Int. Conf. Dependable Syst. Their Appl. (DSA)*, 2021, pp. 600–607.

[32] M. Krčál, O. Švec, M. Bálek, and O. Jašek, "Deep convolutional malware classifiers can learn from raw executables and labels only," in *Proc. 6th Int. Conf. Learn. Represent.*, 2018, pp. 1–4.

[33] S. E. Coull and C. Gardner, "Activation analysis of a byte-based deep neural network for malware classification," in *Proc. IEEE Secur. Privacy Workshops (SPW)*, May 2019, pp. 21–27.

[34] (2017). *The Undocumented Microsoft 'Rich' Header*. [Online]. Available: http://bytepointer.com/articles/the_microsoft_rich_header.htm

[35] L. Demetrio, B. Biggio, G. Lagorio, F. Roli, and A. Armando, "Explaining vulnerabilities of deep learning to adversarial malware binaries," 2019, *arXiv:1901.03583*.

[36] S. Bose, T. Barao, and X. Liu, "Explaining AI for malware detection: Analysis of mechanisms of MalConv," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Jul. 2020, pp. 1–8.

[37] R. Korine and D. Hendler, "DAEMON: Dataset/platform-agnostic explainable malware classification using multi-stage feature mining," *IEEE Access*, vol. 9, pp. 78382–78399, 2021.

[38] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 472–489.

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

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

[41] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," 2015, *arXiv:1511.05493*.

[42] Y. Gao, H. Hasegawa, Y. Yamaguchi, and H. Shimada, "Malware detection using attributed CFG generated by pre-trained language model with graph isomorphism network," in *Proc. IEEE 46th Annu. Comput., Softw., Appl. Conf. (COMPSAC)*, Jun. 2022, pp. 1495–1501.

[43] B. Wu, Y. Xu, and F. Zou, "Malware classification by learning semantic and structural features of control flow graphs," in *Proc. IEEE 20th Int. Conf. Trust, Secur. Privacy Comput. Commun. (TrustCom)*, Oct. 2021, pp. 540–547.

[44] M. Someya, Y. Otsubo, and A. Otsuka, "FCGAT: Interpretable malware classification method using function call graph and attention mechanism," in *Proc. Netw. Distrib. Syst. Secur. (NDSS) Symp.*, 2023, pp. 1–10.

[45] L. Zhang, P. Liu, Y. Choi, and P. Chen, "Semantics-preserving reinforcement learning attack against graph neural networks for malware detection," *IEEE Trans. Dependable Secure Comput.*, vol. 20, no. 2, pp. 1390–1402, Mar. 2023.

[46] W. W. Lo, S. Layeghy, M. Sarhan, M. Gallagher, and M. Portmann, "Graph neural network-based Android malware classification with jumping knowledge," in *Proc. IEEE Conf. Dependable Secure Comput. (DSC)*, Jun. 2022, pp. 1–9.

[47] C. Li et al., "DMalNet: Dynamic malware analysis based on API feature engineering and graph learning," *Comput., Secur.*, vol. 122, Nov. 2022, Art. no. 102872.

[48] Y. He, Y. Liu, L. Wu, Z. Yang, K. Ren, and Z. Qin, "MsDroid: Identifying malicious snippets for Android malware detection," *IEEE Trans. Dependable Secure Comput.*, vol. 20, no. 3, pp. 2025–2039, May/Jun. 2023.

[49] R. Ying, D. Bourgeois, J. You, M. Zitnik, and J. Leskovec, "GNNExplainer: Generating explanations for graph neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32, 2019, p. 9240–9251.

[50] R. Isawa, M. Morii, and D. Inoue, "Comparing malware samples for unpacking: A feasibility study," in *Proc. 11th Asia Joint Conf. Inf. Secur. (AsiaJCIS)*, Aug. 2016, pp. 155–160.

[51] A. Kalysch, J. Götzfried, and T. Müller, "VMAttack: Deobfuscating virtualization-based packed binaries," in *Proc. 12th Int. Conf. Availability, Rel. Secur.*, Aug. 2017, pp. 1–10.

[52] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable code," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 674–691.

[53] S. Choi, T. Chang, C. Kim, and Y. Park, "X64Unpack: Hybrid emulation unpacker for 64-bit windows environments and detailed analysis results on VMProtect 3.4," *IEEE Access*, vol. 8, pp. 127939–127953, 2020.

[54] Hex Rays. *IDA Pro*. Accessed: Oct. 22, 2022. [Online]. Available: https://hex-rays.com/ida-pro/

[55] NSA's Research Directorate. *Ghidra*. Accessed: Oct. 22, 2022. [Online]. Available: https://ghidra-sre.org/

[56] *Radare*. Accessed: Oct. 22, 2022. [Online]. Available: https://rada.re/n/

[57] Z. Lin et al., "A structured self-attentive sentence embedding," 2017, *arXiv:1703.03130*.

[58] U. Alon and E. Yahav, "On the bottleneck of graph neural networks and its practical implications," 2020, *arXiv:2006.05205*.

[59] G. Li, C. Xiong, A. Thabet, and B. Ghanem, "DeeperGCN: All you need to train deeper GCNs," 2020, *arXiv:2006.07739*.

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

[61] W. Lu et al., "SkipNode: On alleviating performance degradation for deep graph convolutional networks," 2021, *arXiv:2112.11628*.

[62] J. Lei Ba, J. Ryan Kiros, and G. E. Hinton, "Layer normalization," 2016, *arXiv:1607.06450*.

[63] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, "TESSERACT: Eliminating experimental bias in malware classification across space and time," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 729–746.

[64] *Desktop Operating System Market Share World*. Accessed: Oct. 22, 2022. [Online]. Available: https://gs.statcounter.com/os-market-share/desktop/world

[65] H. Aghakhani et al., "When malware is Packin' heat; limits of machine learning classifiers based on static analysis features," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2020, pp. 1–20.

[66] Horsicq. *Detect-it-Easy*. Accessed: Oct. 22, 2022. [Online]. Available: https://github.com/horsicq/Detect-It-Easy

[67] L. Van der Maaten and G. Hinton, "Visualizing data using t-SNE," *J. Mach. Learn. Res.*, vol. 9, no. 11, Nov. 2008, pp. 2579–2605.

[68] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. Nicholas, "Malware detection by eating a whole EXE," 2017, *arXiv:1710.09435*.

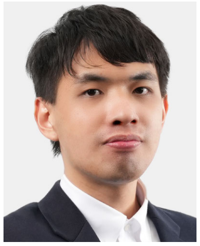
[69] H. Anderson and P. Roth. *Elastic Malware Benchmark for Empowering Researchers*. Accessed: Oct. 22, 2022. [Online]. Available: https://github.com/elastic/ember

[70] A. Berry, J. Homan, and R. Eitzman. (2017). *Wannacry Malware Profile*. [Online]. Available: https://www.fireeye.com/blog/threat-research/2017/05/wannacry-malware-profile.html

[71] X. Zhang. (2020). *Deep Analysis—The Eking Variant of Phobos Ransomware*. [Online]. Available: https://www.fortinet.com/blog/threat-research/deep-analysis-the-eking-variant-of-phobos-ransomware

[72] Hasherezade. (2019). *A Deep Dive Into Phobos Ransomware*. [Online]. Available: https://www.malwarebytes.com/blog/news/2019/07/a-deep-dive-into-phobos-ransomware

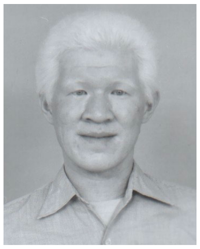
[73] Cyble. *A Deep-Dive Analysis of Azorult Stealer*. Accessed: Oct. 22, 2022. [Online]. Available: https://blog.cyble.com/2021/10/26/a-deep-dive-analysis-of-azorult-stealer



Yi-Hsien Chen received the B.S. degree in computer science from the National Chiao Tung University (NCTU). He is currently pursuing the Ph.D. degree with the Department of Electrical Engineering, National Taiwan University. He is a Research Assistant with the Department of Computer Science, National Yang Ming Chiao Tung University, and a Security Researcher of the CyCraft Research Team. His work has published in IEEE DSC and ACM ASIACCS. He has also spoken at HITB CyberWeek, AVTokyo, HIT CON, SECCON, and CODE BLUE. Additionally, he was a member of the BambooFox CTF Team, NYCU, and has participated in several CTFs. His research focuses on automatic malware analysis techniques and various cybersecurity topics. He has utilized symbolic execution, machine learning, and other techniques to improve malware analysis speed. He has won 12th and second place in DEFCON 26 and DEFCON 27, respectively, with BFS and BFKinesiS CTF teams.

Si-Chen Lin received the B.S. degree in computer science from the National Chiao Tung University and the master's degree in cybersecurity from the National Taiwan University. He received the prestigious Master's Thesis Award. He is a cybersecurity professional specializing in the intersection of cybersecurity and machine learning. His research expertise lies in developing innovative approaches that leverage machine-learning algorithms to identify and mitigate security threats. His research findings have been recognized at renowned conferences, including HITCON and CYBERSEC, where he has engaged in intellectual discourse and fostered knowledge exchange within the cybersecurity community. His exceptional achievements underscore his dedication and profound contributions to the field. He advances the discipline by integrating cutting-edge machine learning techniques into cybersecurity practices and enhancing security measures. His work exemplifies the potential synergies between cybersecurity and machine learning, improving the digital landscape's threat detection and mitigation strategies. He is a member of BambooFox, BFS, and TSJ CTF teams and focuses on binary exploitation and reverse engineering. In CTF competitions, he has achieved top rankings in several world-class CTFs, such as DEFCON and HITCON CTF.

Szu-Chun Huang received the B.S. degree from the National Taiwan University and the M.S. degree from the National Yang Ming Chiao Tung University. She is currently pursuing the Ph.D. degree in computer science with the Faculty of Technology, Policy and Management, Delft University of Technology. Her research focuses on scanning and benchmarking internet-wide vulnerabilities and aims to investigate the hidden factors influencing security patching behaviors.



Chin-Laung Lei (Member, IEEE) received the B.S. degree in electrical engineering from the National Taiwan University in 1980 and the Ph.D. degree in computer science from The University of Texas at Austin in 1986. From 1986 to 1988, he was an Assistant Professor with the Computer and Information Science Department, The Ohio State University, Columbus, OH, USA. In 1988, he joined the Department of Electrical Engineering Faculty, National Taiwan University, where he is currently a Professor. His current research interests include computer and network security, cryptography, parallel and distributed processing, the design and analysis of algorithms, and operating system design. He is a member of the Association for Computing Machinery.

Chun-Ying Huang (Senior Member, IEEE) received the Ph.D. degree from the Electrical Engineering Department, National Taiwan University, in 2007. He is a Professor with the Department of Computer Science, National Yang Ming Chiao Tung University (formerly known as National Chiao Tung University), where he leads the Security and Systems Laboratory. He is also the Director of the Institute of Network Engineering, National Yang Ming Chiao Tung University. His research interests include system security, multimedia networking, and mobile computing. He served as a PI or Co-PI for several teaching, research, and industrial projects in network and system security, including the Information Security Incubation Program (ISIP and AIS3), Ministry of Education (MoE); and the Taiwan Information Security Center Program (TWISC), NYCU, National Science and Technology Council (NSTC).