MalGraph: Hierarchical Graph Neural Networks for Robust Windows Malware Detection

Xiang Ling[1,2], Lingfei Wu[3], Wei Deng[2], Zhenqing Qu[2], Jiangyu Zhang[2], Sheng Zhang[2], Tengfei Ma[4], Bin Wang[5], Chunming Wu[2](✉) and Shouling Ji[2](✉)
[1]Institute of Software, Chinese Academy of Sciences, [2]Zhejiang University, [3]JD.COM Silicon Valley Research Center, [4]IBM Research, [5]Hangzhou Hikvision Digital Technology Co., Ltd. E-mails: lingxiang@iscas.ac.cn, lwu@email.wm.edu, {dengwei, quzhenqing, zhangjiangyu, zs_zjhz, wuchunming, sji}@zju.edu.cn, tengfei.ma1@ibm.com, wangbin2@hikvision.com

***Abstract*—With the ever-increasing malware threats, malware detection plays an indispensable role in protecting information systems. Although tremendous research efforts have been made, there are still two key challenges hindering them from being applied to accurately and robustly detect malwares. Firstly, most of them represent executables with shallow features, but ignore their semantic and structural information. Secondly, they are primarily based on representations that can be easily modified by attackers and thus cannot provide robustness against adversarial attacks. To tackle the challenges, we present MalGraph, which first represents executables with hierarchical graphs and then uses an end-to-end learning framework based on graph neural networks for malware detection. In particular, a hierarchical graph consists of a function call graph that captures the interaction semantics among different functions at the inter-function level and corresponding control-flow graphs for learning the structural semantics of each function at the intra-function level. We argue the abstraction and hierarchy nature of hierarchical graphs makes them not only easy to capture rich structural information of executables, but also be immune to adversarial attacks. Evaluations show that MalGraph not only outperforms state-of-the-art malware detection, but also exhibits stronger robustness against adversarial attacks by a large margin.**

***Index Terms*—Software Security, Malware Detection, Representation Learning, Graph Neural Network**

I. Introduction

With the rapid development and application of softwares, malicious softwares (*i.e.*, malwares) are becoming and spreading one of the most serious threats to perform malicious activities on information systems, including asking for a large ransom, stealing confidential information, and cryptocurrency-mining. According to the AV-TEST statistics [1], the total number of observed malwares in various operating systems (*e.g.*, Windows, Linux, macOS, *etc*) has increasingly surged from 470 million in 2015 to 1,139 million in 2020, in which more than 100 million are newly emerged. Due to the world-wide popularity of the Windows family of operating systems for both personal and business users, it is also observed that over 80% of malwares target the Windows family and the majority of malwares are in the file format of portable executable (PE)[1], which is the main scope of this work.

Meanwhile, tremendous research efforts have been made to defend against such Windows PE malwares. At a high level, these defensive solutions can broadly be classified into static analysis and dynamic analysis. To be exact, static analysis solutions detect the suspicious executables by extracting their static features without executing them, while dynamic analysis solutions detect malwares based on their executing behaviors interacted with an isolated environment. As dynamic analysis solutions are both resource and time expensive, they are hard to be scalable and thus less ubiquitously deployed. Therefore, in this paper, we focus on static malware detection, which allows malwares to be detected before execution.

Traditional static detection can be traced back to the classic signature-based detection, which blacklists the suspicious malwares based on a database of signatures of known malwares that are previously collected from infected systems. Apparently, the fatal drawback of signature-based detection is that they can only detect known malware threats, as they rely on known signatures generated from confirmed malwares.

Aiming at overcoming the above drawbacks of signature-based detection and improving the detecting performance, a variety of machine learning (ML)-based malware detection have been proposed [2]–[5], which are supposed to learn to discriminate malwares from goodwares (*i.e.*, short for benign softwares) based on the supervision of collected malwares and goodwares. Particularly, these ML-based detection methods first extract various feature sets (*e.g.*, PE header statistics, strings, and system call functions) from the PE files and then train various ML models (*e.g.*, SVM, GBDT) based on extracted feature sets. Although some of the existing ML-based detection offer promising detection performance, we argue that they rely heavily on hand-crafted features which are usually artifacts discovered from limited malware samples. The hand-crafted features make ML-based malware detection extremely difficult to generalize to other newly emerging malwares as they do not capture intrinsic properties of malwares.

More recently, to leverage the high learning capacity of deep learning (DL) models, there has been rising interest to explore more advanced neural networks (*e.g.*, MLP, CNN, and RNN) for the task of PE malware detection based on either raw bytes [6] or other representations (*e.g.*, binary grayscale images, n-gram byte sequences) [7], [8] of executable programs. Nevertheless, these DL-based malware detection still do not address the drawbacks of ML-based methods and suffer from two major limitations. i) They ignore the rich semantic

Chunming Wu and Shouling Ji are the corresponding authors.

[1]In our paper, we use the terms “executable programs”, “executables”, and “programs” interchangeably.

information of executables and thus cannot capture the intrinsic properties of malwares, such as how different functions interact in an executable, what are the semantics of different functions, *etc*. ii) They do not offer robust malware detection because they are built on the representations that can be easily modified while not affecting the functionality of executables. For example, employing a slight modification (*e.g.*, appending random bytes at the end of PE files) to malwares can easily bypass and invalidate the malware detection like MalConv [6] which is built on the raw bytes of executables.

To tackle the aforementioned challenges, in this paper, we propose a novel hierarchical graph neural network based on the hierarchical graph for robust malware detection, namely `MalGraph`, with two insights. Firstly, to capture the intrinsic properties of executables, we represent them with graph-structured objects which can offer rich structural information for the malware detection task. As depicted in Fig. 1, we represent an executable with a hierarchical graph, in which a function call graph (FCG) is used to learn the interaction semantics of different functions (at the inter-function level), and a control-flow graph (CFG) is further used to learn the structural semantics of each function (at the intra-function level). Secondly, for the executable, its hierarchical graph representation can be considered as one kind of abstraction, which cannot be easily modified while not affecting its functionality. As a result, we argue that the abstraction of hierarchical graph representation is more robust against adversarial attacks as it is able to abstract away from the induced irrelevant properties and focuses on the intrinsic properties of executables. For example, the hierarchical graph representation can theoretically immunize against the modifications like appending random bytes at the end of PE files and thus exhibits stronger robustness against adversarial attacks based on such modifications.

As shown in Fig. 2, the overview framework of `MalGraph` is mainly built on graph neural network (GNN) and consists of three key layers: ❶ Intra-function graph encoding layer leverages GNN and pooling networks to encode each CFG corresponding to each local function in an executable into an embedding vector; ❷ Inter-function graph encoding layer first uses the obtained vectors of local functions and the one-hot encoding of external functions to initialize nodes of FCG, and further applies GNN and pooling networks to learn an inter-function graph-level embedding vector for representing the executable; ❸ Prediction layer is performed by employing multiple MLPs on the learned inter-function embedding vector to learn the malicious probability for the given executable.

We systematically investigate the overall effectiveness of `MalGraph` compared with three state-of-the-art baseline detection on the wild dataset which contains hundreds of thousands of malwares and goodwares. We also compare and verify the robustness of `MalGraph` and baseline methods under the black-box adversarial attack. Experimental results demonstrate that `MalGraph` not only consistently outperforms state-of-the-art malware detection in terms of all detection metrics, but also exhibits stronger robustness against adversarial attacks by a large margin. In addition, we conduct ablation studies to evaluate the contribution of each part of `MalGraph`. To summarize, the main contributions of this paper are as follows.

- To the best of our knowledge, it is the first to represent an executable program with the hierarchical graph in which structural information is largely preserved and learned.
- We present `MalGraph`, a novel hierarchical graph neural network based on the hierarchical graph, to detect Windows malwares effectively and robustly.
- Extensive evaluation demonstrates that `MalGraph` outperforms three state-of-the-art baseline detection in terms of both detection effectiveness and robustness.

II. Hierarchical Graph Representation

Before introducing the model architecture of `MalGraph` for malware detection, in this section, we first elaborate on how to represent an executable with a hierarchical graph that incorporates the inter-function call graph (§ II-A) with intra-function control flow graphs (§ II-B). Then, we give the problem formulation of malware detection with the hierarchical graph (§ II-C). Furthermore, Fig. 1 illustrates the concept of hierarchical graph representation for an executable, and Table I summarizes the important notations in this paper.

TABLE I
Important notations and symbols in this paper.

Notations	Definitions or Descriptions
e	The executable program.
$\mathcal{G}_e$	The FCG of the given executable e.
n	The number of nodes in $\mathcal{G}_e$, $n = n_S + n_L$.
n_S	The number of nodes w.r.t external functions in $\mathcal{G}_e$.
n_L	The number of nodes w.r.t local functions in $\mathcal{G}_e$.
g_s	The s-th external function ($s = \{1, \cdots, n_S\}$).
f_ℓ	The ℓ-th local function ($\ell = \{1, \ldots, n_L\}$).
$\mathcal{G}_{f_\ell}$	The CFG of the ℓ-th local function f_ℓ.
$\mathbf{h}_{\ell,i}$	The hidden embedding vector of i-th node in $\mathcal{G}_{f_\ell}$.
$\mathbf{h}_{\mathcal{G}_{f_\ell}}$	The graph-level embedding vector of $\mathcal{G}_{f_\ell}$.
$\mathbf{z}_k$	The hidden embedding vector of k-th node in $\mathcal{G}_e$.
$\mathbf{z}_{\mathcal{G}_e}$	The graph-level embedding vector of $\mathcal{G}_e$.
$p_e \in [0,1]$	The predicted malicious probability for the executable e.

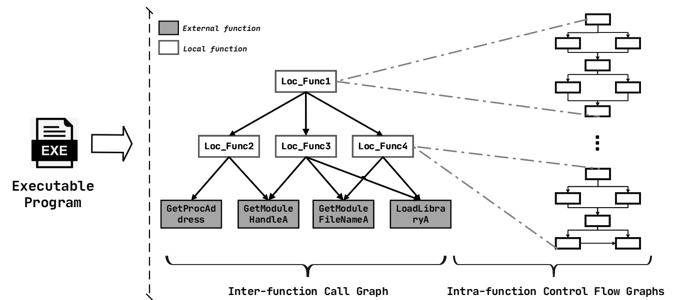


Fig. 1. The concept of hierarchical graph representation for an executable.

A. Inter-function Call Graph

To preserve and learn the structural information of how different functions of an executable interacts, we represent each executable with a directed graph in the form of a function call graph (FCG) [9], [10] that is defined as follows.

***Definition** 1 (Function Call Graph):* An executable program e can be represented as a function call graph (FCG) $\mathcal{G}_e$ =

$(\mathcal{V}_e, \mathcal{E}_e)$, where $\mathcal{V}_e$ is the finite set of nodes and each node $v \in \mathcal{V}_e$ is associated with one specific function; $\mathcal{E}_e \subseteq \mathcal{V}_e \times \mathcal{V}_e$ is the directed edge set and each edge from the caller node v_{i1} to the callee node v_{i2} implies a call from the function represented by v_{i1} to the function represented by v_{i2}.

For an executable, its functions (*i.e.*, nodes in FCG) can be classified into *external functions* and *local functions*. To be specific, external functions are system or library functions provided by the operating system (OS), including statically-linked functions (*e.g.*, "$fclose$" from the C standard library) and dynamically-imported functions (*e.g.*, "$GetProcAddress$" from the Kernel32.dll); while local functions are those functions that are elaborately written by program designers to implement specific functionalities. To make the data amenable to the input of DL models, it is important to convert the FCG of an executable into a graph-structured object as the input of `MalGraph`. That is, we need to encode every function (*i.e.*, node) in FCG with an initial feature vector.

In order to precisely express the implementation intention of each function, we employ different approaches to encode different functions with regard to external functions and local functions as follows. As external functions are system or library functions in the host OS, external functions are heavily reused and invoked by a large number of executable programs with the same function names (*e.g.*, "$fclose$", "$GetProcAddress$", *etc*), which precisely and uniquely express the functionalities of corresponding external functions. Therefore, for representing external functions, each of them is initially encoded as a one-hot vector based on their function names. Assuming there are a total of $|V|$ external functions commonly used in the host system, the initial feature vector of any external function is a $|V|$-dimensional binary vector.

In contrast, for local functions, their original function names are almost not preserved in the disassembled executables, because we cannot access the source codes of executables and function names might be omitted during the compilation and assembly process. Even though some function names of local functions are preserved, these names might not well express the intention of the implemented functions as they are named by millions of independent programmers rather than relatively fixed and official system providers (*e.g.*, libraries in the Windows OS). Therefore, it is obvious that we cannot represent local functions with their function names.

To strike a balance between effectiveness and efficiency for the task of malware detection, we employ a mixed strategy to represent an FCG. In particular, we represent nodes of external functions in the FCG with one-hot encoding based on function names, while nodes of local functions are further disassembled and represented with CFGs, which will be detailed as follows.

B. Intra-function Control Flow Graph

To learn the semantics of local functions in the FCG, we represent each local function with a directed graph in the form of CFG, which has been investigated for the function-level representation [11]–[14] and is formally defined as follows.

***Definition** 2 (Control Flow Graph):* An local function f can be represented as a control flow graph (CFG) $\mathcal{G}_f = (\mathcal{V}_f, \mathcal{E}_f)$, where $\mathcal{V}_f$ is a finite set of nodes and each node $u \in \mathcal{V}_f$ is a basic block, representing a sequence of instructions without branching, and $\mathcal{E}_f$ is a set of edges, representing control flow paths between nodes, *i.e.*, basic blocks.

The main reason for representing each local function with a CFG is that it contains semantic and structural information of assembly instructions, expressing rich logic and functionality of the disassembled local function. In CFG, each node is represented as a basic block, consisting of a maximal sequence of assembly instructions that includes an opcode and zero or more operands (*e.g.*, "$push\ esi$", "$move\ eax,\ 1$"). To be specific, a basic block executes from the first instruction to the last instruction sequentially and does not contain any branching instruction (*e.g.*, "$jmp\ [address]$", *etc*) except possibly for the last instruction.

To numerate the assembly instructions in basic blocks, we directly take the statistical information of instructions in each basic block as the initial node features for CFG. Particular, for each node in CFG, we count a list of numerical features (*i.e.*, $No.\ of\ call/transfer/arithmetic/logic/compare/move/termination/date\,declaration/total\,instructions/string\,or\,integer\ constants/offspring$), which have been extensively employed to represent CFG [11], [13], [15].

C. Problem Formulation of Malware Detection

The goal of malware detection is to classify executables by their functionalities, *i.e.*, malicious or benign. As introduced above, we represent the executable with the proposed hierarchical graph which incorporates the inter-function FCG with intra-function CFGs. Specifically, an executable e can be interpreted as an FCG $\mathcal{G}_e$ with a total of n nodes of functions, consisting of n_S external functions and n_L local functions, *i.e.*, $n = n_S + n_L$. Let $f_s\,(s = \{1,\ldots,n_S\})$ refer to the s-th external function, and $f_\ell\,(\ell = \{1,\ldots,n_L\})$ refer to the ℓ-th local function. As each local function can be further represented as a CFG, the CFG of f_ℓ is denoted as $\mathcal{G}_{f_\ell}\,(\ell = \{1,\ldots,n_L\})$. The goal of `MalGraph` is to train a robust model to predict the probability of malicious or benign for the given executable e.

III. Our Solution: MalGraph

A. Model Overview

As depicted in Fig. 2, `MalGraph` mainly consists of three layers: ❶ Intra-function graph encoding layer; ❷ Inter-function graph encoding layer and ❸ Prediction layer. In the following subsection, we will elaborate on each layer in details.

1) Intra-function Graph Encoding Layer: The first layer of `MalGraph` is the intra-function graph encoding layer, which attempts to learn the semantic information of CFG of every local function at the intra-function level. As formulated in § II-C, for the given executable e, if n_L local functions (*i.e.*, $\{f_\ell\}_{\ell=1}^{\ell=n_L}$) are disassembled, there would be a total of n_L CFGs (*i.e.*, $\{\mathcal{G}_{f_\ell}\}_{\ell=1}^{\ell=n_L}$), correspondingly. More recently, aiming at learning effective node embeddings of graphs, a large number of GNNs have been proposed [16]–[18]. Motivated by

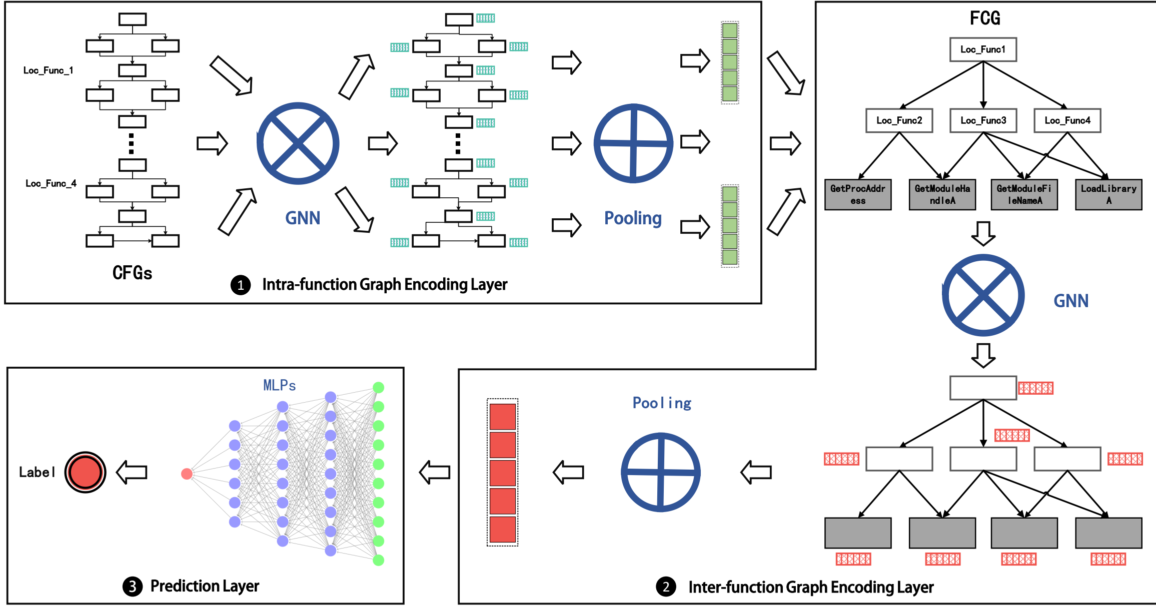


Fig. 2. The overview framework of MalGraph, which mainly contains three components: ❶ the intra-function graph encoding layer, ❷ the inter-function graph encoding layer and ❸ the prediction layer. For simplicity, we denote the multi-layer GNN and the graph pooling operation with $\bigotimes$ and $\bigoplus$, respectively.

the great success of GNN-based models obtained from various graph-related applications (*e.g.*, node classification [19], [20], graph classification [21], [22], graph generation [23], [24], graph similarity learning [15], [25], *etc*), we also consider GNNs for learning node embeddings of CFGs in this layer.

In particular, we first employ multiple layers of simplified GraphSAGE models [19] to iteratively generate the hidden embeddings for all nodes in each graph $\mathcal{G}_{f_\ell}$ $(\ell=\{1,\cdots,n_L\})$. For each node $u_{\ell,i}$ in the CFG $\mathcal{G}_{f_\ell}$ of the function f_ℓ, its hidden embedding vector at the t_1-th layer (*i.e.*, $\mathbf{h}_{\ell,i}^{(t_1)}$) is formulated as follows. Intuitively, $\mathbf{h}_{\ell,i}^{(t_1)} \in \mathbb{R}^{d^{(t_1)}}$ is accumulated from the node itself and its adjacency nodes in the preceding (t_1-1)-th layer with the self-passing function (*i.e.*, f_{node}) and the message passing function (*i.e.*, f_{msg}), respectively. $d^{(t_1)}$ represents the output dimension of the t_1-th GraphSAGE layer.

$$\mathbf{h}_{\ell,i}^{(t_1)} = \sigma\left(f_{node}\left(\mathbf{h}_{\ell,i}^{(t_1-1)}\right) + \frac{1}{|\mathcal{N}(\ell,i)|}\sum_{j\in\mathcal{N}(\ell,i)} f_{msg}\left(\mathbf{h}_{\ell,j}^{(t_1-1)}\right)\right) \quad (1)$$

$$= \sigma\left(W_1^{(t_1-1)}\cdot\mathbf{h}_{\ell,i}^{(t_1-1)} + \frac{1}{|\mathcal{N}(\ell,i)|}\sum_{j\in\mathcal{N}(\ell,i)} W_2^{(t_1-1)}\cdot\mathbf{h}_{\ell,j}^{(t_1-1)}\right) \quad (2)$$

where the subscripts of ℓ and i in either $u_{\ell,i}$ or $\mathbf{h}_{\ell,i}^{(\cdot)}$ denote the indices of all CFGs and their nodes, respectively; σ denotes the activation function; $\mathcal{N}(\ell,i)$ is the neighbor set of the node $u_{\ell,i}$ in $\mathcal{G}_{f_\ell}$. In MalGraph, we implement f_{node} and f_{msg} as neural networks with parameters of $W_1^{(t_1-1)}$ and $W_2^{(t_1-1)}$ to be learned. After a total of T_1 multiple layers, we obtain all node embeddings as $\{\mathbf{h}_{\ell,i}^{(T_1)} \in \mathbb{R}^{d_1}\}_{i=1}^{i=|\mathcal{G}_{f_\ell}|}$ for each graph $\mathcal{G}_{f_\ell}$ $(\ell=\{1,\cdots,n_L\})$, in which $d_1 = d^{(T_1)}$ is the output dimension of the last T_1-th GraphSAGE layer.

Then, to produce one graph-level representation vector per CFG (*i.e.*, $\mathcal{G}_{f_\ell}$ with $\ell=\{1,\cdots,n_L\}$), we directly apply the global pooling function that aggregates over the set of unordered node embeddings for each CFG as follows.

$$\mathbf{h}_{\mathcal{G}_{f_\ell}} = \text{max-pooling}\left\{\mathbf{h}_i^{(T_1)}\right\}_{i=1}^{i=|\mathcal{G}_{f_\ell}|} \in \mathbb{R}^{d_1} \quad (3)$$

Instead of employing some sophisticated and computation-intensive graph pooling models like [21], [22], we employ the simplest and straightforward max-pooling function, which has been proven to be one of the most effective and efficient graph pooling methods [19], [26]. Specifically, the max-pooling function directly calculates the maximum value for all node embeddings in one graph. In summary, for all local functions, we obtain a total of n_L graph-level vectors, *i.e.*, $\{\mathbf{h}_{\mathcal{G}_{f_\ell}}\}_{\ell=1}^{\ell=n_L}$.

2) Inter-function Graph Encoding Layer: Recall that an executable e can be represented as an FCG $\mathcal{G}_e$, in which nodes denote the functions and edges denote the caller-callee relationships between functions. For the FCG $\mathcal{G}_e$ with a total of n functions (*i.e.*, n_S external functions and n_L local functions), this layer aims to learn a graph-level embedding vector to represent the executable e at the inter-function level.

As introduced in § II-A, for each node v_e $(s=\{1,\cdots,n_S\})$ in $\mathcal{G}_e$ that represents an external function, we apply one-hot encoding based on the function name and further map it into a d_1-dimensional vector $\mathbf{h}_s \in \mathcal{R}^{d_1}$, which is equal to the dimension size of the graph-level representation vector in Eq. (3). On the other hand, for each node v_ℓ $(\ell=\{1,\cdots,n_L\})$ in $\mathcal{G}_e$ that represents the local function, we take the corresponding graph-level representation vector $\mathbf{h}_{\mathcal{G}_{f_\ell}} \in \mathcal{R}^{d_1}$ that has been computed by Eq. (3) in the intra-function graph encoding layer as the initial node embedding in $\mathcal{G}_e$.

With all nodes initialized in $\mathcal{G}_e$, we employ a similar model as the intra-function graph encoding layer to capture the structural information of how different functions interact and further pool it into a graph-level vector. To be exact, we first

apply multiple layers of GraphSAGE to iteratively update the hidden embeddings of all nodes in Eq. (4), obtaining the node embeddings of $\{\mathbf{z}_k^{(T_2)} \in \mathbb{R}^{d^{(T_2)}}\}_{k=1}^{k=n}$ $(n = n_S + n_L)$ after T_2 GraphSAGE layers. Then, to obtain a graph-level embedding vector $\mathbf{z}_{\mathcal{G}_e}$ for $\mathcal{G}_e$, we employ the max-pooling function over all node embeddings in Eq. (5) as follows.

$$\mathbf{z}_k^{(t_2)} = \sigma\Big(W_1^{(t_2-1)} \cdot \mathbf{z}_k^{(t_2-1)} + \frac{1}{|\mathcal{N}(k)|} \sum_{j \in \mathcal{N}(k)} W_2^{(t_2-1)} \cdot \mathbf{z}_j^{(t_2-1)}\Big) \quad (4)$$

$$\mathbf{z}_{\mathcal{G}_e} = \text{max-pooling}\left\{\mathbf{z}_k^{(T_2)}\right\}_{k=1}^{k=n} \in \mathbb{R}^{d_2} \quad (5)$$

where $\mathcal{N}(k)$ is the set of neighbors of the node v_k in the graph $\mathcal{G}_e$; $|\mathcal{N}(k)|$ denotes the size of $\mathcal{N}(k)$; $W_1^{(t_2-1)}$ and $W_2^{(t_2-1)}$ are learnable parameters in the (t_2-1)-th GraphSAGE. For simplicity, we denote $d_2 = d^{(T_2)}$ in which $d^{(T_2)}$ is the output dimension of the last T_2-th GraphSAGE layer.

3) Prediction Layer: After the graph-level embedding vector $\mathbf{z}_{\mathcal{G}_e}$ is obtained for representing the executable e, we compute the probability assigned to the malicious class (*i.e.*, p_e) by employing three MLPs as follows, which gradually project the dimension of the resulting vector down to a scalar of the dimension 1. As the expected malicious probability should be in the range of $[0, 1]$, we further perform the sigmoid function to enforce the malicious probability in this range.

$$p_e = \text{sigmoid}\left(\text{MLP}_{\theta 3}\big(\text{MLP}_{\theta 2}\big(\text{MLP}_{\theta 1}(\mathbf{z}_{\mathcal{G}_e})\big)\big)\right) \quad (6)$$

where $\text{MLP}_{\theta 1}$, $\text{MLP}_{\theta 2}$ and $\text{MLP}_{\theta 3}$ are MLPs to be trained.

Finally, suppose we have a threshold $\zeta \in [0, 1]$ which is the hyper-parameter for binary classifiers, we can predict the final label (*i.e.*, $\hat{y}_e$) for the given executable e as follows.

$$\hat{y}_e = \begin{cases} 1 & \text{if } p_e > \zeta\ , \\ 0 & \text{otherwise.} \end{cases} \quad (7)$$

where the label of 1 means it is a malware and 0 is a goodware.

B. Model Training with Mini-batch

Now, we present how to train `MalGraph` by minimizing the binary cross-entropy loss $\mathcal{L}$, which is defined as follows.

$$\mathcal{L} = -\frac{1}{N} \sum_{i=1}^{i=N} \big(y_{e_i} \cdot \log(p_{e_i}) + (1 - y_{e_i}) \cdot \log(1 - p_{e_i})\big) \quad (8)$$

where N is the total number of executables in the training dataset; y_{e_i} is the ground-truth label for the i-th executable e_i; p_{e_i} is the predicted malicious probability for e_i.

For improving the training efficiency on a large dataset, it is necessary to design the mini-batch propagation algorithm for `MalGraph`. However, recall that our proposed `MalGraph` model represents executable programs with hierarchical graphs, in which an executable program is represented with one FCG and some nodes of FCG (*i.e.*, local functions) are further represented with CFGs. Therefore, it is difficult to process over a mini-batch of executables simultaneously because different executables have different numbers of nodes (*i.e.*, different numbers of CFGs) in FCGs and different CFGs have different numbers of nodes as well. To tackle this challenge, we design the mini-batch processing algorithm for `MalGraph` whose pseudocode is presented in Algorithm 1.

Algorithm 1: Mini-batch processing of `MalGraph`.

Input : A batch of $\mathcal{B}$ executable programs $\{e_b\}_{b=1}^{b=\mathcal{B}}$ with corresponding FCGs $\{\mathcal{G}_{e_b}\}_{b=1}^{b=\mathcal{B}}$.
Output: A batch of malicious probabilities $\{p_{e_b}\}_{b=1}^{b=\mathcal{B}}$ for corresponding executable programs.

1 **Begin**
2 $\quad \mathbb{G}_e \leftarrow [\ ]$ ▷ initialize one big disconnected graph with all FCGs in the batch;
3 $\quad \mathbb{G}_f \leftarrow [\ ]$ ▷ initialize one big disconnected graph with all CFGs of all local functions in the batch;
4 $\quad$ **for** $b \leftarrow 1$ **to** $\mathcal{B}$ **do**
5 $\quad\quad \mathbb{G}_e \leftarrow \mathbb{G}_e \cup \mathcal{G}_{e_b}$ ▷ the b-th FCG $\mathcal{G}_{e_b}$;
6 $\quad\quad$ **for** $i \leftarrow 1$ **to** n_{L_b} **do**
7 $\quad\quad\quad \mathbb{G}_f \leftarrow \mathbb{G}_f \cup \mathcal{G}_{f_{b,i}}$ ▷ the i-th CFG $\mathcal{G}_{f_{b,i}}$;
8 $\quad\quad$ **end**
9 $\quad$ **end**
10 $\quad$ update node embedding of all CFGs in $\mathbb{G}_f$ with Eq. (1);
11 $\quad$ pool a set of graph-level representation vectors $\{\mathbf{h}_{\mathcal{G}_{f_{b,i}}}\}_{b=1,\, i=1}^{b=\mathcal{B},\, i=n_{L_b}}$ for all CFGs with Eq. (3);
12 $\quad$ leverage the obtained graph-level vector of local functions to initialize corresponding nodes in $\mathbb{G}_e$;
13 $\quad$ leverage the function names of external functions to initialize corresponding nodes in $\mathbb{G}_e$;
14 $\quad$ Update node embeddings of all FCGs in $\mathbb{G}_e$ with Eq. (4);
15 $\quad$ pool a set of graph-level representation vectors $\{\mathbf{z}_{\mathcal{G}_{e_b}}\}_{b=1}^{b=\mathcal{B}}$ for all FCGs with Eq. (5);
16 $\quad$ Compute a batch of malicious probabilities $\{p_{e_b}\}_{b=1}^{b=\mathcal{B}}$ with Eq. (6);
17 **return** $\{p_{e_b}\}_{b=1}^{b=\mathcal{B}}$.

IV. Experimental Setting & Implementation

A. Dataset Preparation

To evaluate the effectiveness and robustness of `MalGraph`, we prepare a mixed wild dataset that contains a large number of malwares and goodwares of PE files as follows.

1) Dataset Collection: For PE malwares, all raw data samples are collected from the Window suspicious executables in the academic malware sample repository provided by VirusTotal [27]. In the raw dataset provided by VirusTotal, almost every PE file has an associated scan file that contains multiple scan results from a range of 20 to 74 anti-malware products (*e.g.*, product name, detection result, malware family names, *etc*). It is well-known the scan results of different anti-malware products for the same suspicious file are inconsistent [28]–[30]. Thus, to avoid the bias induced by particular anti-malware products and ensure the dataset quality, we use a threshold of $\frac{2}{3}$ among all available products to decide whether a suspicious PE sample is malicious or not. Moreover, filtering malware samples with a threshold of $\frac{2}{3}$ is stricter than other prior work [31], [32] which only use fewer (*e.g.*, 2 or 5) particular anti-malware products for decision. For the collection of PE goodwares, as there is no dataset of goodwares available, we adopt the commonly used collection method in the literature [6], [33]–[35]. In particular, we deploy a virtual machine running a clean version of Windows 10 Pro and automatically install about 3,000 software packages

(*e.g.*, Chrome, Firefox, VS Code, 7-Zip, *etc*), which are the most commonly used by end-users and cover all categories of softwares in the 360 software manager [36]. After installation, we collect the resulting EXE/DLL files as the PE goodwares.

It is demonstrated that different proportions and types of packers employed in malwares or goodwares in the training dataset influence the effectiveness of the final malware detection model [37]. Therefore, to focus on evaluating the impact of the proposed hierarchical graph on the effectiveness and robustness of malware detection and exclude the influences of different packers employed in the dataset, for both malwares and goodwares collected, we first use standard unpacker tools (*i.e.*, UPX [38], PEiD [39], CAPE [40]) to unpack those packed PE files until they are no longer detected as packed [39], [41]. Besides, for a fair comparison with MalConv which forces the input PE files to 2MB in size, we also filter out those that are large than 2MB for both malwares and goodwares. Overall, we obtain a total of 227,197 PE samples, including 118,330 malwares and 108,867 goodwares.

2) FCG & CFG Generations: Similar with the preprocessing of Genius [11], we disassemble all PE samples in the dataset with IDA Pro 6.4 [42] and then generate their FCGs and CFGs accordingly. For each node representing the external function in FCG, it is one-hot encoded based on its function name and we limit the vocabulary size of external functions to 10,000 that are most frequently used in the training dataset. As introduced in § II-B, every local function in FCG is expanded with an associated CFG and each node inside the CFG is initialized with 11 statistics features. As for the number of nodes in FCG and the summary number of nodes for all CFGs follow the typical long-tail distribution, we limit the number of nodes in the FCG to 3,000 and limit the total number of nodes for all CFGs to 10,000, resulting in a total of 183,028 samples and keep more than 80% of the original dataset.

Furthermore, to evaluate the effectiveness of `MalGraph` in detecting new and previously unseen malwares (*i.e.*, zero-day), we employ a time-based train/test split. In particular, we take all malware samples collected before May 6, 2020 as the training set and equally divide the remaining malwares into the validation and testing set. For goodwares, we randomly split them according to the percentages of the training/validation/testing sets in malwares. Table II shows the dataset statistics.

TABLE II
SUMMARY OF DATASET STATISTICS.

Dataset	# Training	# Validation	# Testing	**# Total**
Malwares	79,004	13,773	13,774	106,551
Goodwares	56,592	9,942	9,943	76,477
Total	135,596	23,715	23,717	183,028

B. Baseline Models

To evaluate the effectiveness and robustness of `MalGraph`, we consider three state-of-the-art baselines for comparisons.

1) EMBER [5] is the state-of-the-art ML-based malware detection model which trains a GBDT model based on more than 2,000 kinds of hand-crafted features.
2) MalConv [6] is the representative DL-based malware detection model which trains a CNN-based model based on the raw bytes of executables.
3) MAGIC [43] is the most relevant malware detection to our `MalGraph` model. MAGIC trains a GNN-based model based on only CFGs of executables.

C. Attack Method

To evaluate the robustness of malware detection models in the real adversarial scenario, we consider the black-box attack to generate realistic adversarial malwares rather than adversarial features. As there is no practical black-box attack universally for `MalGraph` and baselines, we adjust the baseline evasion attack (MLSEC-Attack [44]) in the 2020 machine learning security evasion competition hosted by Microsoft Azure. In particular, MLSEC-Attack first considers different kinds of functionality-preserving modifications for PE files as the search space and then applies the tree-structured parzen estimator algorithm [45] for the black-box optimization.

D. Evaluation Metrics

1) Detection Metric: To measure the overall detection performance of malware detection models, we compute their **AUC** scores since AUC is independent of the manually selected threshold ζ. Specifically, by plotting the true positive rate (TPR) against the false positive rate (FPR) at various thresholds, the ROC curve is drawn and AUC is defined as the area under the curve enclosed with the x-axis [46]. Apparently, the larger AUC, the better performance of the detection model.

In addition to AUC, it is important to measure standard detection metrics, *i.e.*, **TPR** and balanced accuracy (**bACC**), at the same level of FPR. For bACC, it is a commonly used metric to evaluate performance on imbalanced datasets. To be specific, for each detection model, we try to compute the threshold such that the model has an FPR of 1% or 0.1%, and then compute the corresponding TPR and bACC as follows.

$$
\begin{aligned}
TPR &= \frac{TP}{TP+FN} \qquad FPR = \frac{FP}{FP+TN} \\
bACC &= \frac{TPR+TNR}{2} = \frac{\left(\frac{TP}{TP+FN}\right)+\left(\frac{TN}{FP+TN}\right)}{2}
\end{aligned}
\tag{9}
$$

where TP, TN, FP, and FN represent the numbers of true positive, true negative, false positive, and false negative, respectively. With the same level of FPR (1% or 0.1%), a higher value of TPR or bACC reflects better detection performance.

2) Robustness Metric: To evaluate the robustness of detection models, we use the metric of attack success rate (ASR), which is the most widely used metric in evaluating the performance of adversarial attacks [47], [48]. For real-world anti-malware products, misclassifying the malwares as goodwares is meaningful, but not vice versa. Therefore, in the scenario of malware detection models, we consider a malware sample that is originally correctly classified as malicious but misclassified as benign after performing the adversarial attack. In particular, for $\forall\, e \in \mathbb{E}$, ASR is defined as follows and a lower value of ASR indicates a more robust detection model.

$$\text{ASR} = \frac{\#\,\text{success malwares}}{\#\,\text{total malwares}} = \frac{|\Gamma(e)=1 \wedge \Gamma(e^*)=0|}{|\Gamma(e)=1|} \quad (10)$$

where Γ denotes the target detection model that outputs the label; $\mathbb{E}$ denotes all candidate malwares to be attacked; e is an original malware and e^* is the generated adversarial malware; $|\cdot|$ denotes the number of counts that meet the condition.

E. Implementation Details

Our implementation is built on PyTorch [49] and PyTorch_Geometric [50]. By default, we use a two-layer GraphSAGE (*i.e.*, T_1=T_2=2) in both the intra-function and inter-function graph encoding layer and each of the GraphSAGE models has an output dimension of 200. After each layer of GraphSAGE, we use ReLU as the activation function along with a dropout layer with the dropout rate being 0.2. To train the model, we set the batch size to 16 and use the AdamW optimizer [51] with a learning rate of 0.001 as well as a weight decay of 0.00001. We train the model for 10 epochs and select the best model based on the lowest validation loss. In general, it takes approximately 10 hours to train, validate and test the model. It is noted that we conducted all experiments on a PC with two Intel Xeon E5-2640 CPUs running at 2.40GHz, 64 GB memory, and 3 GeForce GTX 1080 Ti GPU cards.

V. Experimental Results

A. Evaluation of Effectiveness

To demonstrate the effectiveness of `MalGraph` and compare it with baseline models, we systematically and quantitatively evaluate them by measuring all detection metrics (*i.e.*, AUC, TPR, and bACC) on the testing dataset. Table III presents the summarized evaluation results of `MalGraph`. Recall that for fair comparisons, we compare both TPR and bACC at the same level of FPRs, including 1% and 0.1% FPR. For any malware detection, it is desired that TPR and bACC should be as high as possible at the same level of FPRs. It is also noted that we trained two detection models of MalConv with an almost identical model architecture but a different final prediction layer. In particular, MalConv-1 follows the same prediction layer with `MalGraph` that outputs the 1-dimension malicious probability with sigmoid, while MalConv-2 follows the original model setting in [6] that outputs the 2-dimension malicious/benign probability with the softmax function.

Among all baseline methods, it is clearly observed from Table III that EMBER offers superior performance on the testing dataset, while both MalConv and MAGIC show unstable and inferior performance. In particular, both MalConv-1 and MAGIC can achieve competitive performance with more than 98% of AUC and over 90% TPR/bACC at 1% FPR. However, their performance of TPR and bACC at 0.1% FPR cannot even be calculated because both of them cannot find the exact threshold that satisfies 0.1% FPR, indicating their extremely poor detection performance at 0.1% FPR. For MalConv-2, it can calculate the TPR/bACC at 0.1% FPR, but all its detection metrics reported are significantly inferior to other models.

Compared with all baselines, our proposed `MalGraph` model achieves the state-of-the-art performance in terms of all the five detection metrics. Specifically, `MalGraph` shows the best and stable performance with more than 99% of AUC and TPR/bACC at 1% FPR. Even for the challenging case for TPR and bACC at 0.1%, `MalGraph` has significantly higher performance than the best results of all baseline models by a large margin up to 12.92 and 6.46 absolute value on TPR and bACC, respectively. These observations suggest that the proposed `MalGraph` model is more like to detect as many malwares as possible on the testing dataset while allowing for an extremely low FPR, *i.e.* 0.1%.

TABLE III
DETECTION PERFORMANCE OF `MalGraph` COMPARED WITH BASELINES.

Models	AUC (%)	FPR = 1%		FPR = 0.1%	
		TPR (%)	bACC (%)	TPR (%)	bACC (%)
EMBER	99.89	99.35	99.18	83.53	91.72
MalConv-1	99.90	99.13	99.06	-	-
MalConv-2	96.07	14.68	56.84	2.24	51.07
MAGIC	98.48	90.81	94.91	-	-
`MalGraph`	**99.97**	**99.46**	**99.23**	**96.45**	**98.18**

B. Evaluation of Robustness

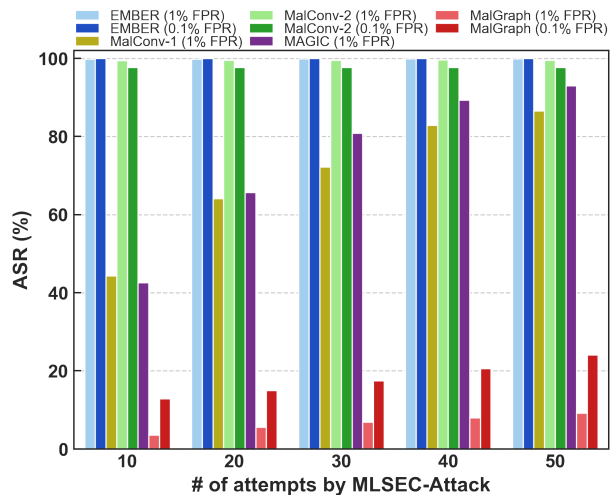


Fig. 3. Robustness performance of `MalGraph` compared with baselines w.r.t different number of attempts M by MLSEC-Attack.

We evaluate the robustness of `MalGraph` and baseline models against the problem-space adversarial attack under black-box settings. In particular, we randomly select 1,000 (if applicable) malware samples that are correctly classified by the corresponding detection models. Then, for each selected sample, MLSEC-Attack is performed with a maximum of attempts M until a successful adversarial malware is generated. In our evaluation, we change M from 10, 20, 30, 40 to 50 and report their robustness performance in Fig.3, in which a lower value of ASR means a more robust model.

Among all baseline methods, it is observed that all of them have extremely high ASRs in both cases of FPRs. First of all,

for EMBER and MalConv-2 at either 1% or 0.1% FPR, all of them obtain nearly 100% ASRs w.r.t different number of attempts by MLSEC-Attack, indicating they are highly vulnerable to the generate adversarial malwares. Secondly, with the increase of a maximum of attempts M, ASR of MalConv-1 (1% FPR) gradually increases from 44.3% to 86.5%, and ASR of MAGIC (1% FPR) also gradually increases from 42.6% to 93.0%. This is evident that increasing the maximum of attempts M will increase the attack ability of MLSEC-Attack, and thus results in an increased ASR for detection models.

Compared with all baselines, it can be seen that our proposed `MalGraph` model in both cases of FPRs offers the superior robustness performance with extremely low ASRs, *i.e.*, much lower than all baseline models. To be exact, with the increase of M from 10 to 50, the ASR of `MalGraph` (1% FPR) gradually increases from 3.6% to 9.2%, and the ASR of `MalGraph` (0.1% FPR) also gradually increases from 12.9% to 24.1%. We conjecture that it is the hierarchical nature makes our hierarchical graph to limit the modifications caused by MALSEC-Attack to a certain small range, thereby reducing the impact on the results of our proposed `MalGraph` detection.

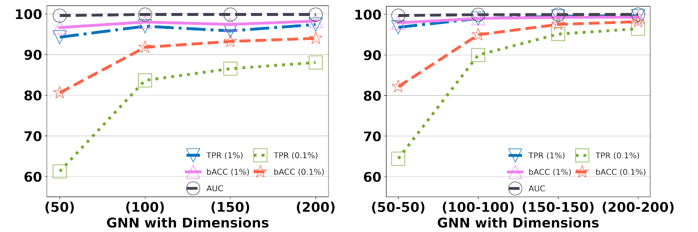
C. Ablation Studies

1) Effect of the hierarchical graph representation: To show the effectiveness of the hierarchical graph representation that incorporates the inter-function FCG with intra-function CFGs in `MalGraph`, we have conducted experiments over two model variants with either one FCG or all CFGs as the input. In particular, `MalGraph`-FCG only replaces the intra-function graph encoding layer with the initialized FCG, in which internal functions are encoded with the accumulated statistics in the assembly instructions and external functions are encoded with the same one-hot encoding based on function names. `MalGraph`-CFGs first uses the same intra-function graph encoding layer and then replaces the inter-function graph encoding layer with the max-pooling function among all graph-level vectors of CFGs. From Table IV, we can observe that `MalGraph`-CFGs shows better performance of AUC and TPR/bACC at 1% FPR, while `MalGraph`-FCG shows better performance of TPR/bACC at 0.1% FPR. However, our full model – `MalGraph`-Full clearly achieves noticeably best performance. These observations highlight the importance of the hierarchical graph that incorporates FCG with CFGs, which could significantly improve the effectiveness.

TABLE IV
EFFECT OF THE HIERARCHICAL GRAPH REPRESENTATION.

Models	AUC (%)	FPR = 1%		FPR = 0.1%	
		TPR (%)	bACC (%)	TPR (%)	bACC (%)
`MalGraph`-FCG	99.08	87.40	93.27	60.28	80.09
`MalGraph`-CFGs	99.71	94.77	96.89	58.78	79.35
`MalGraph`-Full	**99.97**	**99.46**	**99.23**	**96.45**	**98.18**

2) Impact of different dimensions in GraphSAGE: We further study how the GNN dimensions would affect the performance of `MalGraph`. Following the default parameter settings like previous experiments, we only change the layer numbers (*i.e.*, one-layer v.s. two-layer) and feature dimensions (*i.e.*, 50, 100, 150, and 200) of GraphSAGE. To be exact, Fig. 4(a) and Fig. 4(b) depict the performance impacts of `MalGraph` with one-layer GNN (*i.e.*, T_1=T_2=1) and two-layer GNN (*i.e.*, T_1=T_2=2), respectively. It is clearly observed that the performance of `MalGraph` with the same number of GNN layers improves as the feature dimensions of GNN grow and `MalGraph` with more GNN layers offers better performance. The reason is evident that increasing the number of GNN layers or feature dimensions would increase the model capacity and thus improve the learning ability of `MalGraph` in supervised learning. In addition, the overall performance of `MalGraph` with two-layer GNN is relatively stable when the feature dimension reaches 200. Therefore, to make a trade-off between model performance and resource consumption, we take `MalGraph` that contains a two-layer GNN with each layer having a feature dimension of 200 as the default model.



(a) The model with one-layer GNN. (b) The model with two-layer GNN.

Fig. 4. Detection performance of `MalGraph` with various GNN dimensions.

3) Impact of `MalGraph` with different GNNs: We also investigate the impact of different GNNs employed in both the intra-function graph encoding layer and the inter-function graph encoding layer. To be specific, we replace the employed GraphSAGE with two GNN variants, *i.e.*, GCN [16] and TransConv [52]. It is noted that we do not fine-tune any hyper-parameters and all other experimental settings of `MalGraph`-GCN and `MalGraph`-TransConv are kept the same with `MalGraph` as the previous evaluation. In general, it can be seen from Table V that all two variant models achieve similar and stable performance in terms of all detection metrics, implying our model architecture is not sensitive to the choice of GNN variants. Interestingly, compared with our default `MalGraph`, `MalGraph`-GCN performs even better of TPR/bACC at 1% FPR, which indicates our model can be further improved by carefully choosing more advanced GNN models according to different applications in the wild.

TABLE V
IMPACT OF `MALGRAPH` WITH DIFFERENT GNN VARIANTS.

Models	AUC (%)	FPR = 1%		FPR = 0.1%	
		TPR (%)	bACC (%)	TPR (%)	bACC (%)
`MalGraph`-GCN	99.49	**99.71**	**99.35**	92.09	96.00
`MalGraph`-TransConv	99.93	99.35	99.18	95.81	97.86
`MalGraph`	**99.97**	99.46	99.23	**96.45**	**98.18**

VI. Related Work

A. ML/DL for PE Malware Detection

To improve the performance for Windows PE malware detection, extensive research efforts in leveraging the ability of ML models have been ignited since [2], which trains varied ML models based on various sets of features, including system call functions, strings, and hexadecimal byte sequences. After that, almost all follow-up ML-based detection methods [3]–[5] are based on extending the framework of [2] with two main aspects for improvements [53]. The first aspect is to devise more useful static features to represent PE malwares, including PE header statistics, byte sequences, opcodes, API calls, *etc*. The second is to explore more advanced ML models, such as SVM, random forest, *etc*. More recently, to avoid using hand-crafted features devised by domain experts, neural networks (*e.g.*, MLP, CNN, RNN, and GNN) are increasingly being used for PE malware detection based on the corresponding representations, including raw bytes [6], binary grayscale images [7], n-gram byte sequences [8], CFGs [43], *etc*.

Our work differs from prior work in two main aspects: 1) Unlike prior work that ignores the rich semantic information of executables, `MalGraph` can capture the intrinsic properties of malwares with the hierarchical graph, which has been demonstrated to be more effective in detecting PE malwares in this work. 2) It is the first one to evaluate the robustness performance on the task of PE malware detection against existing adversarial attacks like MLSEC-Attack. In particular, our evaluation suggests `MalGraph` that is built on hierarchical graphs is the most robust malware detection than all baseline detection with other representations, including hand-crafted features [5], raw bytes [6] and "non-hierarchical" CFGs [43].

B. Adversarial Attacks

It has been shown that existing ML/DL models are inherently vulnerable to adversarial attacks with crafted adversarial samples, which are maliciously crafted inputs to misbehave the given target model. Initially, researches on adversarial attacks are explored in the context of images and attempt to apply a small (*i.e.*, virtually imperceptible to human perception) perturbation on a given input image to misclassify the given target model. Furthermore, adversarial attacks are also being investigated in many other areas [54] like NLP, RL and GNNs.

To facilitate adversarial attacks against PE malware detection models, research efforts have begun to be gradually devoted to automatically generating *adversarial malwares* such that they are no longer detected as malwares while not breaking the format and functionality. However, most existing adversarial attacks against malware detection are unpractical and ineffective [53], as they either assume an unrealistic white-box adversarial attack [55]–[58] or generate *adversarial feature vectors* (*e.g.*, API calls [55], [56], grayscale images [59], *etc*) instead of realistic adversarial malwares due to the extreme difficulty of inverse feature-mapping [60]. Gym-malware [61] is the first black-box adversarial attack that generates adversarial malwares based on RL with functionality-preserving modifications for PE files. Although gym-malware has shown its effectiveness against malware detection models, its own experiments also show that gym-malware based on RL offers very limited improvement compared with the random policy. MLSEC-Attack [44] is the baseline black-box adversarial attack in 2020 machine learning security evasion competition hosted by Microsoft Azure. In particular, MLSEC-Attack only replaces the RL framework with the tree-structured parzen estimator algorithm [45] for the black-box optimization.

VII. Conclusion and Future Work

In this paper, we propose `MalGraph`, a hierarchical graph neural network to build an effective and robust Windows PE malware detection. In particular, `MalGraph` makes better use of the hierarchical graph representation which incorporates the inter-function call graph with intra-function control flow graphs for representing the executable program. Extensive experiments demonstrate that `MalGraph` significantly outperforms state-of-the-art baseline models by a large margin in terms of both effectiveness and robustness. Although our proposed `MalGraph` model focuses on PE malwares detection on the Windows platform, we argue that such a model could potentially be used as future work for other malware types, such as Linux malwares, Android malwares, PDF malwares, *etc*. Another future work is to devise more advanced and specialized adversarial attacks, which can maliciously manipulate the internal logic of malwares (*e.g.*, function call graph, control-flow graph) in a more subtle and fine-grained manner, so as to further evaluate the robustness of malware detection.

Acknowledgment

We would like to thank VirusTotal for providing the academic malware sample repository. This work was partly supported by the National Key R&D Program of China under No. 2020YFB1804705; the Major Research Plan of National Natural Science Foundation of China under No. 92167203; the National Natural Science Foundation of China under No. 61772507 and No. U1936215; the Science and Technology Development Funds of China under No. 2021ZY1025; the Key R&D Program of Zhejiang Province under No. 2021C01036 and No. 2022C01243; the Zhejiang Provincial Natural Science Foundation for Distinguished Young Scholars under No. LR19F020003; the State Key Laboratory of Computer Architecture (ICT, CAS) under Grant No. CARCHA202001; and the Fundamental Research Funds for the Central Universities (Zhejiang University NGICS Platform).

References

[1] The AV-TEST Institute, "Malware statistics," https://www.av-test.org/en/statistics/malware/, 2021, Online (last accessed Jan. 10, 2021).

[2] M. G. Schultz, E. Eskin, F. Zadok, and S. J. Stolfo, "Data mining methods for detection of new malicious executables," in *S&P*, 2000.

[3] J. Z. Kolter and M. A. Maloof, "Learning to detect malicious executables in the wild," in *KDD*, 2004.

[4] M. Z. Shafiq, S. M. Tabish, F. Mirza, and M. Farooq, "A framework for efficient mining of structural information to detect zero-day malicious portable executables," *1Next Generation Intelligent Networks Research Center (nexGIN RC), Tech. Rep*, 2009.

[5] H. S. Anderson and P. Roth, “EMBER: an open dataset for training static PE malware machine learning models,” *arXiv:1804.04637*, 2018.
[6] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. Nicholas, “Malware detection by eating a whole exe,” *arXiv: 1710.09435*, 2017.
[7] K. He and D.-S. Kim, “Malware detection with malware images using deep learning techniques,” in *TrustCom/BigDataSE*, 2019.
[8] R. Lu, “Malware detection with LSTM using opcode language,” *arXiv: 1906.04593*, 2019.
[9] B. G. Ryder, “Constructing the call graph of a program,” *TSE*, vol. 5, no. 3, pp. 216–226, 1979.
[10] X. Hu, T.-c. Chiueh, and K. G. Shin, “Large-scale malware indexing using function-call graphs,” in *CCS*, 2009.
[11] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, “Scalable graph-based bug search for firmware images,” in *CCS*, 2016.
[12] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, “discovRE: Efficient cross-architecture identification of bugs in binary code,” in *NDSS*, 2016.
[13] 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 *CCS*, 2017.
[14] L. Massarelli, G. A. Di Luna, F. Petroni, R. Baldoni, and L. Querzoni, “SAFE: Self-attentive function embeddings for binary similarity,” in *DIMVA*, 2019.
[15] X. Ling, L. Wu, S. Wang, T. Ma, F. Xu, A. X. Liu, C. Wu, and S. Ji, “Multilevel graph matching networks for deep graph similarity learning,” *IEEE Transactions on Neural Networks and Learning Systems*, 2021.
[16] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” in *ICLR*, 2017.
[17] Y. Rong, T. Xu, J. Huang, W. Huang, H. Cheng, Y. Ma, Y. Wang, T. Derr, L. Wu, and T. Ma, “Deep graph learning: Foundations, advances and applications,” in *KDD*, 2020.
[18] X. Ling, L. Wu, C. Wu, and S. Ji, “Graph neural networks: Graph matching,” in *Graph Neural Networks: Foundations, Frontiers, and Applications*, L. Wu, P. Cui, J. Pei, and L. Zhao, Eds. Singapore: Springer, 2022, ch. 13, pp. 277–295.
[19] W. Hamilton, Z. Ying, and J. Leskovec, “Inductive representation learning on large graphs,” in *NIPS*, 2017.
[20] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, “Graph attention networks,” in *ICLR*, 2018.
[21] Z. Ying, J. You, C. Morris, X. Ren, W. Hamilton, and J. Leskovec, “Hierarchical graph representation learning with differentiable pooling,” in *NIPS*, 2018.
[22] H. Gao and S. Ji, “Graph u-nets,” in *ICML*, 2019.
[23] M. Simonovsky and N. Komodakis, “GraphVAE: Towards generation of small graphs using variational autoencoders,” in *ICANN*, 2018.
[24] B. Samanta, A. De, N. Ganguly, and M. Gomez-Rodriguez, “Designing random graph models using variational autoencoders with applications to chemical design,” *arXiv:1802.05283*, 2018.
[25] Y. Bai, H. Ding, S. Bian, T. Chen, Y. Sun, and W. Wang, “Simgnn: A neural network approach to fast graph similarity computation,” in *WSDM*, 2019.
[26] X. Ling, L. Wu, S. Wang, G. Pan, T. Ma, F. Xu, A. X. Liu, C. Wu, and S. Ji, “Deep graph matching and searching for semantic code retrieval,” *ACM Transactions on Knowledge Discovery from Data*, vol. 15, no. 5, 2021.
[27] VirusTotal.com, https://www.virustotal.com/gui/contact-us, 2020, Online (last accessed Aug. 1, 2020).
[28] A. Mohaisen and O. Alrawi, “AV-Meter: An evaluation of antivirus scans and labels,” in *DIMVA*, 2014.
[29] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero, “AVCLASS: A tool for massive malware labeling,” in *RAID*, 2016.
[30] S. Sebastián and J. Caballero, “AVClASS2: Massive malware tag extraction from AV labels,” in *ACSAC*, 2020.
[31] N. Nissim, R. Moskovitch, L. Rokach, and Y. Elovici, “Novel active learning methods for enhanced PC malware detection in Windows OS,” *Expert Systems with Applications*, vol. 41, no. 13, pp. 5843–5857, 2014.
[32] P. Laskov, “Detection of malicious pdf files based on hierarchical document structure,” in *NDSS*, 2013.
[33] F. Kreuk, A. Barak, S. Aviv-Reuven, M. Baruch, B. Pinkas, and J. Keshet, “Deceiving end-to-end deep learning malware detectors using adversarial examples,” *arXiv:1802.04528*, 2018.
[34] X. Li, K. Qiu, C. Qian, and G. Zhao, “An adversarial machine learning method based on opcode n-grams feature in malware detection,” in *DSC*, 2020.
[35] Y. Junkun, Z. Shaofang, L. Lanfen, W. Feng, and C. Jia, “Black-box adversarial attacks against deep learning based malware binaries detection with GAN,” in *ECAI*, 2020.
[36] 360 Total Security, https://www.360totalsecurity.com/, 2020, Online (last accessed Aug. 1, 2020).
[37] H. Aghakhani, F. Gritti, F. Mecca, M. Lindorfer, S. Ortolani, D. Balzarotti, G. Vigna, and C. Kruegel, “When malware is packin’heat; limits of machine learning classifiers based on static analysis features,” in *NDSS*, 2020.
[38] The UPX Team, “Upx: The ultimate packer for executables,” https://upx.github.io/, 2020, Online (last accessed Dec. 15, 2020).
[39] Jibz and Qwerton and snaker and xineohP, “PEiD: PE iDentifier,” https://www.aldeid.com/wiki/PEiD, 2020, Online (last accessed Dec. 15, 2020).
[40] Kevin O’Reilly, “CAPE: Malware Configuration And Payload Extraction,” https://github.com/kevoreilly/CAPEv2, 2020, Online (last accessed Dec. 15, 2020).
[41] VirusTotal, “YARA in a nutshell,” https://github.com/virustotal/yara, 2020, Online (last accessed Dec. 15, 2020).
[42] Hex-Rays, “IDA Pro,” https://hex-rays.com/ida-pro/, 2021, Online (last accessed Jan. 17, 2020).
[43] J. Yan, G. Yan, and D. Jin, “Classifying malware represented as control flow graphs using deep graph convolutional neural network,” in *DSN*, 2019.
[44] Microsoft Azure, “2020 machine learning security evasion competition,” https://mlsec.io/, 2021, Online (last accessed Jan. 20, 2021).
[45] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, “Algorithms for hyper-parameter optimization,” in *NIPS*, 2011.
[46] A. P. Bradley, “The use of the area under the ROC curve in the evaluation of machine learning algorithms,” *Pattern recognition*, vol. 30, no. 7, pp. 1145–1159, 1997.
[47] X. Ling, S. Ji, J. Zou, J. Wang, C. Wu, B. Li, and T. Wang, “DEEPSEC: A uniform platform for security analysis of deep learning model,” in *IEEE Symposium on Security and Privacy*, 2019.
[48] J. Li, T. Du, S. Ji, R. Zhang, Q. Lu, M. Yang, and T. Wang, “Textshield: Robust text classification based on multimodal embedding and neural machine translation,” in *USENIX Security*, 2020.
[49] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, “Automatic differentiation in pytorch,” in *NIPS Autodiff Workshop*, 2017.
[50] M. Fey and J. E. Lenssen, “Fast graph representation learning with PyTorch Geometric,” in *ICLR Workshop*, 2019.
[51] I. Loshchilov and F. Hutter, “Decoupled weight decay regularization,” in *ICLR*, 2019.
[52] Y. Shi, Z. Huang, S. Feng, and Y. Sun, “Masked label prediction: Unified massage passing model for semi-supervised classification,” *arXiv:2009.03509*, 2020.
[53] X. Ling, L. Wu, J. Zhang, Z. Qu, W. Deng, X. Chen, C. Wu, S. Ji, T. Luo, J. Wu, and Y. Wu, “Adversarial attacks against Windows PE malware detection: A survey of the state-of-the-art,” *arXiv preprint arXiv:2112.12310*, 2021.
[54] H. Xu, Y. Ma, H.-C. Liu, D. Deb, H. Liu, J.-L. Tang, and A. K. Jain, “Adversarial attacks and defenses in images, graphs and text: A review,” *IJAC*, vol. 17, no. 2, pp. 151–178, 2020.
[55] A. Al-Dujaili, A. Huang, E. Hemberg, and U.-M. O’Reilly, “Adversarial deep learning for robust detection of binary encoded malware,” in *IEEE Security and Privacy Workshops*, 2018.
[56] S. Verwer, A. Nadeem, C. Hammerschmidt, L. Bliek, A. Al-Dujaili, and U.-M. O’Reilly, “The robust malware detection challenge and greedy random accelerated multi-bit search,” in *AISec*, 2020.
[57] B. Kolosnjaji, A. Demontis, B. Biggio, D. Maiorca, G. Giacinto, C. Eckert, and F. Roli, “Adversarial malware binaries: Evading deep learning for malware detection in executables,” in *EUSIPCO*, 2018.
[58] L. Demetrio, S. E. Coull, B. Biggio, G. Lagorio, A. Armando, and F. Roli, “Adversarial EXEmples: A survey and experimental evaluation of practical attacks on machine learning for windows malware detection,” *arXiv:2008.07125*, 2020.
[59] B. Chen, Z. Ren, C. Yu, I. Hussain, and J. Liu, “Adversarial examples for cnn-based malware detectors,” *IEEE Access*, vol. 7, 2019.
[60] F. Pierazzi, F. Pendlebury, J. Cortellazzi, and L. Cavallaro, “Intriguing properties of adversarial ml attacks in the problem space,” in *S&P*, 2020.
[61] H. S. Anderson, A. Kharkar, B. Filar, and P. Roth, “Evading machine learning malware detection,” in *Black Hat USA*, 2017.