2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)

Classifying Malware Represented as Control Flow Graphs using Deep Graph Convolutional Neural Network

Jiaqi Yan
Illinois Institute of Technology
jyan31@hawk.iit.edu

Guanhua Yan
Binghamton University, State University of New York
ghyan@binghamton.edu

Dong Jin
Illinois Institute of Technology
dong.jin@iit.edu

***Abstract*—Malware have been one of the biggest cyber threats in the digital world for a long time. Existing machine learning-based malware classification methods rely on handcrafted features extracted from raw binary files or disassembled code. The diversity of such features created has made it hard to build generic malware classification systems that work effectively across different operational environments. To strike a balance between generality and performance, we explore new machine learning techniques to classify malware programs represented as their control flow graphs (CFGs). To overcome the drawbacks of existing malware analysis methods using inefficient and non-adaptive graph matching techniques, in this work, we build a new system that uses deep graph convolutional neural network to embed structural information inherent in CFGs for effective yet efficient malware classification. We use two large independent datasets that contain more than 20K malware samples to evaluate our proposed system and the experimental results show that it can classify CFG-represented malware programs with performance comparable to those of the state-of-the-art methods applied on handcrafted malware features.**

***Index Terms*—malware classification, control flow graph, deep learning, graph convolution**

I. Introduction

While many anti-virus vendors and computer security researchers have fought hard against malicious software for many years, they remain one of the biggest digital threats in the cyberworld. Motivated by the high return on investment ratio, the underground malware industry has been consistently enlarging the sheer volume of the threats on the Internet every year. According to a report by AV-TEST [1], the number of total malware by 2018 is estimated at over 800 million, which has increased 28 times over the past 10 years. Even worse, recently viciously keen cybercriminals made a lot of efforts to diversify their avenues of attacks [2]. Inspired by the astonishing rise in crypto currency values, 47 new coin mining malware families have emerged since early 2017 and was ascribed to the 956% soar of the number of crypto mining attacks in the past year [3]. Traditional signature-based detection approaches have failed to thwart the ever-evolving malware threats. Therefore efficient, robust and scalable anti-malware solutions are indispensable for protecting the trustworthiness of the modern cyberworld.

A number of recent research efforts [4]–[15] have shown that machine learning (ML) offers a promising approach to thwart the voluminous malware attacks. These efforts have been inspired by the great success of ML in improving the accuracy of image classification, language translation, and many other applications. There is however a dilemma when we investigate an effective ML-based malware classification system. On one hand, if we focus too much on finding discriminative malware features to achieve high classification accuracy, the features constructed as such may lose their generality when a different operational environment is encountered. For instance, although features extracted from the PE headers of malware files have been shown useful in classifying PE malware variants belonging to different families [16], a malware classification system trained with these features cannot be used to detect those fileless malware that only exist in the memory. On the other hand, if the features extracted from malware programs are too generic, such as the frequencies of n-gram byte sequences [4], it is difficult to train a malware classifier with high accuracy from them due to their lack of discriminative power [16].

To strike a balance between generality and performance, we aim to build a malware classification system from malware programs represented as their control flow graphs (CFGs), a data structure commonly used to characterize the control flow of any computer program. The generality of CFGs for malware classification can be attributed to two factors: (1) the CFG can be extracted from different formats of malware code, such as binary executable files, exploit code discovered in network traffic [17], emulated malware [18], and attack code chained together from gadgets in return-oriented programming attacks [19], and (2) the CFG can be used to derive a variety of static analysis features widely used in existing works on ML-based malware classification, such as n-grams [4], q-grams [5], opcodes [20] and structural information [21]. Therefore, a malware classification system trained from CFG-represented malware programs can find applications in various operational environments.

Classifying CFG-represented malware programs needs to address two types of performance issues, classification performance (*does the malware classifier achieve high accuracy?*) and execution performance (*can the malware classifier work efficiently in practice?*). As discussed earlier, reducing CFGs to vectors that contain simple aggregate features such as n-

978-1-7281-0057-9/19/$31.00 ©2019 IEEE
DOI 10.1109/DSN.2019.00020

grams and opcodes leads to efficient malware classification but usually with poor classification accuracy. Due to the graph nature of CFGs, previous works have studied use of graph similarity measures to train malware classification models [21]–[23]. However, some techniques for calculating pairwise graph similarity such as those based on graph matching or isomorphism can be computationally prohibitive, letting alone that the time needed to compute pairwise graph similarity for a malware dataset scales quadratically with its size.

Against this backdrop, in this work we propose to use a state-of-the-art deep learning infrastructure, graph kernel-based deep neural network, to classify malware programs represented as control flow graphs. Due to their capability of understanding complex graph data, graph kernel-based deep neural network have found success in a number of application domains, such as protein classification, chemical toxicology prediction, and cancer detection [24]–[27]. Particularly, our work extends a special type of graph kernel-based deep neural network, Deep Graph Convolutional Neural Network (DGCNN) [27] for classifying CFG-represented malware programs. Different from graph classification techniques based on pairwise graph similarity, DGCNN allows attribute information associated with individual vertices to be aggregated quickly through neighborhood defined by the graph structure in breadth-first-search fashion, thus embedding high-dimensional structural information into vectors that are amenable to efficient classification.

To demonstrate the applicability of DGCNN for malware classification, we have developed a new system called MAGIC (an end-to-end malware defense system that classifies CFG-represented malware programs using DGCNN). MAGIC improves the effectiveness of malware classification by extending the standard DGCNN with customized techniques tailored for malware classification. We use two large malware datasets to evaluate the performance of MAGIC and the experimental results show that it can classify CFG-represented malware programs with accuracy comparable to those of the state-of-the-art methods applied on handcrafted malware features.

The remainder of the paper is organized as follows. In Section II, we introduce the overall design of MAGIC. In Section III, we provide a primer on DGCNN and then present our improvements over the standard DGCNN for classifying CFG-represented malware programs. In Section IV we discuss a few implementation details of MAGIC. We present our experimental results in Section V. We discuss related work in Section VI and draw concluding remarks in Section VII.

II. System Overview of MAGIC

This section overviews the three main components of MAGIC, whose workflow is illustrated in Figure 1.

A. Control Flow Graph

MAGIC relies on the state-of-the-art tools, such as IDA Pro [28], to extract CFGs from malware code. In a CFG, a vertex represents a basic block, which contains a straight sequence of code or assembly instructions without any control flow transition except at its exit. Two vertices (u, v) are connected by a directed edge $u \rightarrow v$ if either the last instruction in u falls through the first line of code in v, or there is a jump instruction in u that is destined to some instructions (e.g., jump target) in v. The implementation details on how to build the CFG from disassembled execution code will be given in Section IV-A.

B. Attributed Control Flow Graph

The CFG representation of software program is generic for the purpose of malware classification in several ways. First, this type of representation transcends specific programming languages in which the programs are written or hardware platforms for which the programs are developed. Although other low-level representations such as hexadecimal byte sequences have similar properties, a CFG explicitly expresses the execution logic of a program using a graph data structure. Hence, the semantics of a malware program is embodied by not only the characteristics of the code in individual basic blocks but also their structural dependencies defined by the edges connecting these basic blocks.

To convert CFGs to structures that are amenable to machine learning, we define attributes at each vertex that summarize code characteristics as numerical values. Initially the attributes computed at a vertex do not contain any structural information, which means that their values are independently collected from the corresponding basic block. Table I lists the attributes implemented in our prototype system, although more attributes can be conveniently added to further improve malware classification performance.

TABLE I
BLOCK-LEVEL ATTRIBUTES USED IN MAGIC

Attribute Type	Attribute Description
From Code Sequence	# Numeric Constants
	# Transfer Instructions
	# Call Instructions
	# Arithmetic Instructions
	# Compare Instructions
	# Mov Instructions
	# Termination Instructions
	# Data Declaration Instructions
	# Total Instructions
From Vertex Structure	# Offspring, i.e., Degree
	# Instructions in the Vertex

As the raw attributes in an attributed CFG (ACFG) contain little structural information and the number of vertices in an ACFG varies with the individual program from which the CFG is derived, for the purpose of malware classification it is necessary to aggregate these attributes over all the vertices in the ACFG in an organic manner depending on its graph structure. The task of such attribute aggregation is accomplished with DGCNN, which shall be explained next.

C. Deep Graph Convolution Neural Network

Unlike image or text-based data, graph-based data are of variable sizes and are thus not naturally ordered tensors. To

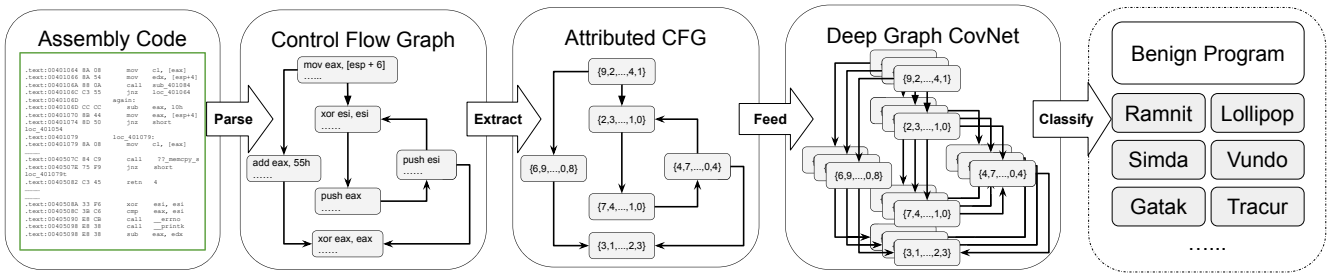


Fig. 1. The workflow of MAGIC, a DGCNN-based malware classification system

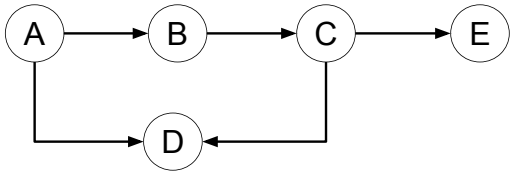
address this challenge, we use the state-of-the-art deep neural network that can automatically learn discriminative latent features from malware data abstracted as ACFGs. Particularly, we use deep graph convolution neural network to transform unordered graph data of varying sizes to tensors of fixed size and order. The transformation algorithm first recursively propagates the weighted attributes in each vertex through the neighborhood defined by the graph structure. Next, it sorts the vertices in the order of their feature descriptors. After the sorting step, the graphs with variant sizes are embedded into fixed-size vectors, which are amenable to ML-based classification. In the next section, we shall elaborate on these operations as well as our extensions based on formal mathematical descriptions.

III. Algorithm Description

Our work on applying DGCNN for malware classification in MAGIC has been inspired by the deep learning model proposed in [27]. In this section, we first introduce how DGCNN aggregates attributes through the neighborhood defined by the graph structure. We then discuss how to extend the existing DGCNN model with our own modifications. To explain the rationale of the DGCNN-based malware classification algorithm clearly, we walk through an example graph with five vertices as shown in Figure 2.

A. Primer on DGCNN

1) Notations: We denote the adjacency matrix of a graph $G = (V, E)$ of n vertices as $\mathbf{A} \in \mathbb{Z}^{n \times n}$. Note that G is a directed graph, and $\mathbf{A}$ is not necessarily symmetric. To allow the attributes of a vertex to be propagated back to the vertex itself, we define the augmented adjacency matrix $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$. Accordingly, the augmented diagonal degree matrix of G is defined as $\tilde{\mathbf{D}}$, where $\tilde{\mathbf{D}}_{i,i} = \sum_j \tilde{\mathbf{A}}_{i,j}$. We assume that each vertex is associated with a c-dimension attribute vector. Therefore, we use $\mathbf{X} \in \mathbb{R}^{n \times c}$ to denote the attribute matrix for all the vertices in the graph. Alternatively, we can also treat $\mathbf{X}$ as the concatenation of c *attribute channels* of the graph. For the sample graph g in Figure 2, we displayed the corresponding augmented adjacency matrix $\tilde{\mathbf{A}}$, the augmented diagonal degree matrix $\tilde{\mathbf{D}}$, and the attribute matrix $\mathbf{X}$ with two attribute channels $F1$ and $F2$. Given $\tilde{\mathbf{A}}$ and $\mathbf{X}$ for graph G, the DGCNN-based algorithm performs three sequential stages to obtain its tensor representation for malware classification. Note that $\tilde{\mathbf{D}}$ can be calculated from $\tilde{\mathbf{A}}$.



$\tilde{\mathbf{A}}$	A	B	C	D	E
A	1	1	0	1	0
B	0	1	1	0	0
C	0	0	1	1	1
D	0	0	0	1	0
E	0	0	0	0	1

$\tilde{\mathbf{D}}$	A	B	C	D	E
A	3	0	0	0	0
B	0	2	0	0	0
C	0	0	3	0	0
D	0	0	0	1	0
E	0	0	0	0	1

$\mathbf{X}$	F1	F2
A	2	3
B	1	5
C	7	4
D	8	9
E	6	1

Fig. 2. An sample graph g used in Section III to illustrate how the extended DGCNN works in MAGIC. Assuming the vertices have two attribute channels, we show g's augmented adjacency matrix $\tilde{\mathbf{A}}$, augmented diagonal degree matrix $\tilde{\mathbf{D}}$, as well as attribute matrix $\mathbf{X}$.

2) Graph Convolution Layer(s): In the first stage, a *graph convolution* technique propagates each vertex's attributes to its neighborhood based on the structural connectivity. To aggregate multi-scale substructural attributes, multiple graph convolution layers are stacked, which can be defined recursively as follows:

$$\mathbf{Z}^{t+1} = f(\tilde{\mathbf{D}}^{-1}\tilde{\mathbf{A}}\mathbf{Z}^{t}\mathbf{W}^{t}) \tag{1}$$

where $Z^0 = X$. The t-th layer takes input $Z^t \in \mathbb{Z}^{n \times c_t}$, mapping c_t feature channels into c_{t+1} feature channels with the graph convolution parameter $\mathbf{W}^t \in \mathbb{R}^{c_t \times c_{t+1}}$. The newly obtained channels of each vertex are then propagated to both its neighboring vertices and itself, first multiplied with the augmented adjacency matrix $\tilde{\mathbf{A}}$, and then normalized row-wisely using the augmented degree diagonal matrix $\tilde{\mathbf{D}}$. This key step enables vertices to pass its own attributes through the graph in a breadth-first-search fashion. Define $\mathbf{F} = \mathbf{Z}^t \cdot \mathbf{W}^t$ and $\mathbf{O} = \tilde{\mathbf{A}} \cdot \mathbf{F}$, where

$$\mathbf{O}[i][j] = \sum_{k=1}^{n} \tilde{\mathbf{A}}[i][k] \times \mathbf{F}[k][j] \tag{2}$$

$\forall 1 \leq i \leq n, 1 \leq j \leq c_t$. In other words, the j-th feature channel of vertex i is computed as a linear combination of all its neighbors' j-th feature channels. The layer finally outputs the element-wise activation using a nonlinear function f. At the end of h graph convolution layers, DGCNN concatenates each layer's output Z^t, denoted as $\mathbf{Z}^{1:h} = [\mathbf{Z}^1, \mathbf{Z}^2, \ldots, \mathbf{Z}^h]$. For the sample graph g, Figure 3 shows how the two sequential graph convolution layers transform the initial attribute matrix $\mathbf{Z}_0 = \mathbf{X}$ to $\mathbf{Z}_1$ and $\mathbf{Z}_2$, both of which together form $\mathbf{Z}^{1:2}$.

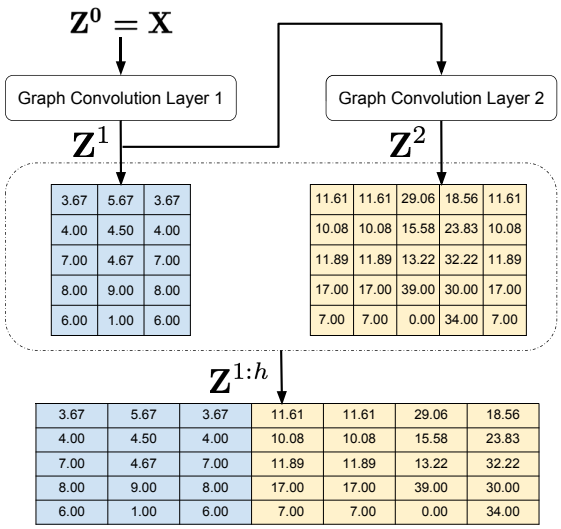


Fig. 3. After applying $h = 2$ graph convolution layers, the sample graph g is transformed to $\mathbf{Z}^{1:h}$. We assume the weight parameters in the two graph convolution layers are $\mathbf{W}^1 = [[1, 0, 1], [0, 1, 0]]$ and $\mathbf{W}^2 = [[0, 1, -2, 2], [1, 1, 7, -2], [1, 0, -1, 4]]$. For simplicity, numbers are of 2 precision, and we perform the element-wise RELU nonlinear activation $f(x) = max(x, 0)$ in both graph convolution layers.

3) SortPooling Layer: Intuitively $\mathbf{Z}^{1:h}$ has n rows and $\sum_1^h c_t$ column, which corresponds to the *feature descriptor* of each vertex at different scales. The second stage, namely the *sortpooling* layer, leverages the feature descriptors to sort the vertices. Vertices in different graphs will be put in similar positions as long as they have similar weighted feature descriptors. The sortpooling layer starts with the last layer because $\mathbf{Z}^h$ is approximately equivalent to the most refined continuous colors as in the Weisfeiler-Lehman graph kernels [29]. More specifically, vertices are first sorted by the last channel of the last layer in a decreasing order. If there are c_h ties on the last layer's output $\mathbf{Z}^h$, sorting continues by using the second last layer's output $\mathbf{Z}^{h-1}$, and the procedure repeats until all ties are broken. The sortpooling layer further truncates or pads the sorted tensors by the first dimension so that it outputs $\mathbf{Z}^{sp}$ of size k by $\sum_1^h c_t$. Hence, the sortpooling process unifies the size of feature descriptors for all graphs. Following our sample graph g, we visualize this process in Figure 4. The row of $\mathbf{Z}^h$ is first sorted using only the value in the last column. The last two rows (i.e., yellow and red) are discarded from the sorted matrix as $n - k = 2 > 0$.

4) Remaining Layer: In the last stage, the authors of the original DGCNN [27] append a one-dimension convolution (Conv1D) layer of kernel size $\sum_1^h c_t$ and stride size $\sum_1^h c_t$. If F is the number of filters in the last one-dimension convolution layer, the sort pooling output $\mathbf{Z}^{sp}$ will be reduced to a one-dimension vector of size $k \times F$, which is then fed into a fully connected one-layer perceptron for graph classification.

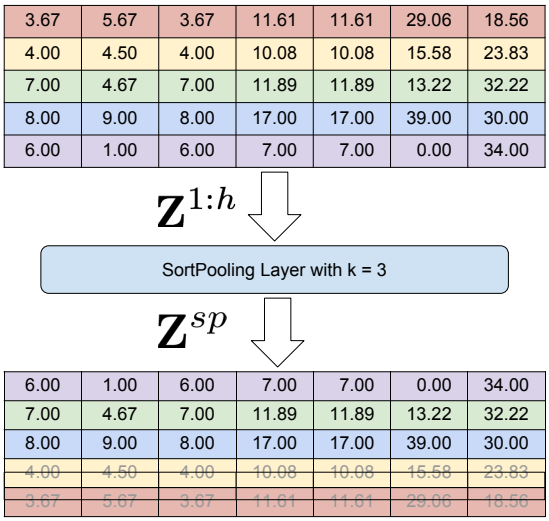


Fig. 4. Given the graph convolution result $\mathbf{Z}^{1:h}$ for the sample graph g in Figure 3, the sortpooling layer with $k = 3$ sorts the feature descriptors (based on only the last feature channel in this example) and then truncates the two 'smallest' rows.

B. WeightedVertices Layer

In the first extension to DGCNN, we observe that the Conv1D layer following the sortpooling layer can alternatively be of kernel size k, stride size k, and single channel. Mathematically, a single channel Conv1D layer can be represented as a row of parameters $W \in \mathbb{R}^{1 \times k}$. Its output $E \in \mathbb{R}^{1 \times \sum_1^h c_t}$, when fed with *transposed* $\mathbf{Z}^{sp}$, will be equivalent to

$$E = f(\mathbf{W} \times \mathbf{Z}^{sp}) \tag{3}$$

This is because

$$E_c = f(\sum_i^k W_i \times \mathbf{Z}^{sp}_{i,c}) \tag{4}$$

where $1 \leq c \leq \sum_1^h c_t$, and f is an element-wise nonlinear activation function. Inspired by the graph embedding idea in [30], our Conv1D layer treats each row of the sort pooling result $\mathbf{Z}^{sp}_i$ as the embedding of the vertices kept by the sortpooling layer.

Equivalently, Equation (3) computes E, the embedding of the graph obtained through a weighted summation of vertex embeddings [30]. For our sample graph g, its "embedding" is computed in Figure 5, where we assume weight vector $\mathbf{W} = [0.4, 0.1, 0.5]$. In reality, $\mathbf{W}$ is updated by gradient descent during the process of minimizing the classification

loss. For ease of presentation, in the remainder of the paper we refer to this special Conv1D layer after the sortpooling layer as the *WeightedVertices* layer. We replace the original Conv1D layer with the WeightVertices layer, because the WeightVertices layer leverages the graph embedding idea to make the output from the sorting pooling layer compatible with the malware classifier.

C. AdaptiveMaxPooling: An Alternative to Sortpooling

The intuition behind sorting from the deeper layer is to treat its output as more refined WL colors [29], [31]. The inner sorting inside the channels of a fixed layer output is, however, less reasonable. Besides, the Conv1D addendum is only aggregating the feature descriptors of per vertex and per convolution channel separately.

Our second extension is to apply the adaptive max pooling (AMP) on the concatenated graph convolution layer output $\mathbf{Z}^{1:h}$. Given an set of two-dimension inputs of various sizes $\{x_i | x_i \in \mathbb{R}^{h_i \times w_i}\}$, The AMP layer divides each input x_i into a $H \times W$ grid with a sub-window size approximately to h_i/H and w_i/W, and then automatically chooses kernel sizes as well as convolution strides for different x_i. Inside each sub-window and each channel, only the maximum element is kept in order to form the set of identical-dimension outputs $\{y_i | y_i \in \mathbb{R}^{H \times W}\}$. The way in which AMP works for our sample graph g is illustrated in Figure 6. Since the dimension of the graph convolution output $\mathbf{Z}_g^{1:h}$ for g is 5×7, AMP uses a max pooling kernel of size 3×3. To show how AMP works for inputs of different dimension sizes, in Figure 6 we also feed $\mathbf{Z}_{g'}^{1:h}$, the graph convolution output for another graph g' (not shown here), to the 3×3 AMP layer. In this case, the kernel size is adaptively adjusted to 2×3.

We have two motivations for using AMP at the end of the graph convolution layer. In addition to unifying the convolution layer output $\mathbf{Z}^{1:h}$, AMP empowers us to aggregate $\mathbf{Z}^{1:h}$ across the dimensions of both feature channels and graph vertices simultaneously, which enables us to capture informative features that vary only by location. This is easily accomplished by applying a two-dimension convolution (Conv2D) layer with an arbitrary number of filters before the AMP layer. The output of the AMP layer is further fed into a multiple-Conv2D-layer neural network, inspired by VGG [32], to predict the probability distribution of the malware families that the input CFG should belong to.

IV. Implementation

In this section, we discuss a few implementation details, which include how we derive CFGs from dissembled code, what kind of loss functions are used in model training, and the open source MAGIC project.

A. Details in Building CFG

To build a control flow graph from disassembled code in possibly different formats, we first pre-process the input files so that the resulting program P is a one-to-one mapping from *sorted addresses* to *assembly instructions*, e.g., $P : \mathbf{Z}^+ \rightarrow \mathbf{I}$.

We then perform a two-pass iteration over P. Instructions $inst \in \mathbf{I}$ are associated with a couple of tags, i.e., {`start`, `branchTo`, `fallThrough`, `return`}, used by the second pass for creating code blocks and connecting blocks. To adapt to (potentially) hundreds of types of instructions, the first pass applies the visitor pattern to implement if-else free instruction tagging. As an example, Algorithm 1 details the tagging operations when visiting a conditional jump instruction cj. This procedure relies on a helper function `findDstAddr(`$inst$`)` to extract the destination address of a jump instruction $inst$. For a conditional jump instruction, it branches to the jump target ($P[dstAddr]$, handled by line 2 – 3) and falls through to the next instruction ($P[cj.addr + cj.size]$, handled by line 4 – 5).

Algorithm 1 `visitConditionalJump(`cj`)`

1: $dstAddr \leftarrow$ `findDstAddr(`cj`)`
2: $cj.branchTo \leftarrow dstAddr$
3: $P[dstAddr].start \leftarrow true$
4: $cj.fallThrough \leftarrow true$
5: $P[cj.addr + cj.size].start \leftarrow true$

The second pass creates code blocks (or vertices) and connects blocks on the fly. The skeleton of the procedure is illustrated in Algorithm 2. Algorithm 2 assumes two trivial helper functions. Firstly, `getBlockAtAddr(`$addr$`)` returns the block starting at $addr$ if it already exists; otherwise it creates a new block starting at $addr$. The second one `getNextInst(`$P, inst$`)` returns the instruction next to $inst$ if it exists; otherwise, `None` is returned. With both helpers, Algorithm 2 works in three steps. The first if statement creates a new block if $inst$ was marked as `start` in the first pass. The second step connects $block$ to $nextBlock$ if the last instruction in $block$ is falling through to the next instruction, which happens to be the start of $nextBlock$. The final step creates an edge (potentially a new block) for any branching operations, e.g., jump or call.

B. Loss Functions Used in Model Training

Another technical advantage of MAGIC is its support for the end-to-end deep neural network training. Regardless of how we change the layer configurations, i.e., whether to use the sort pooling layer or the adaptive max pooling layer, the model's output is always the prediction of the observed input. Therefore, the training procedure always minimizes the mean negative logarithmic loss, which is defined as

$$\mathcal{L} = \sum_{i=1}^{N} \sum_{c=1}^{C} y_{i,c} \log(p_{i,c}) \tag{5}$$

where N is the number of observations in the dataset, C is the number of malware families in the dataset, $y_{i,c}$ is 1 if the ith sample belongs to malware family c, and $p_{i,c}$ is the predicted probability that ith sample is in family c in the output of the model. We adopt the Adam optimization algorithm [33] implemented in PyTorch [34] to auto-generate the gradient

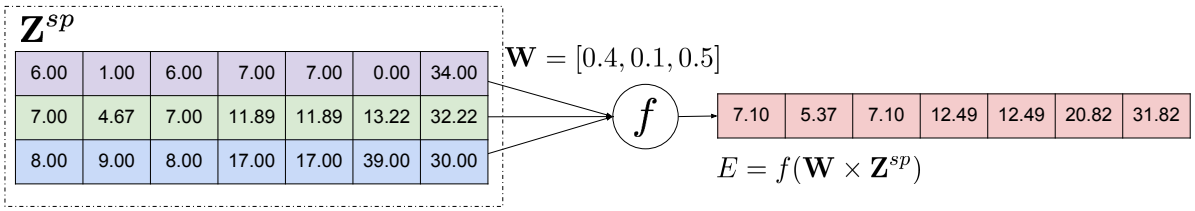


Fig. 5. WeightVertices layer aggregates sample graph g's vertex embeddings, e.g. the output of sort pooling layer $\mathbf{Z}^{sp}$ in Figure 4, to graph embedding E. We choose again RELU as the nonlinear activition function f for simplicity.

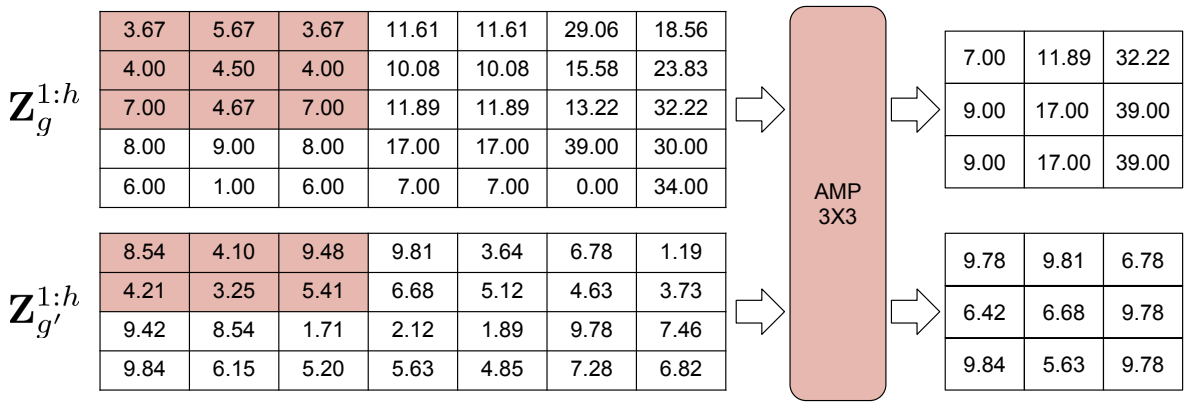


Fig. 6. An example of 3×3 adaptive max pooling layer with two two-dimension inputs with different sizes. The left top matrix $\mathbf{Z}_g^{1:h}$ represents the graph convolution output of the sample graph g in Figure 2. The left bottom matrix $\mathbf{Z}_{g'}^{1:h}$ represents the graph convolution output of another imaginary graph g' with four vertices. For $\mathbf{Z}_g^{1:h}$ of size 5×7, adaptive max pooling's kernel size = 3×3 (shown as red shadow). For $\mathbf{Z}_{g'}^{1:h}$ of size 4×7, adaptive max pooling's kernel size = 2×3 (shown as red shadow). For both inputs, padding = 0, stride = 2×1.

Algorithm 2 `CfgBuilder::connectBlocks()`

for all $inst$ in program P **do**
  if $inst.start$ **then**
    $currBlock \leftarrow$ `getBlockAtAddr`$(inst.addr)$
  end if
  $nextBlock \leftarrow currBlock$

  $nextInst \leftarrow$ `getNextInst`$(P, inst)$
  if $nextInst \neq$ None **then**
    if $inst.fallThrough$ **and** $nextInst.start$ **then**
      $nextBlock \leftarrow$ `getBlockAtAddr`$(nextInst.addr)$
      add $nextBlock$ to $currBlock$'s edge list
    end if
  end if

  if $inst.branchTo \neq$ None **then**
    $block \leftarrow$ `getBlockAtAddr`$(inst.branchTo)$
    add $block$ to $currBlock$'s edge list
  end if

  add $inst$ to $currBlock$
  $currBlock \leftarrow nextBlock$
end for

of the model parameters and update the parameters in the DGCNN (e.g. $W^t, 1 \leq t \leq h$ in graph convolution layers) accordingly.

C. Open Source MAGIC Project

For malware classification tasks, MAGIC runs either in the training mode or in the prediction mode. In the training stage, we repeatedly activate only the first half of the pipeline to obtain a large amount of labeled CFGs. Then, a DGCNN and its classifier are trained using the stochastic gradient descent on the labeled CFGs in a batch mode. When the training finishes, MAGIC takes the CFGs of unknown binary executables as inputs and make predictions.

We have implemented a prototype system of MAGIC with approximately 4,000 line of Python code. The implementation can be divided into two independent parts. The first part generates ACFGs from either assembly code or control flow graphs. Due to the necessity of processing a large number of assembly programs, MAGIC can generate multiple ACFGs in parallel using Python's multi-threading library. The second part handles the training, tuning, and evaluation of the extended DGCNN, which is built upon, but heavily rewritten, from Muhang's PyTorch implementation [35]. For example, we developed the adaptive pooling layer and the WeightedVertices layer. Besides, MAGIC supports automatic and

exhaustive hyper-parameter tuning, cross-validation, training and prediction using GPUs. We will make MAGIC's codebase publicly available at Github in the near future.

V. Experimental Evaluation

We evaluate the performance of MAGIC using two large malware datasets, each with more than 10,000 samples, and present our experimental results in this section.

A. Malware Datasets

The first dataset, which we refer to as the *MSKCFG* dataset, includes the CFGs derived from the malware files used in the 2015 Microsoft Malware Classification Challenge hosted by Kaggle [36]. The dataset contains samples that fall into nine families: {Ramnit, Lollipop, Kelihos_ver3, Vundo, Simda, Tracur, Kelihos_ver1, Obfuscator.ACY, Gatak}. Figure 7 presents the number of samples in each of these nine malware families in the dataset. In the competition, Kaggle provided 10,868 labeled malware samples as the training dataset, for each of which two files were given. The first file contains the raw binary content in a hexadecimal representation (referred as .byte file in the following discussion). The second file is the corresponding assembly code of the binary code, which was generated with the IDA Pro tool [28] (referred as .asm file in the following discussion). The correctness of the .asm file is not guaranteed because PE headers were erased from the raw malware files for sterility before they were disassembled and sophisticated binary packing techniques may also prevent reverse engineering tools from disassembling the malware correctly [37]. In our experiments, only the .asm files were used for malware classification. We generated a total number of 10,868 ACFGs from the training .asm files, which took approximately 17 hours to finish, or averagely 5.8 seconds per malware instance, using a commodity desktop equipped with Intel Core i7-6850K CPU and 64 GB memory.

Another dataset, which we refer to as *YANCFG*, includes the CFGs of 16,351 binary executable files which were used in [8]. Similar to the MSKCFG dataset, the PE headers were not available to us for malware classification from the second dataset. All the CFGs were labeled with a majority voting scheme based on the detection results of five major AV scanners returned by the VirusTotal online malware analysis service [38]. All the CFGs belong to 12 distinct malware families: {Bagle, Benign, Bifrose, Hupigon, Koobface, Ldpinch, Lmir, Rbot, Sdbot, Swizzor, Vundo, Zbot, Zlob}. Figure 8 depicts the number of samples for each of these 12 families in the dataset. These CFGs were further converted to their corresponding ACFGs by MAGIC within 6.8 hours using the same desktop machine as mentioned above.

We did not merge two malware datasets in our experiments due to the following reasons. Firstly, the YANCFG dataset carries pre-processed CFGs, while we developed our own parser to extract CFGs from the malware assembly code in the Microsoft dataset (see Section IV-A). The CFG extracted from the MSKCFG dataset by our own parser has different low-level feature representation from that of the CFGs pre-given in YANCFG; so they cannot be applied to one model. Secondly, testing MAGIC on datasets collected from independent sources also allows us to gain insights into its generality when applied in different operational environments.

B. Model Training and Evaluation

As the malware families are different in the two datasets, we need to create two different MAGIC instances to classify their malware samples separately. However, MAGIC uses the same way to train DGCNN and tune its hyperparameters for both datasets. The first step is hyperparameter tuning. Table II lists the hyperparameters used in both the deep neural network itself (e.g., the sizes of the graph convolution layers) and the algorithm for training the model (e.g., batch size and learning rate). To determine the optimal values of these hyperparameters, we exhaustively search all 208 hyperparameter settings defined by the value ranges listed in Table II. In particular, 64 DGCNN models use adaptive pooling, 96 DGCNN models use the sort pooling and Conv1D layer, and 48 DGCNN models use the sort pooling and WeightVertices layer. We apply the five-fold cross-validation technique to evaluate the performance of a model under a specific hyperparameter setting. To conduct five-fold cross validation, the dataset is splitted into five equal-size subsets. In each fold of the cross validation, four subsets (80%) of the data are used for training a brand new model initialized randomly, and the rest subset (20% of the data), different in each fold, is used to evaluate the resultant model. In this way, the training process never sees the testing samples used for performance evaluation. We train each model with 100 epochs and record the negative log-likelihood validation loss after every epoch.

The validation loss collected after each epoch is used to find the hyperparameters that can mitigate the overfitting issue. Once the validation loss increases for two continuous epochs, we decrease the learning rate by a factor of ten to prevent the model from overfitting the training dataset. For a particular model, when all five training-validation folds are finished, we compute each epoch's validation loss by averaging the five validation losses over the five folds. The minimum validation loss over the 100 epochs is treated as the score of this model and is further used as the criterion for comparing with different hyperparameter settings. In other words, after the five-fold cross-validation for all the 208 model training instances, we choose the best model with the minimum average validation loss. Besides the average validation negative log-likelihood loss, we also measure its precision, recall, and F1 score averaged over the five validation sets (together referred to as the *cross-validation scores*), and then compare the best model's performance against those of previous works. We used four GeForce GTX 1080 Ti graphic cards to train and run the 208 variants of DGCNN. Training a particular model is always done on a single GPU, but the evaluation procedure actually takes up all the four GPUs because our MAGIC implementation supports parallel model training on multiple GPUs. The last two columns in Table II describe the best models chosen by MAGIC for the MSKCFG and YANCFG

TABLE II
HYPERPARAMETERS AND SEARCH RANGES DURING TUNING

Hyperparameter	Choice or Value Range	Best Model for MSKCFG	Best Model for YANCFG
Pooling Type	[Adaptive Pooling, Sort Pooling]	Adaptive Pooling	Adaptive Pooling
Pooling Ratio	[0.2, 0.64]	0.64	0.2
Graph Convolution Size	[(32, 32, 32, 1)[1], (32, 32, 32, 32), (128, 64, 32, 32)]	(128, 64, 32, 32)	(32, 32, 32, 32)
Remaining Layer[2]	[1D Convolution Layer, WeightVertices Layer]	Not Applicable	Not Applicable
2D Convolution Channels[3]	[16, 32]	16	16
1D Convolution Channel Pairs[4]	[(16, 32)]	Not Applicable	Not Applicable
1D Convolution Kernel Size[5]	[5, 7]	Not Applicable	Not Applicable
Dropout Rate	[0.1, 0.5]	0.1	0.5
Batch Size	[10, 40]	10	40
Weight L2 Regularization Factor	[0.0001, 0.0005]	0.0001	0.0005

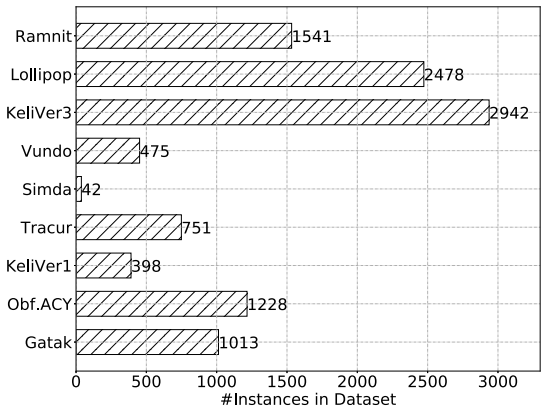


Fig. 7. Malware Family Distribution in MSKCFG Dataset.

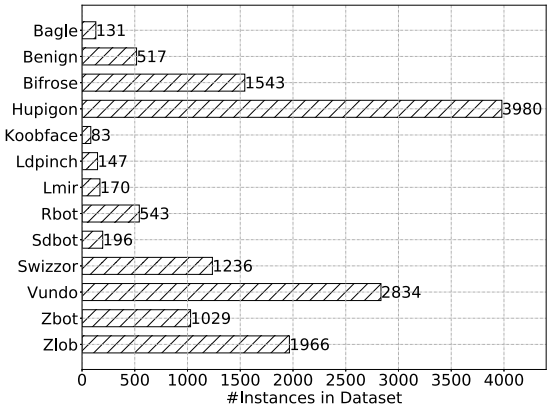


Fig. 8. Malware Family Distribution in YANCFG Dataset.

datasets, respectively. In the following, we report the cross-validation scores for the two datasets.

C. Results on the MSKCFG Dataset

The best cross-validation scores (precision, recall and F1) for the MSKCFG dataset are shown in Figure 9, and the exact score values are listed in Table III. The standard variances are not listed because the scores' variations among five different cross-validation folds are negligible (< 0.004). For all nine

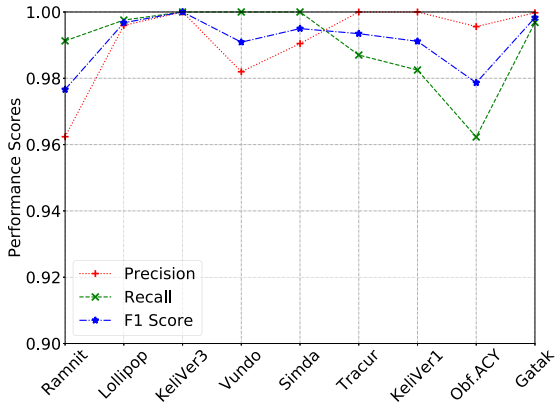


Fig. 9. Cross-Validation Scores of MAGIC on the MSKCFG Dataset.

TABLE III
PERFORMANCES OF MAGIC ON THE MSKCFG DATASET.

Family	Precision	Recall	F1
Ramnit	0.962378	0.991289	0.976615
Lollipop	0.995960	0.997550	0.996754
Kelihos_Ver3	1.000000	1.000000	1.000000
Vundo	0.981975	1.000000	0.990895
Simda	0.990476	1.000000	0.994987
Tracur	1.000000	0.987013	0.993463
Kelihos_Ver1	1.000000	0.982493	0.991156
Obfuscator.ACY	0.995593	0.962293	0.978655
Gatak	0.999775	0.996841	0.998304

malware families, our best model has achieved good validation scores with precisions higher than 0.96, recalls higher than 0.96, and F1-scores higher than 0.97.

Since Microsoft released the competition dataset in 2015, many researchers have used the dataset (completely or partially) to evaluate their techniques for malware detection and classification [9]–[15], [39]. We surveyed the works men-

[1]Only for sort pooling
[2]Applicable only when set *pooling type* to sort
[3]Applicable only when set *pooling type* to adaptive pooling
[4]Applicable only when set *pooling type* to sort pooling and *remaining layer* to 1D convolution
[5]Applicable only when set *pooling type* to sort pooling and *remaining layer* to 1D convolution

tioned above and found that three of them cannot be directly compared with our results because they were using different metrics. The works in [39] and [12] used the Microsoft dataset in the context of *malware detection*, where all the samples contained within it were treated as malicious, and then they were merged with a number of benign programs to construct a new dataset for malware detection. As a result, their methods and performance metrics (two-class AUC, F1 score or accuracy) are not directly comparable to the approaches aimed at classifying malware samples into the corresponding families (e.g., [9]–[11], [13]–[15]), this work included. Note that without loss of generality, benign software can be treated as a special family. The work in [10] did not adopt the cross-validation methodology. Instead, the authors manually split the training dataset into 75% training data and 25% holdout testing data, and reported the mean square error, accuracy and confusion matrix over both training and testing data. The holdout set is not the test set provided by Microsoft. Therefore, we did not compare our work with the evaluation results reported in [39], [12] and [10].

Among the other five papers of malware classification, both [11] and [14] conducted the ten-fold cross validation over the Microsoft dataset, but only reported the overall accuracy. [15] also performed a ten-fold cross validation but reported both the overall accuracy and logarithmic loss. Both [13] and [9] performed a five-fold cross validation and reported both the overall accuracy and logarithmic loss. Since the Microsoft dataset is not balanced across malware families, we compare MAGIC with the five previous works that reported not only the overall accuracy but also the mean logarithmic loss, and the results are shown in Table IV.

The methods listed in Table IV can be classified into either ensemble-learning or single-model based approaches. [13] extracts more than 1800 features and uses gradient boosting based classifier, and it achieves the best log-loss (0.0197) and accuracy (99.42%) using the XGBoost classifier. [11] achieves the second best accuracy (99.3%) by ensembling multiple random forest methods, which already ensembles multiple decision trees. The DGCNN-based technique used by MAGIC achieves highly competitive results. In fact, the logarithmic loss (0.0543) is the second best; the accuracy (99.25%) is the third best, only 0.005 less than the second best one (99.3% reported by [11]). [9] adopts a deep-learning based hybrid approach. It relies on a single deep autoencoder to perform automatic feature learning, and then uses gradient-boosting based classifier to make the prediction. As an alternative work that also applies deep neural network, our DGCNN-based approach outperforms the work in [9] by 27.40% in terms of logarithmic loss and 1.5% in terms of classification accuracy.

D. Results on the YANCFG Dataset

MAGIC's best cross validation scores on the YANCFG dataset are depicted in Figure 10 and the exact values of these scores are listed in Table V. We observe in Figure 10 that MAGIC achieves F1-scores higher than 0.9 for nine of the 13 binary families including {Bagle, Benign, Bifrose, Hupigon, Koobface, Swizzor, Vundo, Zbot, Zlob}. The classification performances on both the Koobface and Swizzor families are perfect with a precision of 1.0 and a recall of 1.0. Regarding the other five families with F1 scores lower than 0.8, they all have relatively small populations in the YANCFG dataset. Our classifier suffers relatively low recalls (around 0.5) for both the Ldpinch and Sdbot families. For the Ldpinch, Rbot and Sdbot families, their precision scores (between 0.64 and 0.70) are not as good as the other ten families (more than 0.8).

In order to further assess MAGIC, we compared our results to the F1 scores obtained in [8]. That work ensembles a group of individual SVM (Support Vector Machine)-based classifiers (refer to **ESVC** hereafter). Our work does not use the raw hexadecimal bytes, the PE headers, and the execution traces in the original dataset.We plot the comparison results in Figure 11 as the relative and absolute amount of improvement to ESVC as achieved by MAGIC. Note that the improvement statistics for the Benign family are not shown in Figure 11 because the F1 score for the benign samples is not reported in [8]. The positive values in Figure 11 mean factual improvement, while the negative values mean degradation. Close to the bottom of the figure, we observe that the only family over which MAGIC performs visibly poorer than ESVC is Rbot, with an approximate performance degradation of 0.07 relatively and 0.05 absolutely. For Hupigon, the downgradation is nearly invisible (less than 0.01 both relatively and absolutely), and both approaches achieve F1 scores higher than 0.94. On the other hand, MAGIC outperforms ESVC for the other ten families. Moreover, the amount of absolute improvement is greater than or equal to 0.2 for each of the Bagle, Koofbace, Ldpinch and Lmir families. Lastly, it is noted that both approaches performed relatively poorly on the Ldpinch and Lmir malware families. Still, the DGCNN-based approach used by MAGIC improves the work in [8] by 70% and 35% in terms of the F1-score for Ldpinch and Lmir, respectively.

E. Discussion

We break down the major execution overhead of MAGIC into three parts: feature extraction time, classifier training time, and malware prediction time. Building ACFGs for the MSKCFG dataset with MAGIC takes less than 6 seconds on average. We gathered the training and testing running time over 20 runs to evaluate MAGIC. The mean and standard deviation of the classifier training time per instance among the 20 runs is approximately 29.69 ± 4.90 milliseconds, while the mean and standard deviation of malware prediction time per instance is only 11.33 ± 1.35 milliseconds. Our measurements of the execution overhead of MAGIC suggest that it is actionable for online malware classification in practice.

By design, MAGIC is aimed at striking a balance between generality and performance. XGBoost [13] achieves impressive performance (99.42% CV accuracy for MSKCFG), but relies on various handcrafted features (more than 1800 from the MSKCFG dataset) and time-consuming feature selection techniques (e.g., forward stepwise selection). In contrast, MAGIC achieves a similar performance (99.25%) with only

TABLE IV
CROSS VALIDATION METRIC COMPARISON ON THE MICROSOFT DATASET.

Approach Brief Description	Mean Logarithmic Loss	Accuracy
MAGIC	0.0543	99.25
XGBoost with Heavy Feature Engineering [13]	0.0197	99.42
Deep Autoencoder based XGBoost [9]	0.0748	98.20
Strand Gene Sequence Classifier [15]	0.2228	97.41
Ensemble Multiple Random Forest Classifiers [11]	Not Reported	99.30
Random Forest with Feature Engineering [14]	Not Reported	99.21

TABLE V
PERFORMANCE OF MAGIC ON THE YANCFG DATASET.

Family	Precision	Recall	F1 Score
Bagle	0.863636	0.950000	0.904762
Benign	0.954128	0.962963	0.958525
Bifrose	0.930380	0.901840	0.915888
Hupigon	0.935287	0.945679	0.940454
Koobface	1.000000	1.000000	1.000000
Ldpinch	0.692308	0.514286	0.590164
Lmir	0.833333	0.731707	0.779220
Rbot	0.641221	0.763636	0.697095
Sdbot	0.700000	0.488372	0.575342
Swizzor	0.995708	0.995708	0.995708
Vundo	0.990859	0.981884	0.986351
Zbot	0.941799	0.936842	0.939314
Zlob	0.967254	0.992248	0.979592

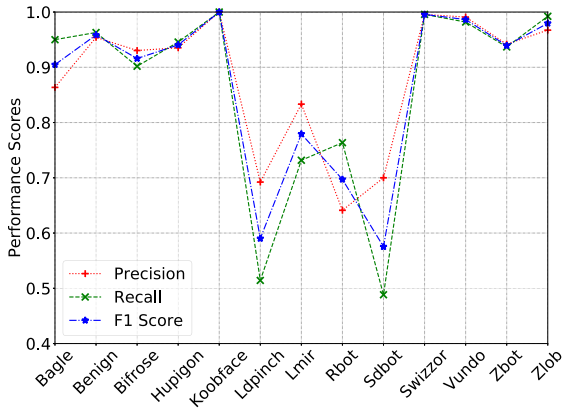


Fig. 10. Cross-Validation Scores of MAGIC on the YANCFG Dataset.

a dozen easy-to-extract attributes embedded within malwares CFG structures. The work in [8] sequentially integrates SVM-based malware classifiers trained from heterogeneous features, but its use of dynamic programming to search an optimal malware classifier with a bounded false positive rate increases model training time significantly.

The YANCFG and MSKCFG datasets are the two largest labeled malware datasets that we could obtain to evaluate our work. It is possible that malware development trends after the collection of these two datasets introduce new challenges to the malware classification problem. We plan to test our models with the latest malware samples in our future work.

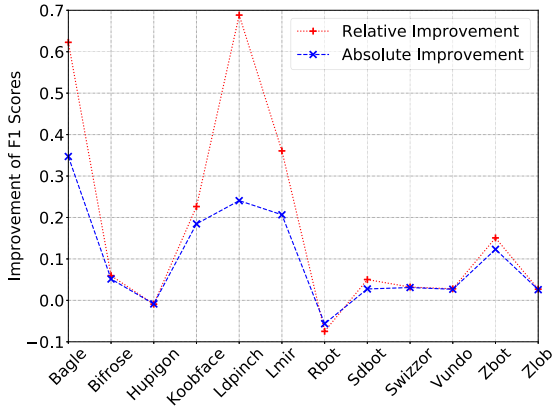


Fig. 11. F1 Score Comparison between MAGIC and ESVC [8] on the YANCFG Dataset. Improvement on the classification accuracy of benign samples is not shown because it is not reported in [8].

VI. RELATED WORKS

We introduce related works on deep learning based malware classification and deep neural networks for graph data.

A. Deep Learning Based Malware Detection

As an important problem in cybersecurity, malware detection and classification have drawn the attentions of many researchers from both communities of cybersecurity and data mining [40], [41]. There has been a recent trend of applying deep learning techniques for malware defense tasks and these works largely fall into two categories. In the first category, which adopts a feature-centric approach, researchers reuse or extend features developed in previous works but extracted from the newer datasets collected from customized, private, and specific environments [9], [10], [42]–[47]. For example, the work in [43] focused on malware collected from Android devices and that in [44] targets malware on the cloud platforms. Off-the-shelf deep learning models have been used as tunable black boxes. Compared with classic machine learning algorithms like decision trees, random forest, and gradient boosting, deep learning models fed with the same training datasets may result in better detection performance [43], [45] and faster execution [42], because of their advantages on big data analysis [45] and the possibility of using parallel computing hardware (i.e., GPUs). Moreover, the works in [9] and [10] have focused on automation of feature learning using unsupervised deep learning techniques.

In the second category, which adopts a model-centric approach, researchers are motivated by finding specific but superior deep learning architectures for malware defense tasks. For example, to find similar success of convolution neural network in the application domains of computer vision and image classification, researchers have proposed methods to transform the byte sequences in binary malware executables into gray or color images, which are amenable to the existing deep learning-based image classification techniques [48], [49]. Other researchers have explored how deep sequence models, such as LSTM, GRU, and the attention mechanism, can be applied to programs the sequences of system calls or API calls transformed from malware programs [46], [47].

Our work takes a hybrid approach that intersects with the works in both categories. First, our work improves the accuracy of malware classification using not only the features that can be explicitly expressed with numeric values (i.e., attributes extracted from basic blocks) but also those that are inherent within the structure of the program (i.e., control flow graphs). On the other hand, deep learning models are not used as black boxes in our work, as we have proposed modifications to the standard DGCNN that are better tailored to the malware classification problem.

B. Deep Learning Models for Graph-Represented Data

There are two parallel lines of research on deep learning algorithms for graph-represented data. In the first setting [50]–[52], a single graph is given and the task is to infer unknown labels of individual vertices, or unknown types of connectivity between vertices. Though this problem has wide applications in social networks and recommendation systems, it does not align well with our goal in this paper. Instead, our work fits into the second setting, where assuming a group of labeled graphs with different structures and sizes, the task is to predict the label of future unknown graphs [26], [27], [53]. In this setting, both the works in [53] and [27] have mentioned their connections with the classic Weisfeiler-Lehman subtree kernel [29] or the Weisfeiler-Lehman algorithm [31]. In contrast, the work in [53] introduces the recurrent neuron based graph kernel, then stacks multiple graph kernel neural layers into deep network. Similar to the sort pooling layer discussed in [27], the work in [26] generalizes the convolution operator and enables it to handle arbitrary graphs of different sizes. In our work, we propose to enhance the architectures introduced in [27] with both the weight vertices layer and the adaptive max pooling layer for the malware classification task.

VII. Conclusion

In this work, we have applied deep graph convolution neural network to the malware classification problem. Different from existing machine learning-based malware detection approaches that commonly rely on handcrafted features and ensemble models, this work proposes and evaluates an end-to-end malware classification pipeline with two distinguishing features. Firstly, our malware classification system works directly on CFG-represented malware programs, making it deployable in a variety of operational environments. Secondly, we extend the state-of-the-art graph convolutional neural network to aggregate attributes collected from individual basic blocks through the neighborhood defined by the graph structures and thus embed them into vectors that are amenable to machine learning-based classification. Our experimental evaluation conducted on two large malware datasets has shown that our proposed method achieves classification performances that are comparable to those of state-of-the-art methods applied on handcrafted features.

We envision MAGIC would be deployed on a cloud, as a typical end user does not have enough labeled malware samples to train good classification models. A user can upload suspicious files to the cloud, which further trains appropriate MAGIC parameters to classify programs newly seen in her operational environment. In this way, even if a particular user may not have labeled programs to train a specific neural network, he can still benefit from MAGIC who learns from many other uses that do have labeled datasets.

Acknowledgment

We would like to thank the anonymous DSN reviewers and our shepherd Long Wang for their valuable feedback. We also thank Ping Liu for the insightful discussions and Muhan Zhang for the bug fixes in the pytorch version of DGCNN. This work is partly sponsored by the Air Force Office of Scientific Research (AFOSR) under Grant YIP FA9550-17-1-0240, the National Science Foundation (NSF) under Grant CNS-1618631, and the Maryland Procurement Office under Contract No. H98230-18-D-0007. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of AFOSR, NSF, and the Maryland Procurement Office.

References

[1] "Malware Statistics & Trends Report by AV-TEST," https://www.av-test.org/en/statistics/malware, accessed: 2018-11-30.

[2] "Executive Summary - 2018 Internet Security Threat Report," https://www.symantec.com/content/dam/symantec/docs/reports/istr-23-executive-summary-en.pdf, accessed: 2018-11-30.

[3] "Crypto Mining Attacks Soar in First Half of 2018," https://www.coindesk.com/crypto-mining-attacks-soar-in-first-half-of-2018, accessed: 2018-11-30.

[4] J. Z. Kolter and M. A. Maloof, "Learning to detect and classify malicious executables in the wild," *Journal of Machine Learning Research*, vol. 7, no. Dec, pp. 2721–2744, 2006.

[5] K. Rieck, P. Trinius, C. Willems, and T. Holz, "Automatic analysis of malware behavior using machine learning," *Journal of Computer Security*, vol. 19, no. 4, pp. 639–668, 2011.

[6] R. Perdisci, A. Lanzi, and W. Lee, "McBoost: Boosting scalability in malware collection and analysis using statistical classification of executables," in *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*. IEEE, 2008, pp. 301–310.

[7] B. Anderson, D. Quist, J. Neil, C. Storlie, and T. Lane, "Graph-based malware detection using dynamic analysis," *Journal in computer Virology*, vol. 7, no. 4, pp. 247–258, 2011.

[8] G. Yan, "Be sensitive to your errors: Chaining neyman-pearson criteria for automated malware classification," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '15. New York, NY, USA: ACM, 2015, pp. 121–132.

[9] M. Yousefi-Azar, V. Varadharajan, L. Hamey, and U. Tupakula, "Autoencoder-based feature learning for cyber security applications," in *2017 International Joint Conference on Neural Networks (IJCNN)*, May 2017, pp. 3854–3861.

[10] T. M. Kebede, O. Djaneye-Boundjou, B. N. Narayanan, A. Ralescu, and D. Kapp, "Classification of malware programs using autoencoders based deep learning architecture and its application to the microsoft malware classification challenge dataset," in *2017 IEEE National Aerospace and Electronics Conference (NAECON)*, June 2017, pp. 70–75.

[11] M. Hassen and P. K. Chan, "Scalable function call graph-based malware classification," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, ser. CODASPY '17. New York, NY, USA: ACM, 2017, pp. 239–248.

[12] D. Yuxin and Z. Siyi, "Malware detection based on deep learning algorithm," *Neural Computing and Applications*, Jul 2017.

[13] M. Ahmadi, D. Ulyanov, S. Semenov, M. Trofimov, and G. Giacinto, "Novel feature extraction, selection and fusion for effective malware family classification," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '16. New York, NY, USA: ACM, 2016, pp. 183–194.

[14] M. Hassen, M. M. Carvalho, and P. K. Chan, "Malware classification using static analysis based features," in *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, Nov 2017, pp. 1–7.

[15] J. Drew, T. Moore, and M. Hahsler, "Polymorphic malware detection using sequence classification methods," in *2016 IEEE Security and Privacy Workshops (SPW)*, May 2016, pp. 81–87.

[16] G. Yan, N. Brown, and D. Kong, "Exploring discriminatory features for automated malware classification," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2013, pp. 41–61.

[17] Q. Zhang, D. S. Reeves, P. Ning, and S. P. Iyer, "Analyzing network traffic to detect self-decrypting exploit code," in *Proceedings of the 2nd ACM symposium on Information, computer and communications security*. ACM, 2007, pp. 4–12.

[18] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Automatic reverse engineering of malware emulators," in *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE, 2009, pp. 94–109.

[19] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 552–561.

[20] D. Bilar, "Opcodes as predictor for malware," *International journal of electronic security and digital forensics*, vol. 1, no. 2, pp. 156–168, 2007.

[21] D. Kong and G. Yan, "Discriminant malware distance learning on structural information for automated malware classification," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2013, pp. 1357–1365.

[22] X. Hu, T.-c. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 611–620.

[23] S. Cesare and Y. Xiang, "Classification of malware using structured control flow," in *Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing-Volume 107*. Australian Computer Society, Inc., 2010, pp. 61–70.

[24] D. K. Duvenaud, D. Maclaurin, J. Iparraguirre, R. Bombarell, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams, "Convolutional networks on graphs for learning molecular fingerprints," in *Advances in neural information processing systems*, 2015, pp. 2224–2232.

[25] P. D. Dobson and A. J. Doig, "Distinguishing enzyme structures from non-enzymes without alignments," *Journal of molecular biology*, vol. 330, no. 4, pp. 771–783, 2003.

[26] M. Simonovsky and N. Komodakis, "Dynamic Edge-Conditioned Filters in Convolutional Neural Networks on Graphs," 2017. [Online]. Available: http://arxiv.org/abs/1704.02901

[27] M. Zhang, Z. Cui, M. Neumann, and Y. Chen, "An End-to-End Deep Learning Architecture for Graph Classification," in *Proceedings of AAAI Conference on Artificial Inteligence*, 2018.

[28] "IDA Pro," https://www.hex-rays.com/.

[29] N. Shervashidze, P. Schweitzer, E. J. v. Leeuwen, K. Mehlhorn, and K. M. Borgwardt, "Weisfeiler-lehman graph kernels," *Journal of Machine Learning Research*, vol. 12, no. Sep, pp. 2539–2561, 2011.

[30] 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 *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 363–376.

[31] B. J. Weisfeiler and A. Leman, "A reduction of a graph to a canonical form and an algebra arising during this reduction," *Nauchno–Technicheskaja Informatsia*, pp. 12–16, 1968.

[32] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *ICLR*, 2015. [Online]. Available: http://arxiv.org/abs/1409.1556

[33] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014. [Online]. Available: http://arxiv.org/abs/1412.6980

[34] "PyTorch," https://www.pytorch.org, accessed: 2018-11-30.

[35] "PyTorch DGCNN," https://github.com/muhanzhang/pytorch_DGCNN, accessed: 2018-11-30.

[36] R. Ronen, M. Radu, C. Feuerstein, E. Yom-Tov, and M. Ahmadi, "Microsoft malware classification challenge," 2018. [Online]. Available: http://arxiv.org/abs/1802.10135

[37] B. Cheng, J. Ming, J. Fu, G. Peng, T. Chen, X. Zhang, and J.-Y. Marion, "Towards paving the way for large-scale windows malware analysis: Generic binary unpacking with orders-of-magnitude performance boost," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: ACM, 2018, pp. 395–411.

[38] "VirtusTotal," https://www.virustotal.com, accessed: 2018-11-30.

[39] J. Yan, Y. Qi, and Q. Rao, "Detecting malware with an ensemble method based on deep neural network," *Sec. and Commun. Netw.*, vol. 2018, pp. 17–, Mar. 2018.

[40] A. Souri and R. Hosseini, "A state-of-the-art survey of malware detection approaches using data mining techniques," *Human-centric Computing and Information Sciences*, vol. 8, no. 1, p. 3, Jan 2018.

[41] N. Idika and A. P. Mathur, "A survey of malware detection techniques," *Purdue University*, vol. 48, 2007.

[42] M. Rhode, P. Burnap, and K. Jones, "Early Stage Malware Prediction Using Recurrent Neural Networks," 2017. [Online]. Available: http://arxiv.org/abs/1708.03513

[43] D. Zhu, H. Jin, Y. Yang, D. Wu, and W. Chen, "DeepFlow: Deep learning-based malware detection by mining Android application for abnormal usage of sensitive data," in *2017 IEEE Symposium on Computers and Communications (ISCC)*, July 2017, pp. 438–443.

[44] Y. Ye, L. Chen, S. Hou, W. Hardy, and X. Li, "DeepAM: a heterogeneous deep learning framework for intelligent malware detection," *Knowledge and Information Systems*, vol. 54, no. 2, pp. 265–285, Feb 2018.

[45] G. E. Dahl, J. W. Stokes, L. Deng, and D. Yu, "Large-scale malware classification using random projections and neural networks," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, May 2013, pp. 3422–3426.

[46] G. Kim, H. Yi, J. Lee, Y. Paek, and S. Yoon, "LSTM-Based System-Call Language Modeling and Robust Ensemble Method for Designing Host-Based Intrusion Detection Systems," 2016. [Online]. Available: http://arxiv.org/abs/1611.01726

[47] B. Athiwaratkun and J. W. Stokes, "Malware classification with LSTM and GRU language models and a character-level CNN," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, March 2017, pp. 2482–2486.

[48] H. Huang, C. Yu, and H. Kao, "R2-D2: ColoR-inspired Convolutional NeuRal Network (CNN)-based AndroiD Malware Detections," 2017. [Online]. Available: http://arxiv.org/abs/1705.04448

[49] D. Gibert, "Convolutional neural networks for malware classification," Ph.D. dissertation, MS Thesis, Dept. of Computer Science, Escola Politcnica de Catalunya, 2016.

[50] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2016, pp. 855–864.

[51] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei, "Line: Large-scale information network embedding," in *Proceedings of the 24th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2015, pp. 1067–1077.

[52] T. N. Kipf and M. Welling, "Semi-Supervised Classification with Graph Convolutional Networks," 2016. [Online]. Available: http://arxiv.org/abs/1609.02907

[53] T. Lei, W. Jin, R. Barzilay, and T. S. Jaakkola, "Deriving Neural Architectures from Sequence and Graph Kernels," 2017. [Online]. Available: http://arxiv.org/abs/1705.09037