Heterogeneous Graph Matching Networks: Application to Unknown Malware Detection

Shen Wang* *Member, IEEE*, Philip S. Yu* *Fellow, IEEE*
Department of Computer Science, University of Illinois at Chicago
Email: *swang224@uic.edu, *psyu@uic.edu

***Abstract*—Information systems have widely been the target of malware attacks. Traditional signature-based malicious program detection algorithms can only detect known malware and are prone to evasion techniques such as binary obfuscation, while behavior-based approaches highly rely on the malware training samples and incur prohibitively high training cost. To address the limitations of existing techniques, we propose MatchGNet, a heterogeneous Graph Matching Network model to learn the graph representation and similarity metric simultaneously based on the invariant graph modeling of the program's execution behaviors. We conduct a systematic evaluation of our model and show that it is accurate in detecting malicious program behavior and can help detect malware attacks with less false positives. MatchGNet outperforms the state-of-the-art algorithms in malware detection by generating 50% less false positives while keeping zero false negatives.**

I. Introduction

With information systems playing ubiquitous and indispensable roles in many modern industries including financial services and retail businesses, cyber security undoubtedly bears the utmost importance in the daily system management. Under constant cyber-attacks happening every day, incidents causing significant financial losses and leakage of sensitive customer data appear in media headlines every now and then. Among these attacks, malware/malicious program attacks are the most widespread and costly type, and they are growing fast to target at more companies and organizations. According to a study report published by Accenture[1], 98% of the participating companies experienced malware attacks in 2016 and 2017, and these attacks cost companies an average of \$2.4 million. Symantec also finds that the number of groups using destructive malware was up by 25% in 2018[2]. With imminent malware attacks, protecting key information systems has become the top priority for many companies and organizations.

Malware/malicious program detection as the first line of defense against attacks mainly uses two types of approaches: signature-based and behavior-based. Signature-based approaches [7], [9] check program signatures against a known malware database. Such signatures can be instruction sequences, system calls, loaded dynamic libraries, and so on. While signature-based approaches are widely adopted by antivirus software due to their time efficiency, they are mostly limited to detect known malware and prone to evasion techniques such as binary obfuscation. Nowadays, such signature-based approaches are facing difficulties from zero-day malware attacks, which are highly polymorphic and rarely have known signatures [7].

Behavior-based approaches [1], [2], [17], [27], [28] as an improvement over the signature-based ones learn the program behaviors in terms of dependencies between system-elements (*i.e.*, system-calls or API calls), and classify a program to either the malicious or the benign category using the learned model. The program behaviors captured by these approaches generally reflect real program intentions, thereby complementing static signatures that can easily be manipulated. On the other hand, behavior-based approaches incur prohibitively high training cost. In order to learn malicious program behavior, one must collect enough malware samples, and then execute them repeatedly in a controlled environment to observe their runtime behaviors. Insufficient sampling and observations of malicious behavior can unavoidably limit the detection capability, and malware attacks can also evolve via adversarial learning to evade detection.

In contrast to those malware detection algorithms focusing on detecting *known* malware, the goal of this paper is to design an effective *data-driven* approach for detecting *unknown* malicious programs. We define a malicious program as a process that is unknown to the execution environment and behaves differently to all existing benign programs. Slightly more formally, given a target program with corresponding process event data (*e.g.*, a program opens a file or connects to a server) during a time window, we perform similarity learning and check whether the behavior of the program is similar to that of any existing benign programs. If some highly similar programs exist, the model outputs *top-k* most similar programs with their IDs/names and similarity scores; otherwise, it triggers an alert.

It is difficult to detect unknown malicious programs due to four major challenges: (1) The nonlinear and hierarchical heterogeneous relations/dependencies among system entities. The operations made by programs have nonlinear and hierarchical heterogeneous dependencies, and they work together in a highly complex and coordinated manner. Simple methods that neglect these dependencies may still yield high false positive rates; (2) Lack of intrinsic distance/similarity metrics [43] between two programs. Consider that in real computer systems, given two programs with thousands of system events related to them, it is a non-trivial task to measure their

[1]www.accenture.com/us-en/insight-cost-of-cybercrime-2017
[2]www.symantec.com/security-center/threat-report

Authorized licensed use limited to: NEC Labs. Downloaded on September 15,2024 at 23:14:44 UTC from IEEE Xplore. Restrictions apply.

distance/similarity based on the categorical event data; (3) The exponentially large event space. An information system typically deals with a large volume of system event data (normally more than $10,000$ events per host per second) [11]; (4) Aliases of programs. Different versions or updates of the same program have different signatures and may also have different executable names. Thus, a simple method that keeps a white-list of program IDs/signatures could be problematic.

To address these challenges, we propose **MatchGNet**, a data-driven graph matching framework to learn the program representation and similarity metric via Graph Neural Network. In particular, we first design an invariant graph modeling to capture the heterogeneous interactions/dependencies between different pairs of system entities. To learn the program representation from the constructed heterogeneous invariant graph, we propose a hierarchical attentional graph neural encoder. Finally, we propose a similarity learning model via Siamese Network to train the parameters and perform similarity scoring between an unknown program and the existing benign programs. **MatchGNet** trains the model on existing benign programs, not on any malware/malicious program samples. As a result, it can identify an unknown malicious program whose behavioral representation is significantly different to any of the existing benign programs by similarity matching. We conduct an extensive set of experiments on real-world system provenance data to evaluate the performance of our model. The results demonstrate that **MatchGNet** can accurately detect malicious programs. In particular, it can detect fake and unknown programs with an average accuracy of 97% and 96%, respectively. We also apply **MatchGNet** to detect realistic malware attacks. The results show that it can reduce false positives of the state-of-the-art by 50%, while keeping zero false negative.

II. Motivating Example and Related Work

In a phishing email attack as shown in Figure 1, to steal sensitive data from the database of a computer/server, the adversary exploits a known venerability of Microsoft Office[3] by sending a phishing email attached with a malicious .doc file to one of the IT staff of the enterprise. When the IT staff member opens the attached .doc file through the browser, a piece of malicious macro is triggered. This malicious macro creates and executes a malware executable, which pretends to be an open source Java runtime (Java.exe). This malware then opens a backdoor to the adversary, subsequently allowing the adversary to read and dump data from the target database via the affected computer.

Signature-based or behavior-based malware detection approaches generally do not work well in detecting the malicious program in our example. As the adversary can make the malicious program from scratch with binary obfuscation, signature-based approaches [2], [9] would fail due to the lack of known malicious signatures. Behavior-based approaches [1], [2], [17]

[3]https:// nvd.nist.gov/vuln/detail/CVE-2008-0081

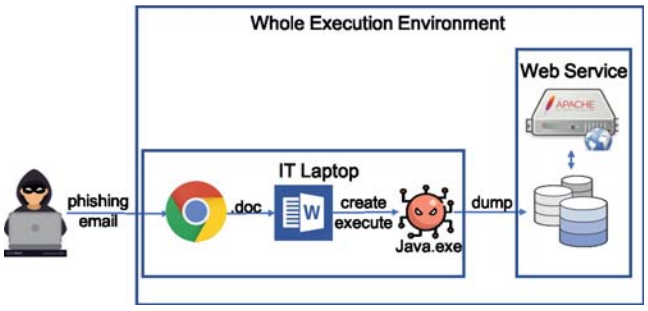


Fig. 1: An example of phishing email attack.

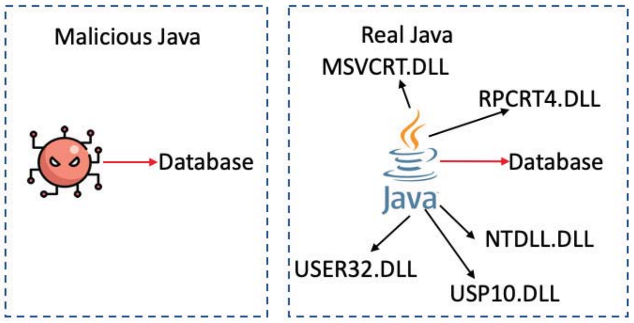


Fig. 2: Malicious Java.exe vs. normal Java runtime.

may not be effective either, unless the malware sample has previously been used to train the detection model.

It might be possible to detect the malicious program using existing host-level anomaly detection techniques [3], [5], [18], [21]. These host-based anomaly detection methods can locally extract patterns from process events as the discriminators of abnormal behavior. However, such detection is based on observations of single operations, and it sacrifices the false positive rate to detect the malicious program [22]. For example, the host-level anomaly detection can detect the fake “Java.exe” by capturing the database read. However, a Java-based SQL client may also exhibit the same operation. If we simply detect the database read, we may also classify normal Java-based SQL clients as abnormal program instances and generate false positives. In the enterprise environment, too many false positives can lead to the alert fatigue problem [15], [22], causing cyber-analysts to fail to catch up with attacks.

To accurately separate the database read of the malicious Java from the real Java instances, we need to consider the higher semantic-level context of the two Java instances. As shown in Figure 2, the malicious Java is a very simple program and directly accesses the database. On the contrary, a real Java instance has to load a set of .DLL files in addition to the database read. By comparing the context of the fake Java instance, we can find that it is not normal and precisely report it as a malicious program instance. Thus, in this paper, we propose a Graph Neural Network based approach to learn the semantic-level context of program instances.

In recent years, Graph Neural Network (GNN) approaches [4], [10], [13], [14], [20], [33], [35], [39], [41] have

Authorized licensed use limited to: NEC Labs. Downloaded on September 15,2024 at 23:14:44 UTC from IEEE Xplore. Restrictions apply.

been proposed for graph-structured data. They try to accelerate convolution operations, reduce the computational cost, and extend the current graph convolution. The goal of GNN is to learn the representation of the graph, in the node level or graph level. Because of its remarkable graph representation learning ability, GNN has been explored in various real-world applications, such as healthcare [24], [36], chemistry [12], [44], and security system [34].

In parallel to the Graph Neural Network, graph similarity matching has been studied extensively in the machine learning community [26], [29], [38], [40]. The similarity metric can be categorized into two types: exact matching (isomorphism test) [29], [40] and structure similarity measures (graph edit distance) [26], [38]. Besides, in the computer vision community, learning-based metrics [30], [37] are proposed to match the image data. They compute the similarity score with either hand-engineered features or hand-designed metrics. Compared to these approaches, our work is different in two folds: 1) we focus on learning from the heterogeneous graph, rather than the homogeneous graph; and 2) we learn the graph representation and similarity metric simultaneously.

III. Heterogeneous Graph Matching Networks

A. Overview

To address the challenges introduced in Section I, we propose a heterogeneous Graph Matching Network framework, **MatchGNet**, with three key modules: Invariant Graph Modeling (**IGM** or Step (A)), Hierarchical Attentional Graph Neural Encoder (**HAGNE** or Step (B)), and Similarity Learning (**SL** or Step (C)) as illustrated in Figure 3. The **IGM** module models the system event data as a heterogeneous invariant graph, which can capture the program's behavior profile. Then, we formulate the malicious program detection as a heterogeneous graph matching problem and solve it with **HAGNE** and **SL**.

Given two graphs $G_{(1)} = (V_{(1)}, E_{(1)})$ and $G_{(2)} = (V_{(2)}, E_{(2)})$, **HAGNE** first generates corresponding graph embeddings $\mathbf{h}_{G_{(1)}}$ and $\mathbf{h}_{G_{(2)}}$ in a hierarchical attentional style and then fuses the two graph embeddings via Similarity Learning (**SL**) and outputs a similarity score $Sim(G_{(1)}, G_{(2)})$. In this way, the representation of the graph and similarity metric can be learned in a joint way, such that the effective graph similarity matching can be performed. During the malware detection stage, the distance between an unknown program and an existing benign program will be maximized in the mapped embedding space under the learned similarity metric.

B. Invariant Graph Modeling

Information systems often generate a large volume of system-event data (*i.e.*, the interaction between a pair of system entities). In a typical enterprise environment, the amount of data collected from a single computer system can easily reach one gigabyte after monitoring process interactions for one hour. Learning a representation over such massive data is prohibitively expensive in terms of both time and space. Recently, a very promising means for studying complex systems has emerged through the concept of invariant graph [6], [23]. Such invariant graph focuses on discovering stable and significant dependencies between pairs of entities that are monitored through surveillance data recordings, so as to profile the system status and perform subsequent reasoning.

Following the idea of the invariant graph, we model the system event data as a heterogeneous graph between different system entities (*e.g.*, processes, files, and Internet sockets). The edges indicate the causal dependencies including a process accessing a file, a process forking another process, and a process connecting to an Internet socket. Formally, given the event data U across several machines within a time window (*e.g.*, one day), each target program can be a heterogeneous graph $G = (V, E)$, in which V denotes a set of nodes. Each node represents an entity of three possible types: process (P), file (F), and INETSocket (I), namely $V = P \cup F \cup I$. E denotes a set of edges (dependencies) (v_s, v_d, r) between the source entity v_s and destination entity v_d with relation r. We consider three types of relations: (1) a process forking another process ($P \rightarrow P$), (2) a process accessing a file ($P \rightarrow F$), and (3) a process connecting to an Internet socket ($P \rightarrow I$). Each graph is associated with an adjacency matrix A. With the help of the invariant graph modeling, we can obtain a global program-dependency profile.

C. Hierarchical Attentional Graph Neural Encoder

The constructed invariant graph is heterogeneous with multiple types of entities and relations. Thus, it is difficult to directly apply the traditional homogeneous Graph Neural Networks (such as GCN and GraphSage) to learn the graph representation. To address this problem, we propose a Hierarchical Attentional Graph Neural Encoder (**HAGNE**) to learn the program representation as a graph embedding through an attentional architecture by considering the node-wise, layer-wise, and path-wise context importance. More specifically, we first propose a **Heterogeneity-aware Contextual Search** (Step (B1) in Figure 3) to find the path-relevant sets of neighbors under the guide of the meta-paths [31]. Then, we introduce a **Node-wise Attentional Neural Aggregator** (Step (B2) in Figure 3) to generate node embeddings by selectively aggregating the entities in each path-relevant neighbor set based on random walk scores. Next, we design a **Layer-wise Dense-connected Neural Aggregator** (Step (B3) in Figure 3) to aggregate the node embeddings generated from different layers towards a dense-connected node embedding. Finally, we develop a **Path-wise Attentional Neural Aggregator** (Step (B4) in Figure 3) to learn the attentional weights for different meta-paths and compute the graph embedding from the Layer-wise Dense-connected Aggregator.

1) Heterogeneity-aware Contextual Search: As the first step of the aggregation layer, traditional GNNs would search for all one-hop neighbors for a target node. Since our invariant graph is heterogeneous, simply aggregating these neighbors cannot capture the semantic and structural correlations among different types of entities. To address this issue, we propose a meta-path [31] based contextual search. A meta-path is a path that connects different entity types via a sequence of relations

Authorized licensed use limited to: NEC Labs. Downloaded on September 15,2024 at 23:14:44 UTC from IEEE Xplore. Restrictions apply.

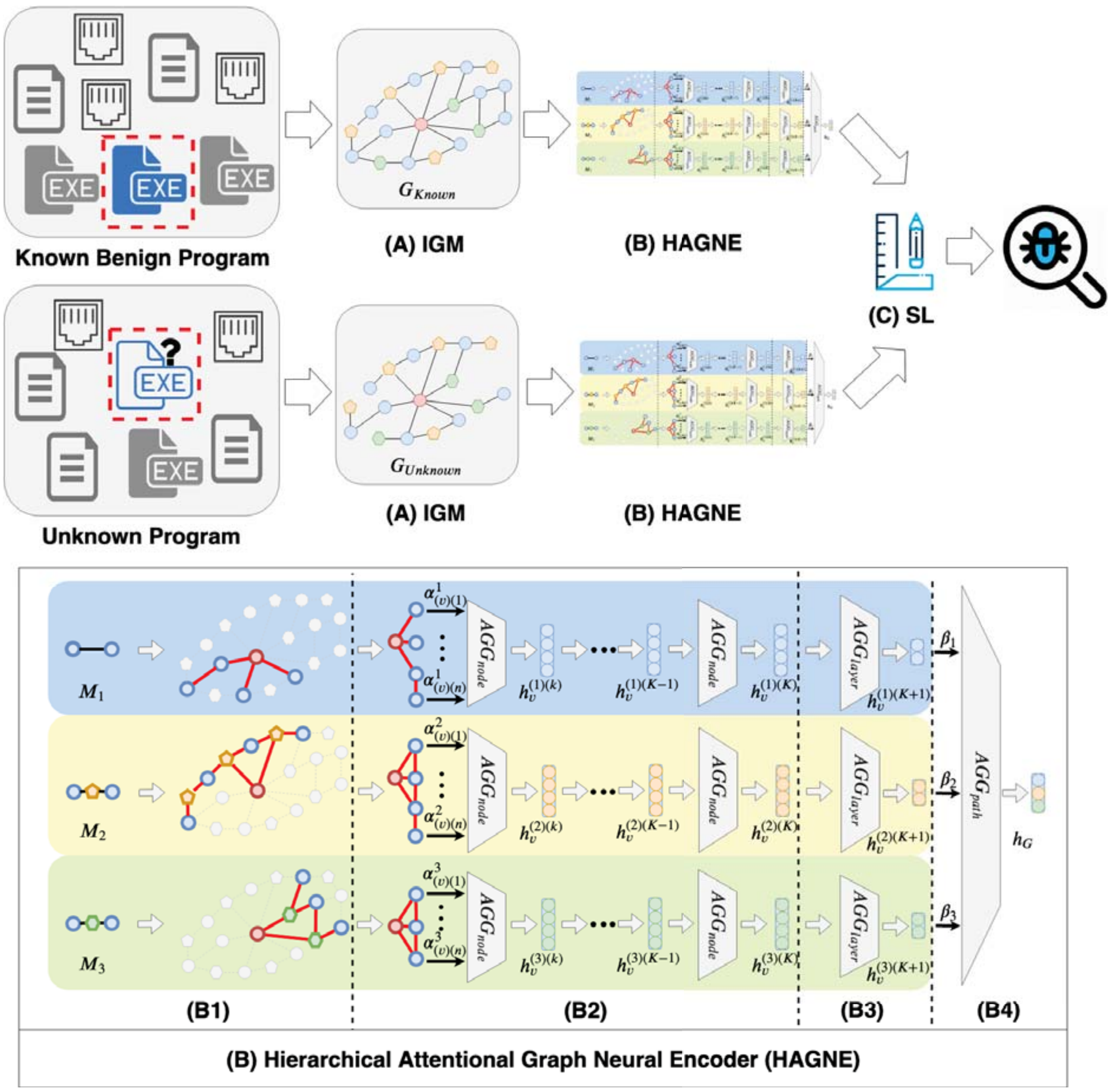


Fig. 3: The architecture of MatchGNet consists of three key modules: (A) Invariant Graph Modeling (IGM), (B) Hierarchical Attentional Graph Neural Encoder (HAGNE), and (C) Similarity Learning (SL). IGM models system event data as a heterogeneous invariant graph. HAGNE encodes the heterogeneous graph into an embedding by four components: (B1) Heterogeneity-aware Contextual Search, (B2) Node-wise Attentional Neural Aggregator, (B3) Layer-wise Dense-connected Neural Aggregator, and (B4) Path-wise Attentional Neural Aggregator. SL trains HAGNE to be more distinguishable between an unknown program and known benign programs and computes the similarity score for unknown malware detection.

in a heterogeneous graph. In a computer system, a meta-path could be: a process forking another process ($P \rightarrow P$), two processes accessing the same file ($P \leftarrow F \rightarrow P$), or two processes opening the same internet socket ($P \leftarrow I \rightarrow P$) with each one defining a unique relationship between two programs. From the invariant graph G, a set of meta-paths $\mathcal{M} = \{M_1, M_2, ...M_{|\mathcal{M}|}\}$ can be generated with each representing a unique multi-hop relationship between two programs. For each meta-path M_i where $i \in \{1, 2, ..., |\mathcal{M}|\}$, we define the path-relevant neighbor set $\mathcal{N}_v^i$ of node v:

$$\mathcal{N}_v^i = \{u|(u,v) \in M_i, (v,u) \in M_i\} \tag{1}$$

where u is a reachable neighbor of v via the meta-path M_i.

2) Node-wise Attentional Neural Aggregator: After constructing the path-relevant neighbor set $\mathcal{N}_v^i$, we are able to leverage these contexts via neighbourhood aggregation. However, due to noisy neighbors, different neighboring nodes may have different impacts on the target node. Hence, it is unreasonable to treat all neighbors equally. To address this issue, we propose a node-wise attentional neural aggregator

Authorized licensed use limited to: NEC Labs. Downloaded on September 15,2024 at 23:14:44 UTC from IEEE Xplore. Restrictions apply.

to compute an attentional weight for each node in the path-relevant neighbor set $\mathcal{N}_v^i$. This module is based on random walk with restarts (RWR) [32], a particularly efficient algorithm for computing the relevance scores between pairs of nodes in a homogeneous graph. We extend the RWR to a heterogeneous graph, such that the walker starts at the target program node v, and at each step it only moves to one of its neighboring nodes in $\mathcal{N}_v^i$, instead of to all linked nodes without considering semantics. After the random walk finishes, each visited neighbor will receive a visiting count. We compute the L_1 normalization of the visiting count and use it as the node-wise attentional weight. Formally, for $\mathcal{N}_v^i = \{u_1^i, ..., u_n^i\}$, the attentional weights are $\boldsymbol{\alpha}_{(v)(:)}^i = [\alpha_{(v)(1)}^i, ..., \alpha_{(v)(n)}^i]$, where $\alpha_{(v)(j)}^i$ is the weight of u_j^i. Then, the program representation can be computed via a neural aggregation function AGG_{node} by

$$\begin{aligned}\mathbf{h}_v^{(i)(k)} &= AGG_{node}(\mathbf{h}_v^{(i)(k-1)}, \{\mathbf{h}_u^{(i)(k-1)}\}_{u \in \mathcal{N}_v^i}) \\ &= MLP^{(k)}((1+\epsilon^{(k)})\mathbf{h}_v^{(i)(k-1)} + \sum_{u \in \mathcal{N}_v^i} \boldsymbol{\alpha}_{(u)(:)}^i \mathbf{h}_u^{(i)(k-1)}) \qquad (2)\end{aligned}$$

where $k \in \{1, 2, ...K\}$ denotes the index of the layer, $\mathbf{h}_v^{(i)(k)}$ is the feature vector of program v for meta-path M_i at the k-th layer, and $\epsilon^{(k)}$ is a trainable parameter that quantifies the trade-off between the previous layer representation and the aggregated contextual representation. $\mathbf{h}_u^{(i)(0)}$ is initialized by X_v. After getting the aggregated representation, a multi-layer perceptron (MLP) is applied to transform the aggregated representation to a hidden nonlinear space. Through the development of Node-wise Attentional Neural Aggregator, we are able to leverage the contextual information, meanwhile considering the different importance of the neighbors.

3) Layer-wise Dense-connected Neural Aggregator: To aggregate the information from a wider range of neighbors, one simple way is to stack multiple node-wise neural aggregators. However, the performance of a GNN model often cannot get improved, because by adding more layers, it is easy to propagate the noisy information from an exponentially increasing number of neighbors in a deep layer. Recently, inspired by residual network, a **skip-connection** method has been proposed. But, even with skip-connections, GCNs with more layers do not perform as well as the 2-layer GCN on many graph data [20]. To address the limitations of existing work, we propose a Layer-wise Dense-connected Aggregator as inspired by the DENSENET [16]. More specifically, our layer-wise aggregator leverages all the intermediate representations, with each capturing a subgraph structure. All the intermediate representations are aggregated in a concatenation way followed by a MLP, such that the resulted embedding can adaptively select different subgraph structures. Formally, the Layer-wise Dense-connected Neural Aggregator $AGG_{layer}()$ can be constructed as follows:

$$\begin{aligned}\mathbf{h}_v^{(i)(K+1)} &= AGG_{layer}(\mathbf{h}_v^{(i)(0)}, \mathbf{h}_v^{(i)(1)}, ...\mathbf{h}_v^{(i)(K)}) \qquad (3) \\ &= MLP([\mathbf{h}_v^{(i)(0)}; \mathbf{h}_v^{(i)(1)}; ...\mathbf{h}_v^{(i)(K)}]) \qquad (4)\end{aligned}$$

where $[\cdot;\cdot]$ represents the feature concatenation operation.

4) Path-wise Attentional Neural Aggregator: After the node-wise and layer-wise aggregators, different embeddings corresponding to different meta-paths are generated. However, different meta-paths should not be treated equally. For example, Ransomware is usually very active in accessing files, but it barely forks another process or opens an internet socket, while VPNFilter is generally very active in opening the internet socket, but it barely accesses a file or forks another process. Therefore, we need to treat different meta-paths differently. To address this issue, we propose a Path-wise Attentional Neural Aggregator to aggregate the embeddings generated from different path-relevant neighbor sets. Our path-wise aggregator can learn the attentional weights for different meta-paths automatically. Formally, given the program embedding $\mathbf{h}_v^{(i)(K+1)}$ corresponding to each meta-path M_i, we define the path-wise attentional weight as follows:

$$\beta_i = \frac{\exp(\sigma(\mathbf{b}[\mathbf{W}_b\mathbf{h}_v^{(i)(K+1)}||\mathbf{W}_b\mathbf{h}_v^{(j)(K+1)}]))}{\sum_{j' \in |\mathcal{M}|}\exp(\sigma(\mathbf{b}[\mathbf{W}_b\mathbf{h}_v^{(i)(K+1)}||\mathbf{W}_b\mathbf{h}_v^{(j')(K+1)}]))} \qquad (5)$$

where $\mathbf{h}_v^{(i)(K+1)}$ is the embedding corresponding to the target meta-path M_i, $\mathbf{h}_v^{(j)(K+1)}$ denotes the embedding corresponding to the other meta-path M_j, $\mathbf{b}$ denotes a trainable attention vector, $\mathbf{W}_b$ denotes a trainable weight matrix, which maps the input features to the hidden space, $||$ denotes the concatenation operation, and σ denotes the nonlinear gating function. We formulate a feed-forward neural network, which computes the correlation between one path-relevant neighbor set and other path-relevant neighbor sets. This correlation is normalized by a Softmax function. Let $ATT()$ represent Eq.(5). The joint representation for all the meta-paths can be represented as follows:

$$\mathbf{h}_G = AGG_{path} = \sum_{i=1}^{|\mathcal{M}|} ATT(\mathbf{h}_v^{(i)(K+1)})\mathbf{h}_v^{(i)(K+1)} \qquad (6)$$

The Path-wise Attentional Neural Aggregator allows us to better infer the importance of different meta-paths by leveraging their correlations and learn a path-aware representation.

D. Similarity Learning

In order to perform effective graph matching, we propose Similarity Learning (**SL**), a Siamese Network based learning model to train the parameters of the Hierarchical Attentional Graph Neural Encoder (**HAGNE**). Siamese Networks [42] are neural networks containing two or more identical subnetwork components, which have been shown to be a powerful way in distinguishing similar and dissimilar objects. Here, we employ the **SL** to learn similarity metric and program graph representation jointly for better graph matching between the pair of unknown program and known benign program as shown in Figure 3.

More specifically, our Siamese Network consists of two identical **HAGNE**s to compute the program graph representation independently. Each **HAGNE** takes a program graph snapshot as the input and outputs the corresponding embedding $\mathbf{h}_G$. A neural network is then used to fuse the two

Authorized licensed use limited to: NEC Labs. Downloaded on September 15,2024 at 23:14:44 UTC from IEEE Xplore. Restrictions apply.

embeddings. The final output is the similarity score of the two program embeddings. During the training, P pairs of program graph snapshots $(G_{i(1)}, G_{i(2)}), i \in \{1, 2, ...P\}$ are collected with corresponding ground truth pairing information $y_i \in \{+1, -1\}$. If the pair of graph snapshots belong to the same program, the ground truth label is $y_i = +1$, otherwise its ground truth label is $y_i = -1$. For each pair of program snapshots, a cosine score function is used to measure the similarity of the two program embeddings and the output of the Similarity Learning is defined as follows:

$$Sim(G_{i(1)}, G_{i(2)}) = cos((\mathbf{h}_{G_{i(1)}}, \mathbf{h}_{G_{i(2)}})) \tag{7}$$

$$= \frac{\mathbf{h}_{G_{i(1)}} \cdot \mathbf{h}_{G_{i(2)}}}{||\mathbf{h}_{G_{i(1)}}|| \cdot ||\mathbf{h}_{G_{i(2)}}||} \tag{8}$$

Correspondingly, our objective function can be formulated as:

$$\ell = \sum_{i=1}^{P} (Sim(G_{i(1)}, G_{i(2)}) - y_i)^2 \tag{9}$$

We optimize this objective with Adam optimizer [19]. With the help of the Similarity Learning, we can learn the parameters that keep similar embeddings closer while pushing dissimilar embeddings apart by directly optimizing the embedding distance.

Since we directly optimize the distance between the two programs, this model can be used to perform unknown malware detection. Given the snapshot of an unknown program, we first construct its corresponding program invariant graph and then feed it to the **HAGNE** to generate the program embedding. After that, we compute the cosine distance scores between the embedding of the unknown program and the ones of the existing programs in the database. If an existing program has more than one embedding generated from multiple graph snapshots, we will only report the highest similarity score with regard to the unknown program. Finally, we rank all the similarity scores. If the highest similarity score among all the existing programs is below our threshold, an alert will be triggered. Otherwise, the top-k most similar programs will be reported.

IV. Experiments

A. Experiment Setup

1) Data: We collect a 20-week period of data from a real enterprise network composed of 109 hosts (87 Windows hosts and 22 Linux hosts). The sheer size of the data set is around three terabytes. We consider three different types of system events as defined in Section III-B. Each entity is associated with a set of attributes, and each process has an executable name as its identifier (ID). In total, there are about 300 million event records, with about $2,000$ processes, $600,000$ files, and $18,000$ Internet sockets. Based on the system event data, we construct a program invariant graph per program per day.

2) Baselines: We compare **MatchGNet** with the following typical and state-of-the-art algorithms:

- *LR* and *SVM*: LR and SVM represent the Logistic regression and Linear Support Vector Machine, respectively. They are two typical classification methods. We extract raw features from each process as the input, including the connectivity features and the graph statistic features.
- *MLP* [25]: Multi-layer Perceptron (MLP) is a deep neural network based classification model with multiple non-linear layers between the input and the output layers. It is a special case of GNN without considering the aggregation operation if we define the propagation layer as an identity matrix.

Since the proposed **MatchGNet** is based on GNN, we also compare it with two popular GNN variants: GCN [20] and GraphSage [14].

3) Evaluation Metrics: Similar to [8], we evaluate the performance of different methods using accuracy (ACC), F-1 score, and AUC score.

B. Fake Program Detection

Our first research question focuses on the accuracy of **MatchGNet** detecting fake programs. Here, we define a fake program as the one that uses the ID of another program. It is a common method for adversaries to hide their attacks.

To simulate the execution of fake programs on a large scale, in our testing dataset (*i.e.*, data from the seventh week), we manually seed $1,000$ fake programs. To do so, before feeding the monitoring data to **MatchGNet**, we randomly replace the ID of a known program to the ID of another known program. This process simulates an adversary who wants to hide the use of a benign system tool in his/her attacks.

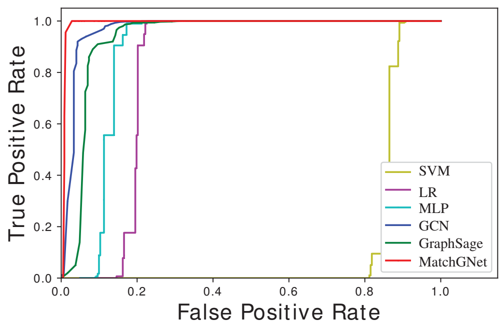


Fig. 4: ROC curves on detecting fake programs.

Method	SVM	LR	MLP	GCN	GraphSage	MatchGNet
AUC	51.11	82.79	85.11	94.53	91.72	**98.51**
F-1	66.07	83.05	86.12	93.63	90.02	**98.49**
ACC	50.45	80.66	86.82	94.87	92.03	**97.03**

TABLE I: Performance on fake program detection.

Method	SVM	LR	MLP	GCN	GraphSage	MatchGNet
AUC	51.11	80.63	83.69	92.05	90.38	**96.08**
F-1	66.07	82.92	84.37	92.03	90.43	**95.55**
ACC	50.01	81.98	85.00	92.24	91.50	**96.53**

TABLE II: Performance on unknown program detection.

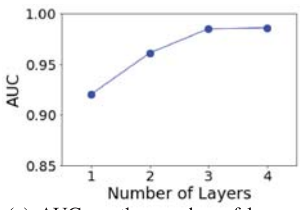
In this task, each fake program has a claimed ID of a known program. Thus, we only need to check whether it is indeed

Authorized licensed use limited to: NEC Labs. Downloaded on September 15,2024 at 23:14:44 UTC from IEEE Xplore. Restrictions apply.

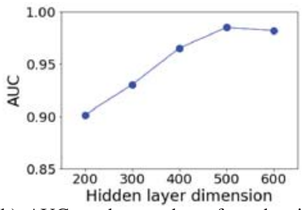
the claimed program: if it matches the behavior pattern of the claimed program, the predicted label should be -1; otherwise, it should be $+1$. We use logistic regression as the classification model for all the methods.

The ROC curve is shown in Figure 4. We also summarize the AUC, F-1 score and ACC of the six models in Section IV-B. From our experiments, the AUC of **MatchGNet** is 99%, which is at least 4% higher than all other baseline models. In terms of ACC, **MatchGNet** could achieve 47%, 16%, 10%, 2%, and 5% higher than the SVM, LR, MLP, GCN, and GraphSage models. This means that while capturing all the fake program instances, **MatchGNet** has less false positives than all other models. This result justifies our design decision of applying the invariant graph structure and Graph Neural Network model to capture the semantics of program instances.

1) Hyper Parameter Selection: We evaluate the selection of hyper parameters of **MatchGNet** with our validating data set (*i.e.,* data from the sixth week). There are two hyper parameters in **MatchGNet**: the number of layers, and the number of hidden neurons. We plot the result for the number of layers and the number of hidden neurons in Figure 5. In these figures, the y-axis is the AUC value and the x-axis is the value of the hyper parameter.



(a) AUC vs. the number of layers



(b) AUC vs the number of stochastic contexts

Fig. 5: Hyper parameters in fake program detection.

We find that when **MatchGNet** has 3 layers and 500 neurons, it reaches the maximal AUC. Larger hyper-parameter values may consume more resources but have little improvement on the AUC. Thus, we use the optimal hyper parameters as a part of the default model and apply them to the other parts of our experiments.

C. Unknown Program Detection

This experiment focuses on evaluating **MatchGNet** on unknown program detection. To simulate unknown program instances, we split the programs in the training data equally into two sets, the known set and the unknown set. In our five weeks' training data, we exclude the programs in the unknown set and only train the model from the programs in the known set. Then, in our testing period, we use program instances from the unknown set as malicious programs.

We plot the ROC curves in Figure 6 and report the AUC, F-1 score, and ACC of all six models in Section IV-B. The AUC of **MatchGNet** in detecting unknown program instances is at least 4% higher than all other models. And in terms of ACC, **MatchGNet** is 46%, 14%, 11%, 4%, and 5% higher than

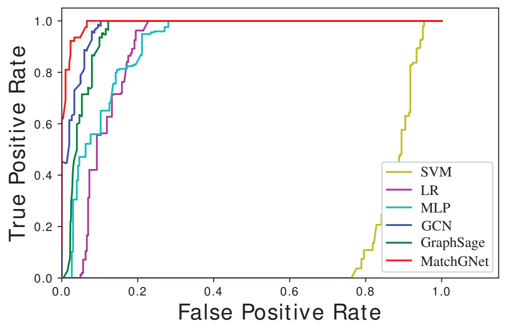


Fig. 6: ROC curves on detecting unknown programs.

the SVM, LR, MLP, GCN, and GraphSage models. This also proves that using the Attentional Graph Neural Network model could better capture the semantic level features of programs and may possibly generate less false positives.

D. Malware Attack Detection

To evaluate the usefulness of **MatchGNet**, we apply it to detect realistic malware attacks in enterprise environment. We randomly download 145 malware from VirusTotal covering all the popular malware categories and create 9 realistic attack cases from the literature and report including WannaCry, Genasom, Sorikrypt, Shinolocker, Puishing Email, ShellShock, Netcat Backdoor, Passing the Hash, and Trojan Attacks. We execute these attacks on our two testing machines and merge the provenance data from these two testing machines with the normal data of the 109 hosts to the data stream of **MatchGNet**. We set up **MatchGNet** with the optimal hyper parameters in Section IV-B. To evaluate the usefulness of **MatchGNet**, we use the state-of-the-art entity-embedding based technique [5] as the baseline.

Based on the experimental results, both **MatchGNet** and the baseline method capture all the 154 true alerts. However, during the same time period and with the same provenance data, **MatchGNet** only generates 57 false positives while the baseline generates 115 false positives. Such a number indicates that **MatchGNet** is possible to reduce nearly 50% of the false positives. This reduction could mean a substantial saving in cyber-analysts' time in a large enterprise.

V. Conclusion

In this paper, we proposed **MatchGNet**, a heterogeneous Graph Matching Network approach to detect the unknown malicious programs in information systems. **MatchGNet** first models the program's execution behavior as a heterogeneous invariant graph. Based on the built program graphs, it learns the graph representation and similarity metric simultaneously to distinguish the benign program and malware. The evaluation results showed that our approach can accurately detect unexpected program instances. In particular, it can detect fake and unknown programs with an average accuracy of 97% and 96%, respectively. We further demonstrate our approach is promising by having 50% less false positives than the state-of-the-art method in detecting malware attacks.

Authorized licensed use limited to: NEC Labs. Downloaded on September 15,2024 at 23:14:44 UTC from IEEE Xplore. Restrictions apply.

References

[1] Domagoj Babić, Daniel Reynaud, and Dawn Song. Malware analysis with tree automata inference. In *Computer Aided Verification*, pages 116–131, 2011.

[2] Mario Luca Bernardi, Marta Cimitile, Damiano Distante, Fabio Martinelli, and Francesco Mercaldo. Dynamic malware detection and phylogeny analysis using process mining. *International Journal of Information Security*, Jun 2018.

[3] Cheng Cao, Zhengzhang Chen, James Caverlee, Lu-An Tang, Chen Luo, and Zhichun Li. Behavior-based community detection: Application to host assessment in enterprise information networks. In *CIKM*, pages 1977–1985, 2018.

[4] Jie Chen, Tengfei Ma, and Cao Xiao. Fastgcn: fast learning with graph convolutional networks via importance sampling. *arXiv preprint arXiv:1801.10247*, 2018.

[5] Ting Chen, Lu-An Tang, Yizhou Sun, Zhengzhang Chen, and Kai Zhang. Entity embedding-based anomaly detection for heterogeneous categorical events. In *IJCAI*, 2016.

[6] Wei Cheng, Kai Zhang, Haifeng Chen, Guofei Jiang, Zhengzhang Chen, and Wei Wang. Ranking causal anomalies via temporal and dynamical analysis on vanishing correlations. In *SIGKDD*, pages 805–814. ACM, 2016.

[7] Anusha Damodaran, Fabio Di Troia, Corrado Aaron Visaggio, Thomas H. Austin, and Mark Stamp. A comparison of static, dynamic, and hybrid analysis for malware detection. *Journal of Computer Virology and Hacking Techniques*, 13(1):1–12, Feb 2017.

[8] Kaustav Das and Jeff Schneider. Detecting anomalous records in categorical datasets. In *SIGKDD*, pages 220–229, 2007.

[9] Baptiste David, Eric Filiol, and Kévin Gallienne. Structural analysis of binary executable headers for malware detection optimization. *Journal of Computer Virology and Hacking Techniques*, 13(2):87–93, May 2017.

[10] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *NIPS*, pages 3844–3852, 2016.

[11] Boxiang Dong, Zhengzhang Chen, Hui (Wendy) Wang, Lu-An Tang, Kai Zhang, Ying Lin, Zhichun Li, and Haifeng Chen. Efficient discovery of abnormal event sequences in enterprise security systems. In *CIKM*, pages 707–715, 2017.

[12] Alex Fout, Jonathon Byrd, Basir Shariat, and Asa Ben-Hur. Protein interface prediction using graph convolutional networks. In *NIPS*, pages 6530–6539, 2017.

[13] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. *arXiv preprint arXiv:1704.01212*, 2017.

[14] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *NIPS*, pages 1024–1034, 2017.

[15] R. Harang and A. Kott. Burstiness of intrusion detection process: Empirical evidence and a modeling approach. *IEEE Transactions on Information Forensics and Security*, 2017.

[16] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *CVPR*, pages 4700–4708, 2017.

[17] S. Jha, M. Fredrikson, M. Christodoresu, R. Sailer, and X. Yan. Synthesizing near-optimal malware specifications from suspicious behaviors. In *2013 8th International Conference on Malicious and Unwanted Software: "The Americas" (MALWARE)*, pages 41–50, Oct 2013.

[18] V Jyothsna and V V Rama Prasad. A review of anomaly based intrusion detection systems. *International Journal of Computer Applications*, 28(7):26–35, 2011.

[19] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[20] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[21] Shih-Wei Lin, Kuo-Ching Ying, Chou-Yuan Lee, and Zne-Jung Lee. An intelligent algorithm with feature selection and decision rules applied to anomaly intrusion detection. *Applied Soft Computing*, 12(10):3285–3290, 2012.

[22] Ying Lin, Zhengzhang Chen, Cheng Cao, Lu-An Tang, Kai Zhang, Wei Cheng, and Zhichun Li. Collaborative alert ranking for anomaly detection. In *CIKM*, pages 1987–1995, 2018.

[23] Chen Luo, Zhengzhang Chen, and *et al.* TINET: learning invariant networks via knowledge transfer. In *SIGKDD*, pages 1890–1899, 2018.

[24] Chengsheng Mao, Liang Yao, and Yuan Luo. Medgcn: Graph convolutional networks for multiple medical tasks. *arXiv preprint arXiv:1904.00326*, 2019.

[25] Sankar K Pal and Sushmita Mitra. Multilayer perceptron, fuzzy sets, and classification. *IEEE Transactions on neural networks*, 3(5):683–697, 1992.

[26] John W Raymond, Eleanor J Gardiner, and Peter Willett. Rascal: Calculation of graph similarity using maximum common edge subgraphs. *The Computer Journal*, 45(6):631–644, 2002.

[27] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. Learning and classification of malware behavior. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 108–125, 2008.

[28] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli. Madam: Effective and efficient behavior-based android malware detection and prevention. *IEEE Transactions on Dependable and Secure Computing*, 15(1):83–97, Jan 2018.

[29] Dennis Shasha, Jason TL Wang, and Rosalba Giugno. Algorithmics and applications of tree and graph searching. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 39–52. ACM, 2002.

[30] Yi Sun, Yuheng Chen, Xiaogang Wang, and Xiaoou Tang. Deep learning face representation by joint identification-verification. In *NIPS*, pages 1988–1996, 2014.

[31] Yizhou Sun, Jiawei Han, and *et al.* Pathsim: Meta path-based top-k similarity search in heterogeneous information networks. *VLDB Endowment*, 2011.

[32] Hanghang Tong, Christos Faloutsos, and Jia-Yu Pan. Fast random walk with restart and its applications. In *ICDM*, pages 613–622, 2006.

[33] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. *CoRR*, abs/1710.10903(2), 2017.

[34] Shen Wang, Zhengzhang Chen, Ding Li, Zhichun Li, Lu-An Tang, Jingchao Ni, Junghwan Rhee, Haifeng Chen, and Philip S Yu. Attentional heterogeneous graph neural network: Application to program reidentification. In *SDM*, pages 693–701, 2019.

[35] Shen Wang, Zhengzhang Chen, Jingchao Ni, Xiao Yu, Zhichun Li, Haifeng Chen, and Philip S Yu. Adversarial defense framework for graph neural network. *arXiv preprint arXiv:1905.03679*, 2019.

[36] Shen Wang, Lifang He, Bokai Cao, Chun-Ta Lu, Philip S. Yu, and Ann B. Ragin. Structural deep brain network mining. In *SIGKDD*, pages 475–484, 2017.

[37] Kilian Q Weinberger and Lawrence K Saul. Distance metric learning for large margin nearest neighbor classification. *Journal of Machine Learning Research*, 10(Feb):207–244, 2009.

[38] Peter Willett, John M Barnard, and Geoffrey M Downs. Chemical similarity searching. *Journal of chemical information and computer sciences*, 38(6):983–996, 1998.

[39] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S Yu. A comprehensive survey on graph neural networks. *arXiv preprint arXiv:1901.00596*, 2019.

[40] Xifeng Yan, Philip S Yu, and Jiawei Han. Graph indexing: a frequent structure-based approach. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 335–346. ACM, 2004.

[41] Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L Hamilton, and Jure Leskovec. Hierarchical graph representation learning withdifferentiable pooling. *arXiv preprint arXiv:1806.08804*, 2018.

[42] Sergey Zagoruyko and Nikos Komodakis. Learning to compare image patches via convolutional neural networks. In *CVPR*, pages 4353–4361, 2015.

[43] Kai Zhang, Qiaojun Wang, Zhengzhang Chen, Ivan Marsic, Vipin Kumar, Guofei Jiang, and Jie Zhang. From categorical to numerical: Multiple transitive distance learning and embedding. In *SDM*, pages 46–54, 2015.

[44] Marinka Zitnik, Monica Agrawal, and Jure Leskovec. Modeling polypharmacy side effects with graph convolutional networks. *Bioinformatics*, 34(13):i457–i466, 2018.

Authorized licensed use limited to: NEC Labs. Downloaded on September 15,2024 at 23:14:44 UTC from IEEE Xplore. Restrictions apply.