Attentional Heterogeneous Graph Neural Network: Application to Program Reidentification

Shen Wang[*][†] Zhengzhang Chen[‡][§] Ding Li[†] Zhichun Li[†] Lu-An Tang[†]
Jingchao Ni[†] Junghwan Rhee[†] Haifeng Chen[†] Philip S. Yu[*]

Abstract

Program or process is an integral part of almost every IT/OT system. Can we trust the identity/ID (*e.g.*, executable name) of the program? To avoid detection, malware may disguise itself using the ID of a legitimate program, and a system tool (*e.g.*, PowerShell) used by the attackers may have the fake ID of another common software, which is less sensitive. However, existing intrusion detection techniques often overlook this critical *program reidentification* problem (*i.e.*, checking the program's identity). In this paper, we propose an attentional heterogeneous graph neural network model (**DeepHGNN**) to verify the program's identity based on its system behaviors. The key idea is to leverage the representation learning of the heterogeneous program behavior graph to guide the reidentification process. We formulate the program reidentification as a graph classification problem and develop an effective attentional heterogeneous graph embedding algorithm to solve it. Extensive experiments — using real-world enterprise monitoring data and real attacks — demonstrate the effectiveness of **DeepHGNN** across multiple popular metrics and the robustness to the normal dynamic changes like program version upgrades.

1 Introduction

Modern enterprises often rely on intrusion detection system (IDS), either misuse-based or anomaly detection based, to protect their IT and OT systems. However, existing IDS techniques overlook one critical problem, which is the *program reidentification*: given a program, with a claimed ID (such as executable name), running in the system, can we confirm the program's identity and guarantee that this program is not a disguised malicious program, or a hijacked program with different behaviors, by comparing it to the normal program with the same ID? To make the attack processes stealthy and avoid detection, hackers often falsify the IDs of the tools they have used to bypass the IDS. For example, a malware may have the ID of a benign software and a system tool (*e.g.*, PowerShell) used by the hackers may have the ID of another software which is less sensitive [17]. In fact, running programs with fake IDs is a strong signal of the system being compromised [11]. Capturing the programs with fake IDs can help identifying very stealthy attacks and reduce the security risks in enterprise networks.

Existing techniques cannot be directly applied to address the problem of *program reidentification*. Digital code signing techniques, such as Public Key Infrastructure (PKI) [13], may help identify the certified authors of programs. However, many open-source programs that are widely used in enterprises may not contain valid signatures. Further, modern programs are evolving at a fast pace. Each version of the program has a unique signature. Handling the fast inflating set of signatures for programs is practically difficult. Malware detection or Anti-Virus may detect malware. Yet, hackers may also use common system tools to finish their attacks, such as malware free attacks [25]. Besides, a sophisticated malware can also bypass the anti-virus by hiding its malicious features, and a hacker can also hijack the memory of a benign program to perform malicious actions [3]. There are more sophisticated program watermarking techniques to identify a program [19], but their computational costs are prohibitively high so that they cannot be widely applied to modern enterprise environments, which contain thousands of programs.

We observe that the system behaviors, such as file accessing, inter-process communications, and network communications, of a program have distinguishable patterns [21]. For example, every instance of excel.exe loads a fixed set of .DLL files while opening a spreadsheet file. If an EXCEL.EXE instance performs a rare operation, such as loading an unseen .DLL file or initiating another process, this excel.exe is very likely to be hijacked or even malware with the ID of EXCEL.EXE. Such pat-

[*]Work done during an internship at NEC Labs America.
[†]University of Illinois at Chicago, {swang224, psyu}@uic.edu.
[‡]NEC Laboratories America, {zchen, dingli, zhichun, ltang, jni, rhee, haifeng}@nec-labs.com.
[§]Corresponding author.

terns are stable during the evolution of the program, and the system behavioral events can be efficiently collected by system monitoring techniques [1, 2]. Based on this observation, in this paper, we propose **DeepHGNN**, an attentional deep heterogeneous graph neural network based approach for program reidentificaion by modeling the system behaviors of the program.

In particular, we design a compact graph modeling, the program behavior graph, to preserve all the useful information from massive system monitoring data and capture the interactions between different system entities. The constructed behavior graph is a heterogeneous graph , which involves a hierarchy of different dependencies from simple to complex. Among all the dependencies, the complex ones are not exposed directly by the edges in the graph but can be inferred by a hierarchy of deep representation. To capture the hierarchical dependencies, we first propose a multi-channel transformation module to transform the heterogeneous graph into the multi-channel graph guided by the meta-paths. After the multi-channel transformation, we feed the resulted graph and its corresponding entity features into a graph neural network for graph embedding. We propose a contextual graph encoder and stack it layer by layer to learn the hierarchical graph embedding via propagating the contextual information. Noticing the different importance of the different channels, channel-aware attention is further developed to align the multi-channel graph embeddings. We conduct an extensive set of experiments on real-world enterprise monitoring data to evaluate the performance of our approach. The results demonstrate the effectiveness of our proposed algorithm. We also apply **DeepHGNN** to real enterprise security systems. The results show our method is effective in identifying the disguised signed programs and robust to the normal dynamic changes like program version upgrade.

In summary, the contributions of this paper are:

- We identify the important problem of program reidentification in intrusion detection, which is often overlooked by the existing intrusion detection systems;
- We propose a heterogeneous graph model to capture the interactions between different system entities from large-scale system surveillance data;
- We develop a multi-channel transformation to transform a heterogeneous information network into a multi-channel graph;
- We propose a heterogeneous graph neural network based approach to learn the graph embedding via propagating the contextual information;
- We propose a channel-aware attention mechanism to learn the representation from different channels jointly;
- Our empirical studies on real enterprise monitoring data demonstrate the effectiveness of our method.

2 Preliminaries and Problem Statement

In this section, we first present the preliminaries, then define the machine learning problem that we are concerned with Deep Program Reidentification.

System Entity There are three main types of system entities in an operating system [6, 16]: *processes*, *files*, and *Internet sockets* (INETSockets). And each entity is associated with a set of categorical attributes. In this paper, we use "program" and "process" interchangeably whenever there is no ambiguity.

System Event A system event is an interaction between a pair of system entities. Formally, a system event $e = (v_s, v_d, t)$ represents a source entity v_s, destination entity v_d, and their interaction happens at time stamp t. There are multiple types of system events, due to the existence of different types of entities. We consider three different types of system events, including: (1) a process forking another process (P-P), (2) a process accessing a file (P-F), and (3) a process connecting to an INETSocket (P-I).

Problem Statement Given a target program with corresponding event data during a time window $U = \{e_1, e_2, ...\}$ and a claimed name/ID, we check whether it belongs to the claimed name/ID. If it matches the behavior pattern of the claimed name/ID, the predicted label should be +1; otherwise it should be −1. More formally, given event data U of a program, a mapping function f is used to map U to a binary label $Y \in \{+1, -1\}$, such that $f : U \to \{+1, -1\}$.

3 Algorithm

3.1 Overview In this section, we introduce **DeepHGNN**, a graph neural network based approach to verify the system program in a data-driven manner. The framework of **DeepHGNN** (as shown in Figure 1), consists of the following six components:

- **Surveillance Data Collection**. This module collects three different types of system events (see Section 2 for definition) from IT/OT systems;
- **Behavior Graph Modeling**. Based on the collected program event data, a compact heterogeneous program behavior graph is constructed for the target process to capture the complex interactions and eliminate the data redundancies;
- **Multi-Channel Transformation**. This module transforms the generated heterogeneous behavior graph to a multi-channel graph with each channel modeling one type of meta-path relationship;
- **Contextual Graph Encoder**. Based on the generated multi-channel graph, we propose an effective

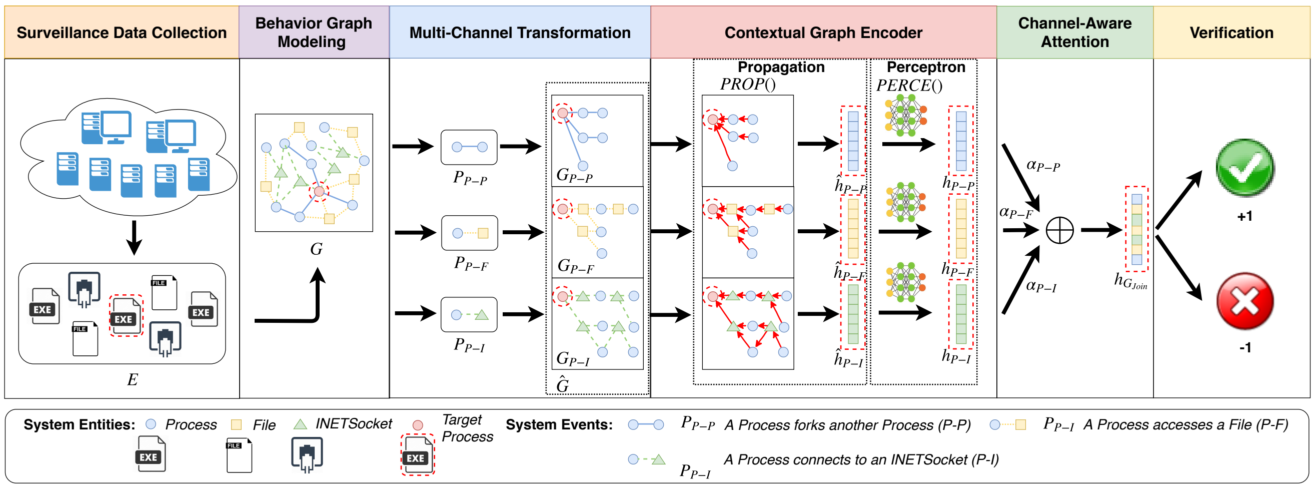


Figure 1: System architecture of deep program reidentification.

graph neural network with propagation layer and perceptron layer to extract the graph embedding;

- **Channel-Aware Attention**. In this module, an attention mechanism is proposed to align the graph embeddings extracted from different channels and generate the attentional joint embedding;
- **Program Reidentification** After the joint embedding being extracted, the resulted low-dimensional vector is fed to a classification layer to generate the prediction.

3.2 Behavior Graph Modeling Real information networks like the enterprise networks often generate a large volume of system behavioral data with rich information on program/process level events. We utilize a system monitoring tool (*i.e.*, Windows ETW [2]) to collect the program behavioral event data. The data has four properties: (1) High Volume. The system event data collected from a single computer system in one day can easily reach 20 GB and the number of events relates to one specific program can easily reach thousands. It is prohibitively expensive to perform the analytic task on such massive data in terms of both time and space; (2) High Dimensionality. Each system event is associated with hundreds of attributes, including information of involved system entities and their relationships, which causes the curse of dimensionality [23]; (3) Categorical. All available attributes are categorical, and each attribute has hundreds of categories. Analysis of categorical data is very challenging due to the lack of intrinsic proximity measure [24]; (4) Redundant. The attributes are redundant. Most of the attributes of system entities are the same. Repeatedly saving these attributes results in a significant redundancy. The events are also redundant. The events with the same system entity and relationship are usually repeatedly stored, which also causes a notable redundancy. These challenges motivate our idea of graph modeling.

We construct the compact graph to model the program behaviors. Formally, given the program event data U across many machines within a time window (*e.g.*, 1 day), a heterogeneous graph $G_{Behavior} = (V_{Behavior}, E_{Behavior})$ is constructed for the target program. $V_{Behavior}$ denotes a set of nodes, with each representing an entity of three types: process (P), file (F), and INETSocket (I). Namely, $V_{Behavior} = P \cup F \cup I$. $E_{Behavior}$ denotes a set of edges (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, including: (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 behavior graph modeling, we reduce the data redundancy significantly but keep important information.

3.3 Attentional Heterogeneous Graph Neural Networks After the graph modeling, a heterogeneous program behavior graph is constructed. There exist a hierarchy of different dependencies from simple to complex. Among all the dependencies, the complex ones are not exposed directly by the edge in the graph but can be inferred by a hierarchy of deep representation. For example, a process accessing a file (P$\rightarrow$F) is a simple dependency, two processes accessing the same file (P$\rightarrow$F$\leftarrow$P) is a complex dependency. To infer the relationship of two processes accessing the same file, we can check whether they share the same context. Due to the complex and obscure dependency, it requires a deep graph structure learning model to capture the hierarchical dependency. However, recent deep graph neural network methods [9, 12, 15, 22] only focus on the homogeneous graph and can not capture the heterogeneity within the heterogeneous graph. To learn the hierarchical dependency representation (hierarchical graph embedding) from heterogeneous graph, we propose an attentional heterogeneous graph neural network, which consists of multi-channel transformation, input layer, contextual graph encoder, and channel-aware attention.

Multi-channel Transformation Due to the heterogeneity intrinsic of entities (nodes) and dependencies (edges) in a heterogeneous graph, the diversities between different dependencies vary dramatically, which significantly increase the difficulty of applying graph neural network. To address this challenge, we design a multi-channel transformation module to transform the heterogeneous graph to a multi-channel graph guided by the meta-paths. A meta-path [20] is a path that connects entity types via a sequence of relations over a heterogeneous network. For example, in a computer system, a meta-path can be the system events (P→P, P→F, and P→I), with each one defining a unique relationship between two entities.

The multi-channel graph is a graph with each channel constructed via a certain type of meta-path. Formally, given a heterogeneous graph $\mathcal{G}$ with a set of meta-paths $M = \{M_1, ..., M_{|C|}\}$, the transformed multi-channel network $\hat{G}$ is defined as follows:

$$\hat{G} = \{G_i | G_i = (V_i, E_i, A_i), i = 1, 2, ..., |C|\} \tag{3.1}$$

where E_i denotes the homogeneous links between the entities in V_i, which are connected through the meta-path M_i. Each channel graph G_i is associated with an adjacency matrix A_i. $|C|$ indicates the number of meta-paths. Notice that, the potential meta-paths induced from the heterogeneous network $G_{Behavior}$ can be infinite, but not everyone is relevant and useful for the specific task of interest. Fortunately, there are some algorithms [8] proposed recently for automatically selecting the meta-paths for particular tasks.

Input Layer After the multi-channel transformation, an input layer is applied to construct the feature vector for each entity. Specifically, two types of features are constructed: connectivity features and statistic features. Since these features are constructed without any expert knowledge, they are knowledge free.

Connectivity features are constructed to describe the pairwise proximity between entities. The connectivity feature of entity v is defined as $X_v^{con} = \{e_{v,1}..., e_{v,|V|}\}$, which denotes the first-order proximity between v and other entities. Graph statistical feature of an entity v is generated based on the graph theory. It can be defined as $X_v^{stat} = \{X_v^{s1}, X_v^{s2}, X_v^{s3}, X_v^{s4}\}$, including four graph measurements: degree centrality, closeness centrality, betweenness centrality, and clustering coefficient.

Contextual Graph Encoder[1] In this step, we feed the multi-channel graph and the corresponding entity features to a graph neural network for graph embedding. To learn the hierarchical graph representation, we propose contextual graph encoder (CGE) and stack a deep graph neural work. Given the one-channel graph $G = (V, E, A)$ with each V associated with a corresponding feature X, our target is to learn an encoding function $f_G : G \to h_G$, that encodes the graph to a low dimensional vector (graph embedding), where h_G denotes the generated graph embedding vector. Specifically, we use the representation of the target entity h_{v_t} (target contextual entity embedding), which encodes both contextual structure and its features by a mapping function $f_V : V \to h_V$, as the representation of the graph. In this way, we reduce the graph embedding problem into the target entity embedding. Formally, our graph encoder can be defined as follows:

$$h_G = h_{v_t} = f_V(V_t) \tag{3.2}$$

The basic contextual graph encoder consists of a propagation layer $PROP()$ and a perceptron layer $PERCE()$. The propagating layer propagates the information from the context of the target entity. We define the propagation layer $PROP()$ as follows:

$$\hat{h}^l = PROP(h^l) = P^l h^l \tag{3.3}$$

where l denotes the layer number, P^l denotes the propagation matrix at layer l, and $\hat{h}^l$ indicates the output of $PROP()$ at layer l. The first layer $h^0 = X$ takes the features of each entity. The propagation layer propagates information to the target entity within a region, named graph receptive field $\mathcal{F}$. The graph receptive field $\mathcal{F}$ usually consists of all L-hop contexts of target entities. The propagation operation encodes both the graph structure and the entity features, which is similar to performing a graph convolution operation [15]. After the propagation is performed, a perceptron layer $PERCE()$ is applied to the propagated representation of the entities, such that:

$$h^{l+1} = PERCE(\hat{h}^l) = \sigma(\hat{h}^l W^l) \tag{3.4}$$

where W^l is the shared trainable weight matrix for all the entities at layer l and $\sigma()$ is a nonlinear gating function. Benefit from weight sharing, it is both statistically and computationally efficient compared with traditional entity embedding. Weight sharing can act as powerful regularization to preserve the invariant property in the graph, and the number of parameters is significantly reduced.

[1]In this subsection, we remove the subscripts of channel indicator for all the graph related symbols for simplification.

We design our propagation layer by performing the random walk on the graph via a diffusion process characterized by a specific probability $q \in [0,1]$ and a state transition matrix $D^{-1}A$. Here, D denotes the degree matrix of the adjacency matrix A, such that $D = diag(A\mathbf{1})$. $\mathbf{1}$ is a all one vector. After some transitions, such Markov process converges to a stationary distribution $P \in \mathcal{R}^{N \times N}$ with ith row indicates the likelihood of diffusion from the entity, hence the proximity of that entity. This stationary distribution of the diffusion process is proven to have a closed form solution [4]. When considering the 1-step truncation of the diffusion process, the propagation layer is defined as follows:

$$\hat{h}^l = PROP(h^l) = D^{-1}Ah^l = \sum_{u \in N(v_t)} P_{uv_t} h^l \tag{3.5}$$

The receptive field is defined as $\mathcal{F} = \{N(v_t)\}$, with $N(v_t)$ denoting all the contexts of target entity v_t. The propagation layer computes the weighted sum of the contexts' current representation. We set $P^0 = P^1 = P^2 = D^{-1}A$ and build a three-layer CGE as follows:

$$h^1 = \text{ReLU}((P^0X)W^0) \tag{3.6}$$

$$h^2 = \text{ReLU}((P^1h^1)W^1) \tag{3.7}$$

$$h^3 = \text{ReLU}((P^2h^2)W^2) \tag{3.8}$$

$$h_G = h_{v_t} = h^3 \tag{3.9}$$

where RELU() is a element-wise rectified linear activation. We perform the CGE to each channel graph and generate corresponding graph representation respectively.

Channel-Aware Attention Going through the CGE, the graph embeddings for each channel graph are extracted. However, different channels should not be considered equally. For example, Ransomware is usually very active in accessing the files, but it barely forks another process, or opens an internet socket, while the VPN 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 channels differently. Here, we propose a channel-aware attention, an attention mechanism, to align the graph embeddings from different channels and generate the joint embedding. Specifically, we learn the attention weights for different channels automatically.

Formally, given the corresponding graph embedding h_{G_i} for each channel $i = 1, 2, ..., |C|$, we define the attention weight as follows:

$$\alpha_i = \frac{\exp(\sigma(a[W_a h_{G_i} || W_a h_{G_k}]))}{\sum_{k' \in |C|} \exp(\sigma(a[W_a h_{G_i} || W_a h_{G_{k'}}]))} \tag{3.10}$$

where h_{G_i} is graph embedding of the target channel, h_{G_k} denotes the representation of the other channels. a denotes a trainable attention vector, W_a denotes a trainable weight mapping 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 that is used to compute the correlation between one channel and other channels. This correlation is normalized by a Softmax function. Let $ATT(h_{G_i})$ represent Eq.(3.10). The joint representation of each channel can be represented as follows:

$$h_{G_{Join}} = \sum_{i=1}^{|C|} ATT(h_{G_i}) h_{G_i} \tag{3.11}$$

The channel-aware attention allows us to better infer the importance of different channels by leveraging their correlations and learn a channel-aware representation.

3.4 Program Reidentification

After the joint graph embedding is generated from program event data, a binary classifier is used to verify whether the program matches its claimed name. Specifically, we trained the binary classifier for each known program. Our framework takes a claimed program event data as the input and generates the corresponding prediction. The final output is +1 or -1, indicating the identified or unidentified prediction result.

To train a verification model for a particular program, we collect a set of program events $\mathcal{U} = \{U_1, U_2, ..., U_m\}$ including events belong to that program and the ones do not belong to that program. Their corresponding labels are $Y = \{y_1, y_2, ..., y_m\}$ with $y_i \in \{+1, -1\}$. If the event data belongs to the claimed program, its ground truth label $y_i = +1$, otherwise its ground truth label $y_i = -1$. Our target is to design an end-to-end binary classifier. Specifically, we propose to use logistic regression classifier, and the objective function is defined as follows:

$$l = \frac{1}{m} \sum_{i=1}^{m} [y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)] \tag{3.12}$$

where $\hat{y}_i$ denotes the predicted label. We optimize the above objective with Adam optimizer. The gradients of the parameters are calculated recursively according to the graph topology. Since our approach is end to end, we directly optimize the reidentification objectives. Once the model achieves a good performance (*e.g.*, using accuracy (ACC) as the measure), the training process terminates, and the trained model is suitable for program reidentification.

4 Experiments

4.1 Data

In the experiments, we evaluate the proposed method on the real-world system monitoring data. The data is collected from a real enterprise network composed of 87 machines, in a time span of 20 consecutive weeks. The sheer size of the data set is around 3 Terabytes. We consider three different types of system events as defined in Section 2. 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 the heterogeneous behavior graph (see Section 3.2) per program per day. For each entity, we construct three different types of features according to Section 3.3: *fea-1*: connectivity feature, *fea-2* statistics feature, and *fea-3*: the combination of *fea-1* and *fea-2*.

4.2 Experiment Setup

4.2.1 Baselines We compare the proposed method **DeepHGNN** with the following typical and state-of-art classification models:

- *LR* and *SVM*: LR and SVM represent the Logistic Regression and Linear Support Vector Machine, respectively. They are two typical classification methods. The raw features extracted from each process are used as the input, including the connectivity features and the graph statistics feature. The LR and SVM are implemented using sciketlearn.
- *XGB*: XGB represents the gradient boosting. It is a decision tree based classification model implemented using XGBoost [7]. It is the state-of-art linear classification model for most of the tasks. We set maximum 500 trees with the learning rate equals to 0.1.
- *MLP*: MLP represents the Multi-layer Perceptron, which is a deep neural network based classification model with multiple non-linear layers between the input layer and the output layer. It is a special case of our contextual graph encoder if we define the propagation layer as an identity matrix.

Since our approach **DeepHGNN** consists of six components, we consider different variants as well.

4.2.2 Evaluation Metrics Similar to [6, 16], we evaluate the performance of different methods using a variety of measures, including accuracy (ACC), F-1 score, AUC score, precision, and recall.

4.3 Synthetic Experiments

To evaluate the proposed method in a more controlled setting, we conduct three sets of synthetic experiments on 500 most active programs as follows: (1) We evaluate the effectiveness of multi-channel transformation and channel-aware attention modules; (2) We evaluate the performance of our method on normal program reidentification; (3) We perform the parameter sensitivity analysis. We perform 5-fold cross-validation on each program and report the average testing results of all the programs for each evaluation metric. We conduct a grid search on the parameter of each method to identify the parameter setting that yields the best result, which is done using cross-validation. In particular, we set the dimension of the hidden layer to 500.

Meta-Path	Evaluation Criteria		
	ACC	F-1	AUC
DeepHGNN$_{pp}$	0.838	0.864	0.843
DeepHGNN$_{pf}$	0.821	0.855	0.838
DeepHGNN$_{pi}$	0.579	0.635	0.592
DeepHGNN$_{con}$	0.876	0.901	0.890
DeepHGNN$_{att}$	**0.905**	**0.929**	**0.908**

Table 1: Reidentification results of different meta-paths.

4.3.1 Evaluation of Different Meta-paths In this experiment, we evaluate the performance of different kinds of meta-paths and their combinations. Giving each kind of meta-path, such as P→P, P→F, or P→I, corresponding to one type of system event, we construct the multi-channel graph with entity features and then feed them to the contextual graph encoder to generate the graph embedding. After the graph embedding obtained, we feed it to logistic regression to train the classification model for program reidentification. We denote these baselines as **DeepHGNN**$_{pp}$, **DeepHGNN**$_{pf}$, and **DeepHGNN**$_{pi}$. To effectively evaluate our attention module, we consider both direct concatenation and our proposed attentional version, which are denoted as **DeepHGNN**$_{con}$ and **DeepHGNN**$_{att}$. From Table 1, we can observe that: (1) The **DeepHGNN**$_{att}$ outperforms **DeepHGNN** with any single type of meta-path and their simple combination; (2) The combinations of multiple meta-paths perform better than any single meta-path, since they cover multiple types of dependency and provide complementary information; (3) The one with channel-aware attention performs better than simple concatenation, since the importance of different meta-paths vary for different programs. Channel-aware attention captures that difference and computes a weighted combination of different meta-paths.

4.3.2 Evaluation on Normal Program Reidentification In this experiment, we evaluate the performance on normal program reidentification by comparing **DeepHGNN** with other four baselines including LR, SVM, XGB, and MLP. We use the constructed features as described in Section 4.1. To show the effectiveness of

Method	Settings	Evaluation Criteria				
		ACC	F-1	AUC	Precision	Recall
LR	*fea-1*	0.693	0.755	0.699	0.632	0.948
	fea-2	0.705	0.770	0.703	0.655	0.950
	fea-3	0.724	0.772	0.727	0.675	0.948
SVM	*fea-1*	0.502	0.662	0.502	0.505	0.970
	fea-2	0.795	0.778	0.725	0.701	0.935
	fea-3	0.504	0.652	0.504	0.505	**0.975**
XGB	*fea-1*	0.775	0.802	0.776	0.732	0.930
	fea-2	0.833	0.860	0.846	0.821	0.936
	fea-3	0.855	0.866	0.856	0.827	0.937
$MLP_{shallow}$	*fea-1*	0.633	0.745	0.643	0.626	0.938
	fea-2	0.775	0.808	0.779	0.724	0.932
	fea-3	0.778	0.808	0.780	0.726	0.932
MLP_{deep}	*fea-1*	0.633	0.743	0.653	0.625	0.945
	fea-2	0.801	0.830	0.805	0.769	0.921
	fea-3	0.815	0.831	0.816	0.778	0.923
DeepHGNN$_{shallow}$	/	0.905	0.929	0.908	0.905	0.933
DeepHGNN$_{deep}$	/	**0.929**	**0.961**	**0.935**	**0.932**	0.936

Table 2: Comparison on normal program reidentification.

the deep learning model, we consider both shallow and deep versions of the neural network based model, which is denoted as "$XXX_{shallow}$" and "XXX_{deep}", respectively. We consider the one-layer configuration as the shallow model and the three-layer configuration as the deep model. For the baseline methods, we use the constructed raw features: *fea-1*, *fea-2*, or *fea-3* as the input, respectively. Table 2 shows that overall, **DeepHGNN** consistently and significantly outperforms all baselines in terms of all metrics. More specifically, (1) *fea-3* (concatenate the connectivity feature and graph statistic feature) helps to improve the performance for all baseline classification models, but since they are all raw features without considering graph structure, their performance can not catch up with the proposed method; (2) *fea-2* (graph statistic feature) is more useful than *fea-1* (graph connectivity feature), since *fea-1* is very sparse, which is difficult to use for some state-of-art methods, such as SVM; (3) The deep model outperforms the shallow model for our approach, since the deep model can capture the hierarchical dependency representation.

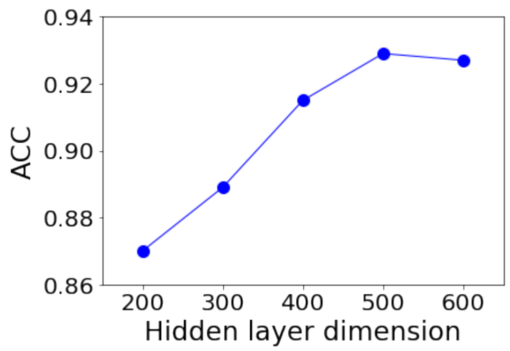
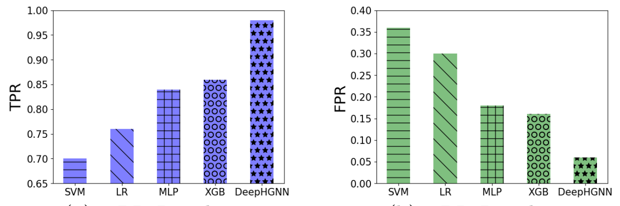


Figure 2: Parameter sensitivity analysis results.

In this experiment, we also conduct the sensitivity analysis of how different choices of parameters will affect the performance of our proposed approach. Specifically, **DeepHGNN** has an important parameter: the number of dimensions in each perceptron layer. Figure 2 shows that the performance of **DeepHGNN** is not significantly affected by the number of hidden layer dimensions.

4.4 Real-world Experiments In this section, we evaluate the performance of the **DeepHGNN** in two real-world tasks: disguised program detection and normal program upgrade detection.



(a) TPR Results. (b) FPR Results.

Figure 3: Disguised program detection results.

4.4.1 Disguised Program Detection In order to avoid detection, hackers usually disguise their malicious program as a legitimate program. To test the performance of **DeepHGNN** in disguised program detection, we simulate five different types of real advanced attacks including: (1) *WannaCry*: A crypto worm which disguised itself as 7z.exe; (2) *Phishing Email*: A malicious Trojan is downloaded as an Outlook attachment, and the enclosed macro is triggered by Excel to create a fake java.exe, and the disguised malicious java.exe further exploits a vulnerable server to start cmd.exe in order to create an info-stealer; (3) *Emulating Enterprise Environment*: The hackers generate telnet process to create a Trojan malware binary with a disguised normal program name. Then, DLL is injected through the running process notepad.exe. The hackers use mimikaz and kiwi for memory operation inside the meterpreter context. Finally, malware PwDump7.exe and wce.exe are copied and run on target hosts; (4) *Diversifying Attack Vectors*: The hacker first writes malicious PHP file by HTTP connection, then downloads the malware process (Trojan.exe), and connects back to the hacker host. The process notepad.exe is run to perform DLL injection. The hacker further uses mimikaza and kiwi to perform memory operation inside meterpreter context. Finally, it copies and runs Pwdup7.exe and wce.exe on the target host; (5) *Domain Controller Penetration*: The hackers first send an email attaching a document that includes the malware python32.exe. This malware opens a connection back to the hackers so that they can run notepad.exe and perform reflective DLL injection to obtain the needed privileges. Finally, they transfer password enumerator and run the process gsecdumpv2b5.exe to get all user credentials. We try each type of the attacks ten times during different time windows to generate different testing samples.

The detection performance is evaluated using the true positive rate (TPR) and false positive rate (FPR). The TPR defines the fraction of intrusion attacks that are detected during the evaluation. FPR, on the other hand, describes the fraction of normal event sequences that trigger an alert during the evaluation. We compare our method with the four baselines and use the deep model for all neural network based techniques. Figure 3 shows that **DeepHGNN** outperforms all the other baselines by at least 12% in TPR and 10% in FPR.

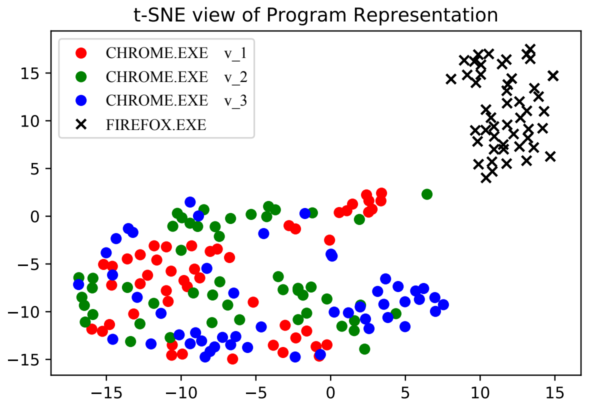


Figure 4: Scatter plot embedding of different versions of CHROME.EXE vs FIREFOX.EXE.

4.4.2 Robustness Study on Program Upgrade

The normal program usually gets the upgrade with multiple versions. However, different versions have different signatures, which may cause false alarms in intrusion detection systems. In this experiment, we evaluate the robustness of **DeepHGNN** on normal program upgrades. In particular, we select the web browser software CHROME.EXE for a case study. We collect the data of CHROME.EXE with three different versions and one version of another web browser software FIREFOX.EXE. Then, the graph embedding before the classifier for these CHROME.EXE data and FIREFOX.EXE data are extracted and visualized using t-SNE. From Figure 4, we can see that in the embedding space, the data points belong to the different versions of CHROME.EXE are close to each other, even though they have different signatures, while the data points of FIREFOX.EXE are far away from all the CHROME.EXE data points, which demonstrates our method is robust to the normal program upgrades.

5 Related Work

5.1 Intrusion Detection In general, intrusion detection refers to the process of monitoring the events occurring in a computer system or network and analyzing them for signs of intrusions. Currently, there are two main types of approaches, namely anomaly detection and misuse detection [14]. Anomaly detection approaches define and characterize correct/wrong behaviors of the system, while the misuse detection approaches monitor for explicit patterns, with the intrusion patterns known in advance. Existing anomaly detection approaches in large-scale enterprise network systems have been separately considering different data representations. In particular, host-based anomaly detection methods [6, 10, 16] locally extract patterns from process-level events as the discriminators of abnormal intrusion. In contrast, network-based anomaly detection methods [18] focus on disclosing abnormal subgraph structures from network-level events, most of which are inspired by graph properties.

5.2 Graph Representation Learning Representation learning [5] has become a very promising topic in machine learning with wide applicability. Many representation learning work aim at learning representations with deep learning due to its powerful feature learning ability. In recent years, several deep learning approaches have been proposed on graph-structured data, including [9, 12, 15, 22]. They leverage convolution operation in the spatial domain or spectral domain. [9] proposes the fast localized convolutions. [15] proposes a first-order approximation scheme to reduce the computation cost of graph filter spectrum. [12] extends the graph convolution with a more general form of sample and aggregation function for node context. More recent work [22] learns the context aggregation with considering the weights between the neighborhood and current nodes. However, these work mainly focus on the node classification for heterogeneous network. Different from theirs, our work focuses on learning the embedding from the heterogeneous graph.

6 Conclusion

In this paper, we investigate the problem of program reidentification in IT/OT systems, which is often overlooked by traditional intrusion detection techniques. We propose **DeepHGNN**, an attentional heterogeneous graph neural network method to verify the program's identity based on its heterogeneous behavior graph. Different from the existing homogeneous graph neural network methods, **DeepHGNN** is able to capture and encode the heterogeneous complex dependency among different entities in a hierarchical way. The experimental results show that **DeepHGNN** outperforms all the baseline methods by at least 10% in terms of all the metrics. We also apply **DeepHGNN** to a real enterprise system for disguised program detection. Our method can achieve superior performance and demonstrate robustness across the normal dynamic changes.

7 Acknowledgements

The first author is supported in part by NSF through grants IIS-1526499, IIS-1763325, and NS-1626432.

References

[1] `https://wiki.archlinux.org/index.php/Audit_framework` (2016)

[2] `https://msdn.microsoft.com/en-us/library/ff357719` (2017)

[3] Arefi, M.N., Alexander, G., *et al.*: Faros: illuminating in-memory injection attacks via provenance-based whole-system dynamic information flow tracking. In: 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. pp. 231–242 (2018)

[4] Atwood, J., Towsley, D.: Diffusion-convolutional neural networks. In: Advances in Neural Information Processing Systems. pp. 1993–2001 (2016)

[5] Bengio, Y., Courville, A., Vincent, P.: Representation learning: A review and new perspectives. IEEE Transactions on Pattern Analysis and Machine Intelligence 35(8), 1798–1828 (2013)

[6] Cao, C., Chen, Z., *et al.*: Behavior-based community detection: Application to host assessment in enterprise information networks. In: 27th ACM International Conference on Information and Knowledge Management. pp. 1977–1985 (2018)

[7] Chen, T., Guestrin, C.: Xgboost: A scalable tree boosting system. In: 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 785–794. ACM (2016)

[8] Chen, T., Sun, Y.: Task-guided and path-augmented heterogeneous network embedding for author identification. In: Tenth ACM International Conference on Web Search and Data Mining. pp. 295–304 (2017)

[9] Defferrard, M., Bresson, X., *et al.*: Convolutional neural networks on graphs with fast localized spectral filtering. In: Advances in Neural Information Processing Systems. pp. 3844–3852 (2016)

[10] Dong, B., Chen, Z., *et al.*: Efficient discovery of abnormal event sequences in enterprise security systems. In: 26th ACM International Conference on Information and Knowledge Management. pp. 707–715 (2017)

[11] Gao, P., Xiao, X., *et al.*: Saql: a stream-based query system for real-time abnormal system behavior detection. arXiv preprint arXiv:1806.09339 (2018)

[12] Hamilton, W., Ying, Z., *et al.*: Inductive representation learning on large graphs. In: Advances in Neural Information Processing Systems. pp. 1024–1034 (2017)

[13] Hendric, W.: A complete overview of trusted certificates. In: CA/Browser Forum (2015)

[14] Jones, A.K., Sielken, R.S.: Computer system intrusion detection: a survey. Computer Science Technical Report pp. 1–25 (2000)

[15] Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907 (2016)

[16] Lin, Y., Chen, Z., *et al.*: Collaborative alert ranking for anomaly detection. In: 27th ACM International Conference on Information and Knowledge Management. pp. 1987–1995 (2018)

[17] Liu, Y., Zhang, M., *et al.*: Towards a timely causality analysis for enterprise security. In: Network and Distributed Systems Security Symposium (2018)

[18] Noble, C.C., Cook, D.J.: Graph-based anomaly detection. In: ACM International Conference on Knowledge Discovery and Data Mining. pp. 631–636 (2003)

[19] Ren, C., Chen, K., Liu, P.: Droidmarking: resilient software watermarking for impeding android application repackaging. In: 29th ACM/IEEE International Conference on Automated Software Engineering. pp. 635–646 (2014)

[20] Sun, Y., Han, J., *et al.*: Pathsim: Meta path-based top-k similarity search in heterogeneous information networks. VLDB Endowment (2011)

[21] Tang, Y., Li, D., *et al.*: Nodemerge: template based efficient data reduction for big-data causality analysis. In: 25th ACM Conference on Computer and Communications Security (2018)

[22] Velickovic, P., Cucurull, G., *et al.*: Graph attention networks. arXiv preprint arXiv:1710.10903 (2017)

[23] Verleysen, M., François, D.: The curse of dimensionality in data mining and time series prediction. In: International Work-Conference on Artificial Neural Networks. pp. 758–770. Springer (2005)

[24] Zhang, K., Wang, Q., *et al.*: From categorical to numerical: Multiple transitive distance learning and embedding. In: 2015 SIAM International Conference on Data Mining. pp. 46–54 (2015)

[25] Zimba, A.: Malware-free intrusion: a novel approach to ransomware infection vectors. International Journal of Computer Science and Information Security 15(2), 317 (2017)