ScienceDirect

Engineering Applications of Artificial Intelligence

Volume 152, 15 July 2025, 110524

Research paper

A novel malware detection method based on audit logs and graph neural network

Yewei Zhen [a], Donghai Tian [a b], Xiaohu Fu [a c], Changzhen Hu [a]

Show more

Outline | Share | Cite

https://doi.org/10.1016/j.engappai.2025.110524

Get rights and content

Full text access

Abstract

Malicious programs pose a significant threat to cyberspace security, making practical and low-cost malware detection a pressing need. To address this problem, we propose a novel malware detection method based on audit logs and graph neural networks. This method first performs fine-grained parsing of the logs for obtaining the process event sequence and process invocation relationship. Then, we employ a graph convolutional network to generate an embedding vector representation for each extracted process event, effectively capturing both local and global co-occurrence information. Next, the process structure and event semantic information are used to construct an event relationship graph for each log sample. Based on the event relationship graphs, we leverage an attention gated graph neural network (AGGNN) for malware detection. The evaluation shows that our approach can detect malware effectively with explainable results, and it outperforms the recent malware detection methods based on audit logs.

Previous Next

Keywords

Audit log; Malware detection; Graph neural network

1. Introduction

The rise and development of the Internet has gradually made it an indispensable part of people's daily lives. According to the 2022 edition of Facts and Figures,[1] an estimated 5.3 billion people use the Internet. This represents a growth rate of 6.1% over 2021, up from 5.1% for the period of 2020–2021. With the rapid development of the Internet, the amount and variety of malware has also increased dramatically. These malware can cause various forms of damage to computer systems or software, damage computer hardware, steal computer system files or user information, etc. To defend against malware attacks, researchers propose various detection methods.

Previous solutions to address the malware detection problem can be classified into two classes: static method and dynamic method. The first method utilizes static analysis to analyze the target program. By extracting the discriminative features (e.g., strings and byte sequences) of the program, the static method can detect the malicious program efficiently. However, the static method can easily be evaded by advanced obfuscation techniques (e.g., encryption and packing).

Different from a static method, the dynamic method needs to obtain the program's runtime behavior for analysis so that the target program should be executed first. Even if the target program is well obfuscated, its malicious behavior will be revealed dynamically. Therefore, the dynamic method is capable of dealing with the obfuscated malicious program. However, the existing dynamic approaches may have 3 limitations: (1) many approaches introduce considerable performance cost; (2) some approaches require running the target programs in virtualized environments embraced various probes, which may hinder their real deployment; (3) many approaches based on machine (or deep) learning lack explanation for malware detection.

For example, many methods (Zhou et al., 2023, Han et al., 2019, Canzanese et al., 2015, Xiaofeng et al., 2019) depend on dynamically intercepting system calls for malware interception. If the target program invokes system calls frequently, the performance cost would be considerable due to the system call hooking. To intercept system calls,

additional agent needs to be placed into the host system. Moreover, the recent studies (Fadadu et al., 2019, Chen et al., 2023) show that the system call-based methods could be evaded by advanced malware.

In order to address the above problems, we propose a novel malware detection method based on audit logs. The basic idea of our approach is to extract semantic features from the audit logs and then leverage graph neural network for malware detection. To obtain the meaningful features, we make use of the domain knowledge to extract the process events from the audit logs. Our method relies on the built-in audit log mechanism, which can be well utilized for collecting process events efficiently (Xiong et al., 2022, Bansal et al., 2023). Before analyzing the process events, we propose an optimization method based on information gain to reduce data redundancy. After that, a heterogeneous graph is constructed based on the optimized process events. Next, the heterogeneous graph is analyzed using a graph convolutional network to derive the semantic embeddings of events. To facilitate analyzing these features, the process structure and event semantic information are used to construct an event relationship graph for each log sample. Finally, we apply an attention gated graph neural network (AGGNN) to analyze the event relationship graph for malware detection.

We have implemented our approach based on PyTorch. To evaluate the effectiveness of our approach, we adopt the public dataset constructed by Berlin et al. (2015), which contains the labeled Windows audit logs. The experiments demonstrate that our solution can identify malicious program behavior effectively via analyzing the logs.

In summary, we make the following contributions:

1. We propose a novel malware detection method by leveraging the audit logs and graph neural network.

2. We utilize a graph convolutional network to vectorize the log events. Based on the vectorized log events and their connections, we make use of an attention gated graph neural network (AGGNN) for malware detection.

3. We carry out extensive experiments to evaluate our approach on public datasets with a variety of experimental settings. The experimental results show that our approach outperforms the recent audit log-based methods.

2. Related work

Malware detection methods can be categorized into two main types: static and dynamic approaches. Log-based methods fall under the category of dynamic approaches.

2.1. Static approach

Static detection typically revolves around parsing the executable file formats, extracting bytecode, opcodes, control-flow graphs, and other information as static features for analyzing malicious programs. Numerous studies (Christodorescu et al., 2005, Moskovitch et al., 2008) made use of instruction information to detect malware. Certain solutions (Kong and Yan, 2013, Narouei et al., 2015) took advantage of program structural information for malware detection. In recent times, a wide array of researchers (S.L and CD, 2019, Ni et al., 2018, Sun and Qian, 2018, Kalash et al., 2018) adopted various machine learning and deep learning techniques to classify grayscale (or color) images converted from program byte streams. This process of image classification aids in identifying malicious programs. In particular, Naeem et al. (2020) proposed a malware detection solution that is centered around color image visualization and convolutional neural networks. Yakura et al. (2018) applied a Convolutional Neural Network (CNN) model with the attention mechanism for classifying the images transformed from the binary programs. Similarly, Ma et al. (2019) presented a new classification framework based on attention mechanism and Convolutional Neural Networks (CNN) mechanism. The primary strength of static methods lies in their capability to analyze target programs efficiently without the need for execution. However, these static methods face challenges when addressing programs that employ advanced obfuscation techniques, such as packing and encryption.

2.2. Dynamic approach

Dynamic detection methods typically concentrate on capturing software behavior information or system resource changes during program execution, extracting dynamic features within the process to identify malicious behavior. Ahmed et al. (2009) introduced a machine learning approach that leverages spatio-temporal information in API calls for malware detection. Lanzi et al. (2010) described a dynamic method that concentrates on analyzing API invocations to access crucial resources. Canzanese et al. (2015) initially applied TF-IDF transformation for feature extraction from system call traces and then employed machine learning techniques to identify malware. Han et al. (2019) employed both static and dynamic features to detect malicious programs. Jindal et al. (2019) developed a system that utilizes deep learning to analyze dynamic analysis reports, which primarily contain API call information. Xiaofeng et al. (2019) combined

API sequence and statistics features for malware detection. Zhou et al. (2023) utilized a HAN model to analyze the API sequence features to identify malware. Burnap et al. (2018) conducted an analysis of system-level activity metrics, such as RAM usage (count), SWAP usage (count), and received packets (count), by executing samples of both malicious and benign executables in a sandbox environment. They utilized Self Organizing Feature Maps (SOFM) to process the machine activity data to identify malicious program. Abdelsalam et al. (2018) conducted an analysis by running diverse program samples on virtual machines and gathering performance indicators (e.g., number of used threads, number of opened file descriptors) of processes. They utilized convolutional neural networks for malware detection based on the performance information. Oliveira and Sassi (2019) extracted behavioral graphs from API invocation sequences and subsequently employed depth graph convolutional neural networks (DGCNN) for malware detection. Li et al. (2022) proposed a malware detection method based on graph convolutional network. Recently, Deng et al. (2023) presented a GNN model to analyze the API call graph for IoT malware analysis.

Compared with the recent dynamic studies, our method mainly has three differences. The first one is that the detection scenario is different. In our application scenario, we do not decide which process could be the detection target beforehand. In Deng's work, they assume the detection target can be identified first. The second difference is that we rely on the built-in audit logs for malware detection while Deng's study depends on the additional tool that acquires hooking system calls to detect malware. The third difference is that we utilize a different GNN model for detecting malware. Particularly, our method could provide explainable detection results.

2.3. Log-based approach

Many dynamic detection methods necessitate additional tools for acquiring program features, which could introduce performance overhead on the system. Consequently, an increasing number of researchers (Bao et al., 2018, Bertero et al., 2017, Du and Li, 2016, Du et al., 2017, Li et al., 2019, Liu et al., 2019, Nandi et al., 2016, Yen et al., 2019, Yuan et al., 2019, Zhang et al., 2019) are turning to log information as features for malware detection. This approach typically involves four main steps, as outlined in a study by Zhang et al. (2020): log collection and preprocessing, log parsing, feature extraction, and anomaly detection.

The standard built-in audit log data stream on the Windows platform provides sufficient information to detect malicious behavior (Ma et al., 2015). However, there is a scarcity of

studies utilizing Windows audit logs in the field of malicious code detection, and few publicly available datasets exist for this purpose. Berlin et al. (2015) were the first to construct a Windows audit log dataset by extracting log event sequential features through n-gram analysis and utilizing a simple logistic regression classifier, achieving a detection rate of 83%. Building on this work, Ring et al. (2021) employed a natural language processing approach for log event embedding and utilized an LSTM model for detection, resulting in an approximately 7% point improvement in detection accuracy compared to the work (Berlin et al., 2015). Guo et al. (2021) present a self-supervised framework for log anomaly detection based on Bidirectional Encoder Representations from Transformers (BERT). Some researchers (Rhode et al., 2018, Roy and Chen, 2021, Verma and Bridges, 2018) have also achieved promising results using Windows logs to detect and classify ransomware.

Most studies utilizing audit logs for malware detection encounter several challenges. Firstly, many of these studies lack enough granularity in log parsing, leading to the loss of structural information such as process invocation during the embedding of log events. Secondly, these studies mainly focus on the sequential relationship of log event sequences, neglecting the extraction of global and local structural information among log events. Thirdly, most models rely on high-quality training datasets, resulting in weak robustness. Lastly, most studies lack interpretability for the detection results.

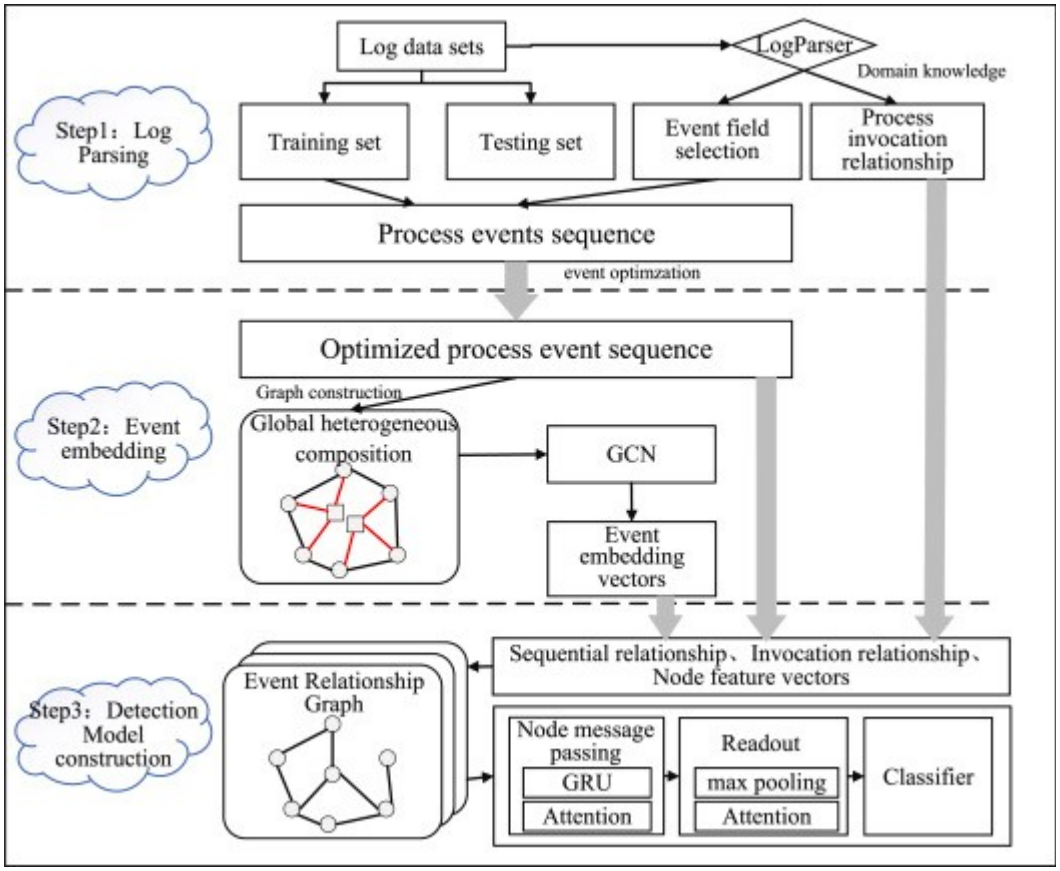
3. Approach

3.1. Overview of our approach

The primary task of our approach is to detect malicious programs based on audit logs. Similar to previous dynamic methods, we rely on analyzing the run-time behaviors of programs for dynamic detection. Different from previous solutions, we leverage the audit logs to collect the program behaviors. To analyze the audit logs intelligently, we make use of graph neural networks so that malicious behaviors could be identified. As shown in Fig. 1, the workflow of our approach can be divided into three stages: log parsing, process event embedding, and detection model construction.

In the first stage, we granularly parse the logs to extract both the event sequence of the process and the process invocation relationship. To the best of our knowledge, the majority of existing studies overlook the significance of process invocation relationship information, which is crucial for effective system-wide malware detection. In the second

stage, we construct a heterogeneous graph with logs and events as its nodes. After that, we employ a graph convolutional neural network to extract both global and local co-occurrence data of events from this graph, thereby enriching the semantic embedding of events. In the third stage, we first utilize the process invocation and event sequential relationship to construct an event relationship graph for each log. We observe that an event graph contains rich program semantics information. Next, we make use of an attention gated graph neural network to anatomize the event relationship graph enabling the detection of malicious behaviors. The attention mechanism within the model effectively summarizes essential event features based on the attention weights, thereby enhancing the model's interpretability.



Download: Download high-res image (568KB)

Download: Download full-size image

Fig. 1. Research ideas for malicious code detection based on Windows audit logs.

3.2. Log parsing

As shown in Fig. 2, each log sample contains a time frame and a list of processes, each with a name, pid (process identifier), and associated events. An event is the smallest unit

in a log file and contains its corresponding attributes, including time, event id, action, target, etc. Particularly, the action field indicates the operation type (e.g., Spawn, Execute, Write), and the target field specifies the other objects being operated on (e.g., file, directory, registry key). The primary objectives of the log parsing step encompass two parts. The first part is to acquire the process invocation relationships, and the second part is to derive the process event sequences from the logs.

For the first objective, we aim to identify the process invocation relationships initiated by the sample software within a pre-defined time window. This involves extracting relevant processes from the logs of all operating system processes and constructing a process tree structure.


```
{
 "start_time": "2014-09-21T22:25:35.000038477",
 "end_time": "2014-09-21T22:27:37.000038858",
 "size": 322,
 "Processes": [
  {
   "pid": 3712,
   "name": "[python]\\pythonw.exe",
   "events": [
    {
     "time": "2014-09-21T22:25:35.000038477",
     "event_id": 4663,
     "ignored": false,
     "string_id": "WINDOWS_FILE:Execute:[system]\\cmd.exe",
     "action": "Execute",
     "target": "[system]\\cmd.exe",
     "abstraction": ""
    },
    ...
  ]}
  ...
}
```


Download: Download high-res image (370KB)

Download: Download full-size image

Fig. 2. A sample software log example.

To achieve the second objective, we leverage the insights from Ring et al. (2021), utilizing the field **action** and **target** of each event as its semantic representation. Subsequently, we sort the events within a process based on the timestamp information from the audit log, resulting in a sequence of process events.

The specific implementation details can be found in Algorithm 1. The algorithm begins by creating an empty list called **pids**, which records all processes related to the log sample. Then, starting from the first process, it iterates through the process events. If an event

with a specific id (i.e., 4688) is encountered, indicating the event is for starting a new process, the **new pid**field in the event provides the pid of the new process. This pid is added to the **pids** list, resulting in a sequence of events for all relevant processes. For example, as shown in Fig. 3, events E2 and E3 of process 1 are both events with event id 4688. The behavior of event E2 is to start a new process P2, while the behavior of event E3 is to start a new process P3. For easy representation, we construct two dummy events E0 for the new processes P2 and P3. Similarly, the behavior of event E2 in process P2 is to start process P4, so an initial event E0 is created for P4, and a connection is established between processes P2 and P4.

Algorithm 1 Log Parsing

```
Require: Path of log files to be parsed: logFilePath
Ensure: Parsed log file list: parsedList
  parsedList ← NULL
  function LogParser(logFilePath)
    for all logFile ∈ os.walk(logFilePath) do
      Initialize the pids and fileEvents
      Get the pid of the first process and add it to the pids
      for all pid ∈ pids do
        processEvents ← GetProcessEvents(pid)
        if event.event_id == specific value then
          newPid ← GetNewProcessPid(event)
          pids.append(newPid)
        end if
      end for
      fileEvents.append(processEvents)
    end for
    parsedList.append(fileEvents)
    return parsedList
  end function
```

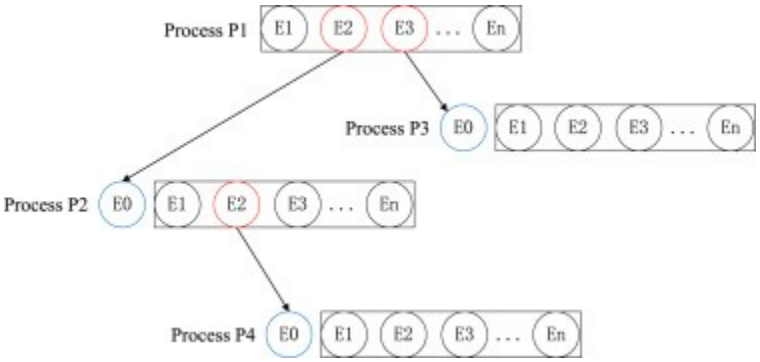
Download: Download high-res image (403KB)

Download: Download full-size image

After log parsing, we can get the process events. To reduce the manual efforts in identifying the useful process events for discriminating benign and malicious programs, we leverage the information gain method. This method has been extensively used for extracting features. There are three advantages for us doing so. First, it removes redundant or irrelevant features. Second, it enhances model performance by eliminating unnecessary features. Third, it improves prediction accuracy by focusing on the most informative features for decision-making. Before introducing our method, the information entropy joint entropy, and conditional entropy should be explained first. Information entropy[2] is an indicator used to describe the uncertainty of random variables. It is a measure related to the probability distribution of random variables. Assume that the sample set of our data set is S, and the samples have a total of n categories. The probability that one sample in S belongs to the set X is $p(x)$, then the

definition of the information entropy of the set S is shown as follows:

$$H(x) = -\sum_{x \in X} p(x) \log p(x) \tag{3.1}$$



Download: Download high-res image (185KB)

Download: Download full-size image

Fig. 3. A log parsing example.

We can extend the one-dimensional random variable distribution to a multidimensional random variable distribution, so its joint entropy is defined as follows:

$$H(X,Y) = -\sum_{x \in X, y \in Y} p(x,y) \log p(x,y) \tag{3.2}$$

Conditional entropy represents the entropy of a random variable under certain conditions. The conditional entropy is defined as follows:

$$H(Y \mid X) = \sum_{x \in X} p(x) \cdot H(Y \mid X = x) = \sum_{x \in X, y \in Y} p(x,y) \cdot \log \frac{p(x)}{p(x,y)} \tag{3.3}$$

Information gain refers to the reduction of the uncertainty of the original variable under a certain condition of the random variable. The information gain is defined as follows:

$$IG(Y,X) = H(Y) - H(Y \mid X) \tag{3.4}$$

In our case, we need to calculate the information gain of each process event. If the information gain value is bigger than the threshold, the process event will be preserved. Otherwise, the process event will be removed.

3.3. Process event embedding

After the process event sequences are optimized, the next step is to vectorize the process event to enable the deep learning model to effectively analyze them. For this purpose, we utilize the embedding method based on graph convolutional networks. Firstly, we make

use of the sliding window method (Yao et al., 2019, Huang et al., 2019) to construct a heterogeneous graph. This graph contains two different types of nodes: log file nodes and process event nodes. As shown in Fig. 4, the square represents the log file node while the circle represents the process event node. In this example, the sliding window size is 3 (The optimal size is established based on experimental results). There are two types of edges: Type I and Type II. Type I edges refer to the ones between process events, and Type II edges represent the edges between process events and log documents. Type I edges are established by the sliding window, and Type II edges are constructed according to the inclusion relationship between process events and log documents. We assume that there are M log documents, and there are N distinct process events in these documents. Then, we can define a heterogeneous graph $G = (V, E)$ where V is the set of nodes, and E refer to the set of edges. The number of nodes in the graph V is M (number of log documents) plus N (number of process events). Let n be the number of nodes and $X \in R^{n \times d}$ be the feature matrix of nodes, where d refers to the dimension of the feature vectors. Let $A \in R^{n \times n}$ be the adjacency matrix. To set the weight of Type I edge, we leverage point-wise mutual information (PMI), which could make full use of process event co-occurrence information. Formally, the PMI value of a process event pair E_i, E_j is computed as follows:

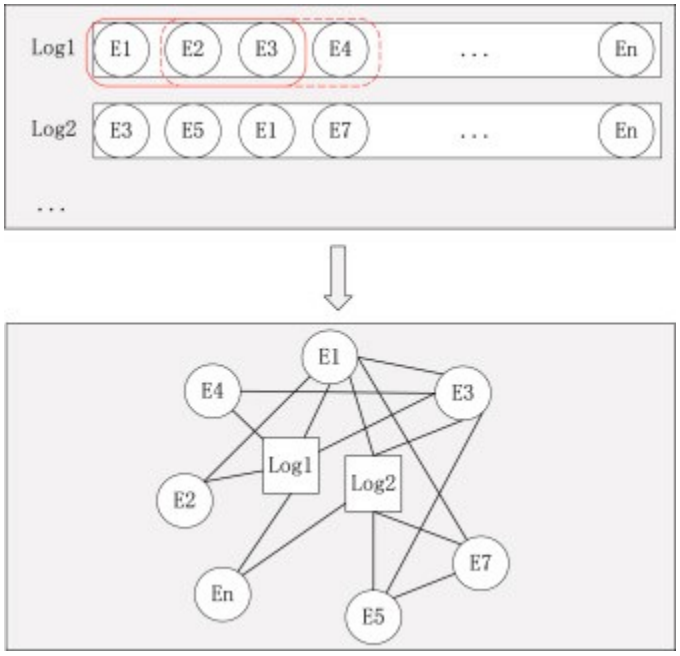
$$\mathrm{PMI}(E_i, E_j) = \log \frac{p(E_i, E_j)}{p(E_i)p(E_j)} \tag{3.5}$$

$$p(E_i, E_j) = \frac{\mathrm{count}(E_i, E_j)}{|\mathbb{W}|} \tag{3.6}$$

$$p(E_i) = \frac{\mathrm{count}(E_i)}{|\mathbb{W}|} \tag{3.7}$$

where $|\mathbb{W}|$ represents the total number of sliding windows in our corpus, $count(E_i)$ refers to the count of sliding windows that contain a process event, and $count(E_i, E_j)$ is the count of sliding windows that contain both process event E_i and E_j.

To set the weight of Type II edge, we utilize the TF-IDF (Term Frequency-Inverse Document Frequency)[3] for measurement, where TF (Term Frequency) refers to the number of times the process event in the log document, where IDF (Inverse Document Frequency) is calculated by taking the logarithm of the inverse proportion of log documents containing the process event.



Download: Download high-res image (239KB)

Download: Download full-size image

Fig. 4. The process of building a global heterogeneous graph.

After the heterogeneous graph is built, we input it to a Graph Convolutional Network (GCN) (Kipf and Welling, 2016) for generating process event embedding. According to the theory of GCN, the diagonal elements of adjacency matrix $\boldsymbol{A}$ are set to 1. Each layer of GCN is defined as follows:

$$\boldsymbol{L} = \boldsymbol{\rho}\left(\tilde{\boldsymbol{A}}\boldsymbol{X}\boldsymbol{W}\right) \tag{3.8}$$

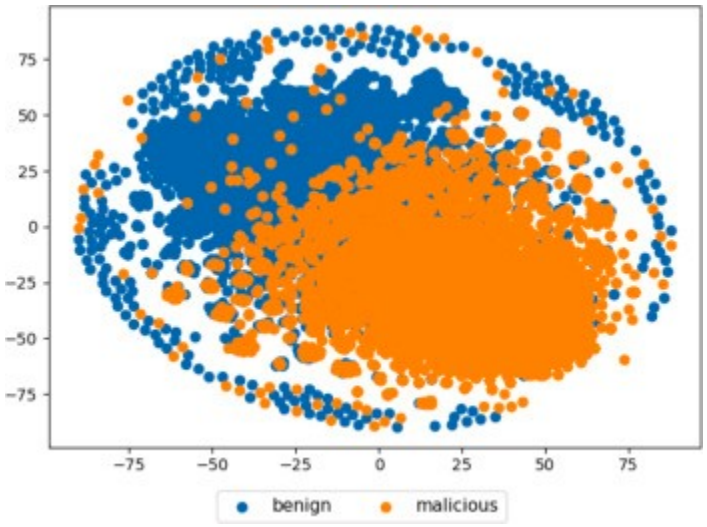
where $\tilde{\boldsymbol{A}} = \boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{A}\boldsymbol{D}^{-\frac{1}{2}}$ refers to the matrix after symmetric normalization of $\boldsymbol{A}$, D is the degree matrix, $\boldsymbol{W}$ is the parameter matrix trained in GCN, and $\boldsymbol{\rho}$ is the ReLU activation function.

To obtain the process event embedding, we make use of a two-layer GCN model. Generally, the first GCN layer is trained for getting the global semantic vectors of process events while the second GCN layer is used for log document classification. It is important to note that we do not use the label information from the test set. Instead, only the label information from the training set log documents is utilized to train the parameter matrix of the GCN through the Backpropagation (BP) algorithm.

There are two types of nodes in the heterogeneous graph: unlabeled process event nodes and labeled log document nodes. These nodes both update their feature vectors by

receiving information from their neighboring nodes during the training process. As a result, the log document nodes in the training set can be visualized to evaluate the effect of hidden layer node embedding.

As shown in Fig. 5, the log document nodes with labels in the training set are downscaled and visualized using t-SNE (Van der Maaten and Hinton, 2008). It can be seen that there is a relatively clear distinction between malicious and benign samples, so it is feasible to use the GCN hidden layer embedding as the node feature vector.



Download: Download high-res image (408KB)

Download: Download full-size image

Fig. 5. t-SNE visualization results.

3.4. Detection model construction

After the process events are vectorized, the next step is to build a detection model to perform binary classification. Previous methods (Ring et al., 2021) use only the sequential information between log events but do not consider the structural information between processes. To address this issue, we employ an attention-based gated graph neural network to construct a detection model, which considers both the invocation relationship between processes and the sequential relationship of event sequences.

3.4.1. Event relationship graph construction

To incorporate the temporal relationships within event sequences and the invocation relationships between processes, the first step is to construct a graph with process structural information for each log sample. In this study, the process structural

information related to each log sample has already been extracted during the log parsing phase. Each log sample consists of multiple process event sequences and the invocation relationships between processes. The embedding vectors of process events have been obtained using graph convolutional neural networks in the global heterogeneous graph. Therefore, constructing the event relationship graph only requires consideration of its adjacency matrix.

The event relationship graph differs from the global heterogeneous graph, which is constructed for the entire dataset. In the training process, the global graph is updated as a whole to refine node features. However, for the event relationship graph, a separate graph is built for each sample, allowing for batch training. The learned graph relationships can be directly applied to new log samples. Although new samples may contain event sequences not present in the training set, their structural information in the event relationship graph can still be extracted. Therefore, the model exhibits relatively higher robustness.

In the event relationship graph, each node represents a process event within the sample. The relationships between events need to be extracted from the event sequence information and process invocation relationships. Based on the characteristics of the log dataset, this study uses a sliding window method to establish the relationships between events. There are three main rules involved:

1. If two process events occur within the same sliding window, a connection is established by creating an edge between them.
2. The closer two events are within the sequence, the greater the weight of their edges.
3. If there is an invocation relationship between two processes, an edge is constructed for the event of the invoking process and the event being invoked.

As shown in Fig. 6, this is the process of constructing an event relationship graph for a sample. Assume that this sample extracts two related processes through log parsing and that the invocation relationship between them is that process P1 opens a new process P2 through event E2. We first construct an event E0 that opens the new process for P2. The relationship between the edges of event E2 and event E0 is the process invocation relationship, and the event sequences relationship within two processes is built using a

sliding window, which is shown to be of size 3.

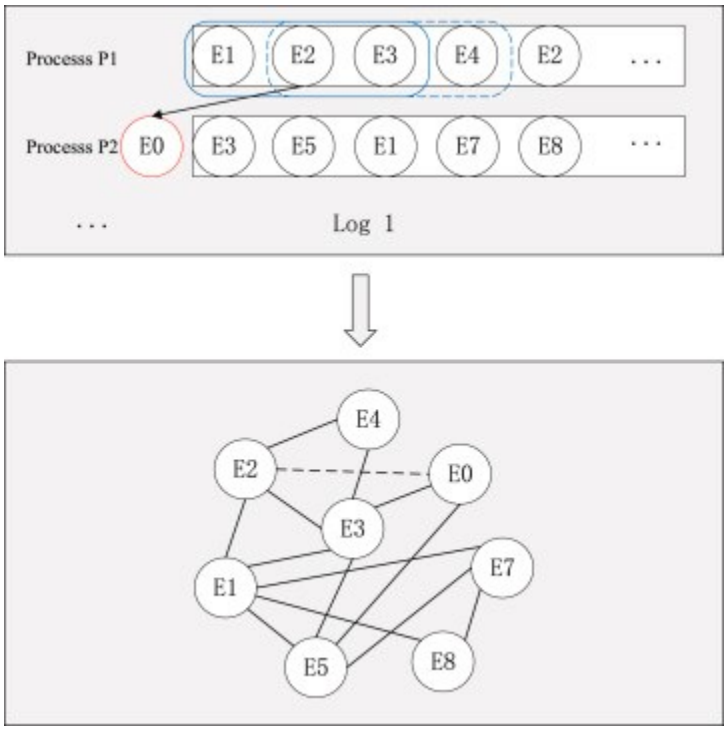
Formally, an event relationship graph can be defined as $G = (V, E)$, where V is the set of all events in the sample and E is the set of all edges. A represents its adjacent matrix. For event i and event j, the weight of their edge is calculated as follows:

$$A_{ij} = \sum_{k=1}^{n} \frac{1}{d_k(E_i, E_j)} \tag{3.9}$$

where k is the frequency of co-occurrence of event i and event j within the sliding window. $d_k(E_i, E_j)$ represents the distance between two events each time they co-occur, where the distance between two events is defined as the difference in their sequence indices. To set the weight of the edge between the events that have the invocation relationship, we use the following formula:

$$A_{\text{invoke}} = \max_{a_{ij} \in A}(a_{ij}) \tag{3.10}$$

This expression just selects the maximum value from all the edge weights calculated previously, ensuring that the most significant relationship is preserved when determining the edge weight. This approach prioritizes the most relevant invocation relationships for graph construction.



Download: Download high-res image (265KB)

Download: Download full-size image

Fig. 6. Construction of event relationship graphs.

Since the number of nodes in different event relationship graphs may vary, we select the graph with the highest number of nodes among all log samples as the standard. Let its number of nodes $|V| = n$, and each log sample has an adjacency matrix $A \in R^{n \times n}$. In other words, the number of nodes in an event relationship graph, which is less than n, has been expanded to n. The feature matrix $X \in R^{n \times d}$ represents the event embedding vectors obtained from the previous embedding model, where d is the dimension of the embedding vector.

3.4.2. Attention gated graph neural network

Once the event relationship graph has been constructed for each sample log, the graph is trained by using a message-passing-based graph neural network to aggregate information about neighboring nodes for each node. To account for the interconnectedness of events particularly those that are not adjacent, we utilize the Attention Gated Graph Neural Network (AGGNN). This graph model will assign an attention weight to each node as it aggregates information from its neighbors. As a result, the nodes in the graph would pay more attention to more important neighbors. In addition, AGGNN enhances the long-term memory ability of the network, surpasses traditional graph neural networks in depth, and is more suitable for processing data with both semantics and graph structures. Compared to models like GAT and GIN, AGGNN provides greater flexibility in learning hierarchical representations and controlling the contribution of neighboring nodes.

The internal structure of AGGNN is shown in Fig. 7. In general, the input of AGGNN comprises two parts: the state vectors of event nodes, denoted as $h \in R^{|v| \times d}$, and the information a aggregated from the adjacent neighbors. In our case, h is initialized with our pre-trained process event embedding. The computation procedure can be represented as follows:

$$a^t = \widehat{A}^{t-1} h^{t-1} W_a \tag{3.11}$$

$$z^t = \sigma\left(W_z a^t + U_z h^{t-1} + b_z\right) \tag{3.12}$$

$$r^t = \sigma\left(W_r a^t + U_r h^{t-1} + b_r\right) \tag{3.13}$$

$$\tilde{h}^t = \tanh\left(W_h a^t + U_h\left(r^t \odot h^{t-1}\right) + b_h\right) \tag{3.14}$$

$$h^t = \tilde{h}^t \odot z^t + h^{t-1} \odot (1 - z^t) + \widehat{h}^t \tag{3.15}$$

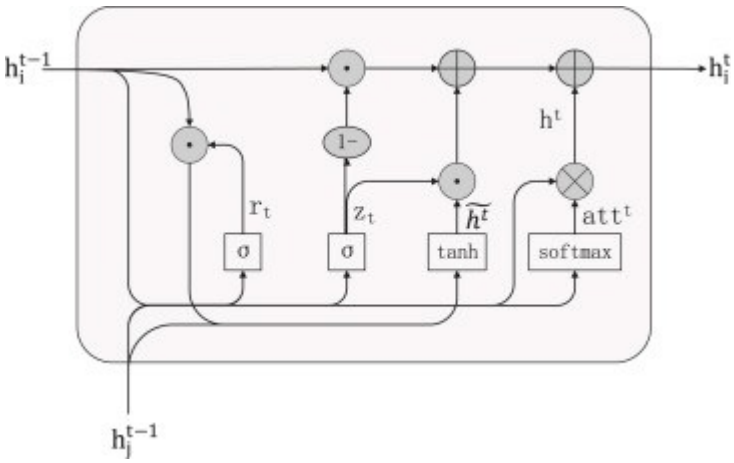
where $\widehat{A} = D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$ is the symmetric normalized adjacency matrix, D is the degree matrix, σ is the sigmoid function, $\odot$ is the Hadamard product, and W, U and b are trainable weight and bias matrices. The update gate z and reset gate r are used to determine the extent to which the neighbor node feature information contributes to the current node embedding. The attention gate att^t is designed to capture the relationship between the center node and its adjacent nodes, determining the extent to which neighbor information influences with the central node. For this purpose, the calculation of the attention score for each neighbor of the center node is carried out as follows:

$$\mathrm{att}_{v_i}^t = \mathrm{softmax}\left(h_{v_i}^{t-1}\right) = \frac{\exp\left(h_{v_i}^{t-1} W\right)}{\sum_{v_j \in N_{(v_i)}} \exp\left(h_{v_j}^{t-1} W\right)} \tag{3.16}$$

where W refers to a shared linear transformation and $N(v_i)$ represents the set of adjacent neighbors of node v_i (including v_i itself). Then, the attention-weighted embedding vector of each node v_i is calculated as follows:

$$\widehat{h}_{v_i}^t = \sum_{v_j \in N_{(v_i)}} att_{v_j}^t \cdot h_{v_j}^{t-1} \tag{3.17}$$

$\widehat{h}_{v_i}^t$ summarizes all the neighbor information of the current node v_i. After the above computation, each node feature information will be well updated by its adjacent neighbors. Particularly, the attention gate att^t boosts the transmission of semantic information from the 1-hop neighbors to the central node. One AGGNN layer enables an event node to update its feature information from the first-order neighbors. By stacking multiple AGGNN layers, we can obtain high-order feature interactions, where a process node can aggregate information from higher-order neighboring nodes. In our approach, we employ two AGGNN layers based on experimental results.



Download: Download high-res image (161KB)

Download: Download full-size image

Fig. 7. The internal structure of AGGNN.

3.4.3. Readout function

After each node feature information is sufficiently updated, we could combine these node features to get a graph-level representation for the log document. As shown in Fig. 8, the graph-level vector is composed of two parts. One part is obtained by weighting the sum of each node feature vector in the event relationship graph. The other part is to select the vector by performing max pooling on the node feature vectors within the event relationship graph. Specifically, we define the readout function as follows:

$$\bar{h} = \frac{1}{n}\sum_{i=1}^{n}\sigma(f_1(h_i)) \odot \mathbf{ReLU}(f_2(h_i)) \tag{3.18}$$

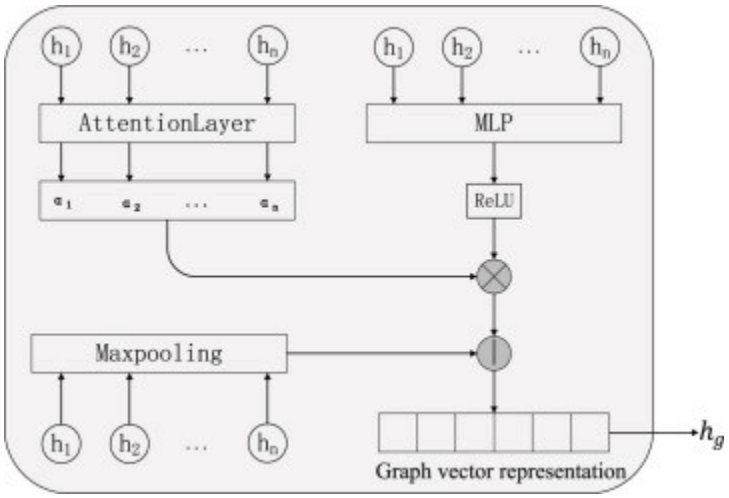
$$h_g = \bar{h} \parallel \mathbf{Maxpooling}(h_1, h_2, \ldots, h_n) \tag{3.19}$$

where $f1$ and $f2$ refer to two multilayer perceptrons (MLPs). $f1$ functions as an attention layer to learn attention scores for each event, where these attention scores represent the importance of each event within the event relationship graph. By applying the logistic sigmoid function, the attention scores will be between 0 and 1. $f2$ serves as a non-linear feature transformation. By applying the ReLU function, only the positive part in the node feature vector will be kept.

After obtaining the vector of the entire event relationship graph, we utilize a multi-layer perceptron (MLP) as the classifier. Specifically, the graph-level vector is fed into a fully connected layer, followed by the softmax activation function:

$$\hat{y}_i = \mathbf{softmax}(W h_g + b) \tag{3.20}$$

where W is the trainable weight matrix, b is the bias vector, and $\hat{y}_i$ is the final prediction vector for the log sample i. $\hat{y}_i$ is a two-dimensional vector, with each dimension corresponding to the probability of a particular class.



Download: Download high-res image (192KB)

Download: Download full-size image

Fig. 8. The internal structure of Readout module.

To measure the difference between the true labels and the predictions, we leverage the cross-entropy loss function:

$$\mathbf{Loss} = -\sum_i y_i \log \hat{y}_i \tag{3.21}$$

where y_i is the true label for the log sample i.

4. Evaluation

Our experiments are conducted on a Lenovo Legion PC. This PC is equipped with an Intel 10700K CPU, 32 GB memory, and one NVIDIA 2060 GPU. The host system is a 64-bit Ubuntu 20.04 system. The neural network model of our system is implemented in Pytorch 1.10.1.

4.1. Dataset

We use the public dataset constructed by Berlin et al. (2015), which contains the labeled Windows audit logs. To generate these logs, Berlin et al. first collect a large number of program samples and then leverage Cuckoo Sandbox to run these samples for recording their execution logs. These logs comprise individual files, with each log file containing four minutes of Windows process events. Each log file includes relevant meta information such as the recording time and the number of events.

As shown in Table 1, in our experiments, the whole dataset is split into training set, validation set and test set with a ratio of 8:1:1. The distribution of samples in each subset is consistent and relatively balanced.

Table 1. Dataset in our experiments.

Statistic	Training set	Validation set	Testing set
benign samples	8899	1112	1112
malicious samples	7561	945	945
sum	16460	2057	2057
portion	80%	10%	10%

4.2. Evaluation metrics

We make use of the standard metrics to evaluate the performance of the detection model: Accuracy, Precision, Recall, F1-Score, FPR (False Positive Rate). These metrics are defined as follows:

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN} \tag{4.1}$$

$$Precision = \frac{TP}{TP+FP} \tag{4.2}$$

$$Recall = \frac{TP}{TP+FN} \tag{4.3}$$

$$F1 = 2 * \frac{Precission*Recall}{Precission+Recall} \tag{4.4}$$

$$FPR = \frac{FP}{FP+TN} \tag{4.5}$$

In the above equation, True Positive (TP) indicates the count of malicious samples

correctly detected. False Positive (FP) represents the number of benign samples that were incorrectly identified as malicious. True Negative (TN) indicates the number of benign samples correctly identified. False Negative (FN) represents the count of malicious samples incorrectly identified as benign. Precision represents the proportion of malicious samples that are accurately detected. Accuracy measures the percentage of both benign and malicious samples that are correctly identified. Recall indicates the capability to correctly identify malicious samples. FPR (False Positive Rate) quantifies the percentage of benign samples that are mistakenly labeled as malicious. The F1 score characterizes the balanced measure of Precision and Recall, using their harmonic mean.

4.3. Effect of the event optimization methods

To optimize process events for our detection model, there are two method: event frequency-based method and information gain based method. The first method just keep the process event whose frequency is bigger than the threshold value. Table 2 shows the evaluation results of the two methods. We can see the information gain method performs better than on most evaluation metrics.

Table 2. Results under different process event optimization methods.

Optimization method	Accuracy	Precision	Recall	F1Score	FPR
Event frequency	0.9212	0.9124	**0.9181**	0.9153	0.0761
Event information gain	**0.9421**	**0.9642**	0.9083	**0.9354**	**0.0289**

4.4. Effect of the event embedding methods

In this part, we explore the effect of the GCN-based event embedding method on the detection results. For this purpose, we first conduct a word2vec model as the comparison target. This embedding method uses a corpus of event sequences from the training set to obtain the semantic vector of events. Moreover, we apply a random method to vectorize the log events randomly. Table 3 shows the comparison results. The results demonstrate the validity and necessity of using the GCN for event semantic embedding. The primary reason is that the GCN-based embedding method excels in capturing both local and global event co-occurrence information. This is achieved through the effective node message passing mechanism on the graph neural network.

Table 3. Results under different event embedding methods.

Embedding method	Accuracy	Precision	Recall	F1Score	FPR
Word2vec	0.9205	0.8976	0.9337	0.9153	0.0905
Random	0.9164	0.9066	0.9120	0.9093	0.0798
GCN	**0.9421**	**0.9642**	**0.9083**	**0.9354**	**0.0289**

4.5. Effects of the sliding window size

To examine whether the size of the sliding window affects the detection results, we conduct the experiments under different sizes of sliding windows for obtaining event embedding which will be fed to the AGGNN model. Table 4 shows the evaluation results. When the sliding window size is set to 4, the detection model can achieve optimal performance in general. Adjusting the window size appropriately ensures the model maintains a balance between capturing adequate contextual information and reducing noise in the data. Smaller window sizes might miss important interactions between events, reducing the model's ability to identify complex malicious behaviors. Larger window sizes might include too many unrelated or less relevant events, introducing noise into the graph structure.

Table 4. Experimental results of the model in the testing set under different sliding window sizes.

WindowSize	Accuracy	Precision	Recall	F1Score	FPR
2	0.9314	0.9447	0.9089	0.9264	**0.0481**
3	0.9319	0.9220	0.9590	0.9402	0.1022
4	**0.9426**	0.9212	**0.9764**	**0.9480**	0.0963
5	0.9348	0.9608	0.8965	0.9276	0.0318
6	0.9358	**0.9655**	0.8917	0.9271	0.0269
7	0.9290	0.9227	0.9256	0.9236	0.0671

4.6. Effect of the attention mechanism

To explore the effect of the attention mechanism used in readout function, we carry out experiments to compare the weighted aggregation based on the attention mechanism

and the average aggregation method without attention. As shown in Table 5, with applying the attention mechanism, the Accuracy, Precision, F1 score are all improved and the FPR is decreased. The main reason for the effectiveness of the attention mechanism is that when the vector representation of the event relationship graph is extracted, the addition of the attention mechanism allows more important nodes to influence the generation of the graph vector, enabling the detection model to focus on the most relevant features while suppressing noise.

Table 5. Results under different readout methods.

Readout	Accuracy	Precision	Recall	F1Score	FPR
Without attention mechanism	0.9159	0.9159	**0.9159**	0.9159	0.0687
With attention mechanism	**0.9421**	**0.9642**	0.9083	**0.9354**	**0.0289**

4.7. Effects of the AGGNN layer

To evaluate the impact of different graph neural network layers on the detection results, we carry out the experiments under different AGGNN layers. Table 6 illustrates the experiment results. When the AGGNN layer is set to 2, the detection model can achieve the best performance. If the AGGNN layer is just one, the node information transmission is inadequate. On the other hand, increasing the network depth would lead to the over-smoothing issue (Rusch et al., 2023).

Table 6. Experimental results of the model in the testing set under different AGGNN layers.

AGGNN layer	Accuracy	Precision	Recall	F1Score	FPR
1	0.9217	0.8870	0.9412	0.9133	**0.0934**
2	**0.9426**	**0.9212**	**0.9764**	**0.9480**	0.0963
3	0.9159	0.8699	0.9500	0.9082	0.1107
4	0.9120	0.8622	0.9512	0.9045	0.1185

4.8. Effect of the different models

Our method is evaluated against two existing methods (Berlin et al., 2015, Ring et al., 2021) that have been tested on the same dataset as ours. Berlin et al. (2015) used an n-gram method to extract continuous events as features and then employed a logistic regression model to classify audit logs. Ring et al. (2021) used an LSTM model for automatic feature extraction and classification, which resulted in an improvement of approximately 7 percent in detection accuracy compared to the findings reported by Berlin et al. (2015). Moreover, we have ported three recent methods (Deng et al., 2023, Li et al., 2022, Guo et al., 2021) to our application scenario. To evaluate our method, we conducted comparison experiments with these five methods and the gated graph neural network approach without attention mechanism (GGNN).

Ring et al. (2021) utilized a sequence of all logged events from the operating system during the runtime of the program as the initial data, which may lead to events from other processes affecting the detection. Therefore, it would be unfair to simply replicate Ring's model. As a result, the data used for the LSTM model in this paper excludes the events of irrelevant processes. This approach ensures that all conditions for the comparison process are controlled to be the same, except for the differences in the models themselves.

Table 7 showcases the comparison results, clearly demonstrating the superior performance of the AGGNN model in terms of detection recall and excelling in most other metrics. This is primarily attributed to our graph model's comprehensive utilization of sequential and structural process information. In contrast, the other methods mainly focus on the sequential relationship of the event sequence, disregarding crucial structural information such as the process invocation relationship.

We also conducted experiments to compare the robustness of the AGGNN model and the LSTM model. In this context, robustness refers to a model's ability to maintain high accuracy even when the test set includes a significant number of events that were not present in the training set. To simulate real-world scenarios where a large number of events unseen in the training set appear in the test set, we designed an experiment involving the creation of two distinct partitioned datasets, as outlined in Table 8, where the number of new events in the test set refers to the events that did not appear in the training set.

Table 7. Comparison of similar studies.

	AGGNN	GGNN	LSTM	GIN (Deng et al., 2023)	GCN (Li et al., 2022)	n-gram+LR (Berlin et al., 2015)	Host log+LSTM (Ring et al., 2021)	LogBERT (Guo et al., 2021)
Accuracy	**0.9426**	0.9259	0.9237	0.9281	0.9353	0.83	0.9114	0.9402
Recall	**0.9764**	0.9012	0.8931	0.8868	0.9376			0.9122
FPR	0.0963	0.0687	0.1271	0.0369	0.0665	**0.001**	0.1242	0.0359
F1Score	**0.9480**	0.9137	0.9149	0.9189	0.9302			0.9334

For the two datasets, the AGGNN model and the LSTM model were trained separately and then evaluated on the test set. Table 9 presents the experimental evaluation metrics for the test set. The results demonstrate that the LSTM model struggles to detect samples in Dataset 2, where a significant number of events exhibit patterns not encountered in the training set. On the other hand, the AGGNN model still maintains a relatively high level of detection accuracy with only a slight reduction. This is mainly due to the AGGNN model having both pre-trained node feature embeddings and the ability to learn structural information from event relationship graphs, enabling it to effectively aggregate information from neighboring nodes when facing new events. This experiment illustrates the proposed AGGNN model's robustness.

Table 8. Dataset partition for checking the robustness.

Dataset	Proportion		Number	
	Training set	Testing set	Events in training set	Unseen Events in testing set
Dataset 1	80%	10%	66681	5620
Dataset 2	10%	80%	12723	59269

Table 9. Experimental results of AGGNN and LSTM with different datasets.

	AGGNN in Dataset 1	LSTM in Dataset 1	AGGNN in Dataset 2	LSTM in Dataset 2
Accuracy	0.9426	0.9237	0.9162	0.2665
Recall	0.9764	0.8931	0.8848	0.0095

	AGGNN in Dataset 1	LSTM in Dataset 1	AGGNN in Dataset 2	LSTM in Dataset 2
Fl Score	0.9480	0.9149	0.9065	0.0118
FPR	0.0963	0.1271	0.0571	0.5153

4.9. Explainability of the model

As previously discussed, the attention layer within the readout function computes an attention score for each event. This score signifies the relative importance of the event within the log sample. We then extract the events that rank within the top 10 in terms of importance from the malicious log samples. Please be aware that the top 10 important events may vary across different samples. Table 10 displays the top 30 events based on their frequency in all malicious log samples, with the candidates originating from the individual top 10 important events. For easy representation, the part before the '@' delimiter is the event action field, and the part after the '@' delimiter is the event target field.

By analyzing the event semantics, we could get the key insight of the program behavior. For example, process-related events, such as "WINDOWSPROCESSClose@sha1", "WINDOWSStartNewProcess@sha1", and "WINDOWSPROCESSSpawn[temp]@sha1" signify the prevalence of multi-process behavior in malware, consistent with the conclusions drawn by Peeler (Ahmed et al., 2021). Events such as "WINDOWSFILEExecute[system]@cmd.exe" and "WINDOWSFILEExecute[system]@powershell.exe" indicate malicious actions, with the latter being particularly concerning due to its flexibility, which attackers often exploit for malicious downloads (Hendler et al., 2018). Suspicious behaviors encompass executing files from temporary folders and modifying previously executed files (Berlin et al., 2015).

Table 10. Top 30 important event features of malicious samples.

Event	Frequency
WINDOWSFILEExcute[system]@wshtcpip.dll	554
WINDOWSFILEExcute[system]@sechost.dll	524
WINDOWSPROCESSClose@sha1	517
WINDOWSFILEWrite[cuckoo]@sha1	487

Event	Frequency
WINDOWSFILEExcute[system]@cryptbase.dll	477
WINDOWSFILEExcute[system]@cryptsp.dll	474
WINDOWSFILEExcute[system]@imm32.dll	424
WINDOWSFILEExcute[windows]@system.directoryserices.ni.dll	410
WINDOWSFILEExcute[system]@shdocvw.dll	403
WINDOWSFILEExcute[windows]@system.transactions.dll	402
WINDOWSFILEPremissions[cuckoo]@sha1	398
WINDOWSFILEExcute[system]@uxtheme.dll	368
WINDOWSStartNewProcess@sha1	366
WINIX)WSFILEExcute[system]@userenv.dll	364
WINDOWSFILEExcute[system]@cmd.exe	363
WINDOWSFILEExcute[system]@powershell.exe	361
WINIX)WSFILEExcute[system]@winbrand.dll	360
WINIX)WSFILEExcute[system]@ntmarta.dll	338
WINDOWSFILEExcute[system]@version.dll	297
WINDOWSFILEExcutingEditedFile@sha1	285
WINDOWSFILEExcute[system]@atl.dll	275
WINIX)WSFILEExcute[system]@mscoree.dll	272
WINDOWSFILEExcute[temp]@sha1	269
WINDOWSFILEExcute[system]@apphelp.dll	267
WINDOWSFILEExcute[windows]@system.configuration.install.ni.dll	262
WINDOWSFILEExcute[system]@sspicli.dll	250
WINEX)WSFILEExcute[windows]@system.ni.dll	248
WINDOWSFILEExcute[system]@iphlpapi.dll	242
WINDOWSPROCESSSpawn[temp]@sha1	240

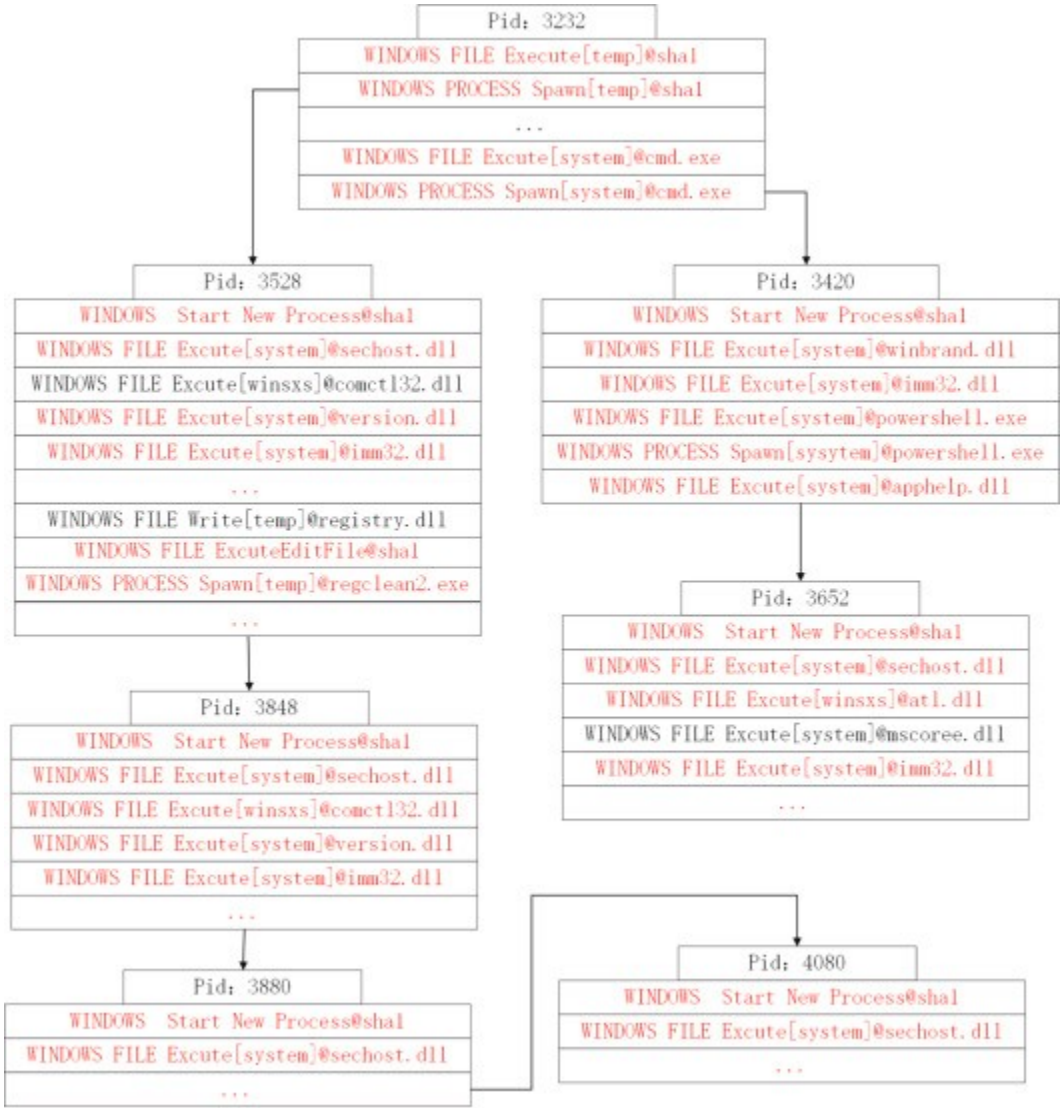
For some targets (i.e., certain DLL files) that cannot be explained directly, we leverage the well-known online malware analysis service HybridAnalysis[4] to search the DLL information. As shown in Table 11, the malicious event targets extracted by the attention scores are labeled as malicious by most Antivirus (AV) engines. Furthermore, by using domain knowledge, we find some DLL files are usually used for malicious purposes. For example, ransomware use the cryptsp.dll for encrypting files; Trojan applies the wshtcpip.dll for network communication.

Case Study. Fig. 9 shows the process tree structure of the malicious sample and the log event sequences for each process. The red log event in the figure represents a highly malicious event obtained in this paper. Due to the large number of log events for each process, most of the unimportant events have been omitted. From the process tree structure, it can be observed that the malware has spawned multiple processes to carry out malicious functionalities. Additionally, upon analyzing the specific sample, the process with PID 3580 wrote a temporary file in the temporary folder and subsequently executed it, which is clearly a malicious operation. The process with PID 3420 executed a CMD process and then initiated a process to run a PowerShell script. Although the logs do not provide details about the command line or the specific operations of the PowerShell script, it does indicate that it called certain DLLs related to encryption, which is suspicious. The process with PID 3880 modified a registry entry and launched a background process. The purpose of the modified registry entry is to enable a process to start automatically during system boot-up, which is highly suspicious behavior. From this case, we could see our method based on the attention mechanism could provide a certain level of assistance in analyzing the malicious behavior of individual samples.

Table 11. The DLL analysis results provided by Hybrid Analysis website.

DLL	Threat score(100)	AV detection	Label
cryptsp.dll	100	51%	Lazy.Generic
cryptbase.dll	100	38%	Malware.Generic
uxtheme.dll	100	57%	Ulise.Generic
winbrand.dll	100	77%	Trojan.Agent
version.dll	100	77%	Trojan.Generic
atl.dll	93	41%	Trojan.Heur.Generic

DLL	Threat score(100)	AV detection	Label
ntmarta.dll	74	47%	Graftor.Generic
mscoree.dll	100	68%	Bulz.Generic
apphelp.dll	100	66%	WebToolbae.SearchSuite
sspicli.dll	84	54%	Trojan.Generic
iphlpapi.dll	95	70%	Ulise.Generic



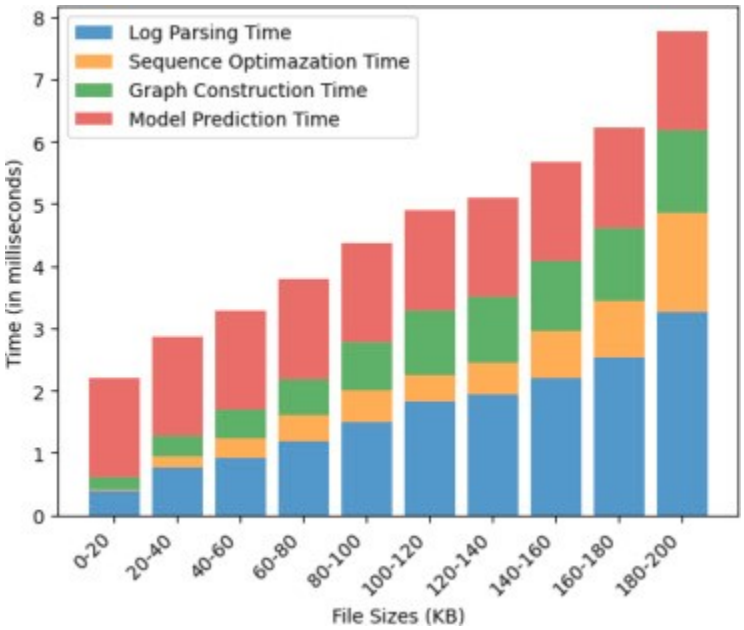
Download: Download high-res image (944KB)

Download: Download full-size image

Fig. 9. A sample log analysis.

4.10. Performance cost

To evaluate the processing time of handling the audit log file, we first randomly select 2000 different log files from the dataset. Then, we group the files according to their size. After that, we measure the average execution time for processing these log files. The total execution time encompasses several components: the time taken for log parsing, event sequence optimization, graph construction, and model prediction. Fig. 10 shows the measurement results on different files. We can see that our method analyzes a log file of less than 200KB in under 8 ms.



Download: Download high-res image (248KB)

Download: Download full-size image

Fig. 10. Measurement results from log file processing.

5. Discussions

In this paper, there are several areas for improvement. Firstly, in terms of log parsing, our current approach focuses solely on extracting information from the action and target fields within the log events. However, log data often contains valuable timestamp information, which could be utilized to enhance our understanding by revealing sequential patterns within process event sequences. Incorporating this sequential dimension could lead to more insightful analysis. Secondly, in model interpretation, our current methodology identifies individual suspicious event features. However, it is important to acknowledge that malicious activities often manifest as continuous

sequences of event actions. Therefore, we should mine patterns within event sequences to provide a more comprehensive and convincing interpretation of malicious behavior. Thirdly, in constructing the event relationship graph, we primarily rely on process invocation and event sequence information. Sophisticated adversaries may exploit this information to bypass our detection. To address this issue, we aim to enhance the graph's edge connectivity and incorporate graph structure regularization. These improvements are expected to boost both the detection accuracy and robustness of our approach. Finally, while we prioritize interpretability through the attention mechanism, increasing model complexity (e.g., multi-layer attention mechanisms) could improve detection accuracy but might hinder explainability. In future work, we aim to strategically balance model complexity with interpretability requirements.

6. Conclusion

In this paper, we present a novel malware detection method by analyzing audit logs. This method first parses the logs to extract the process event sequence and invocation relationship among different processes. To vectorize process events, we utilize a graph convolutional network. Next, an event relationship graph is constructed based on the process structure and event semantic information for each log sample. To well analyze the event relationship graphs, we make use of an attention gated graph neural network. The evaluation results show that our method can achieve high detection accuracy and provide explainable detection results on the public dataset. Moreover, the comparison experiments illustrate that our solution outperforms the recent log-based methods.

CRediT authorship contribution statement

Yewei Zhen: Writing – original draft, Visualization, Conceptualization. **Donghai Tian:** Writing – review & editing, Methodology. **Xiaohu Fu:** Visualization, Software, Data curation. **Changzhen Hu:** Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work is partly supported by the Intelligent Policing Key Laboratory of Sichuan

Province , No. ZNJW2025KFQN008.

Recommended articles

Data availability

The authors are unable or have chosen not to specify which data has been used.

References

Abdelsalam et al., 2018 Mahmoud Abdelsalam, Ram Krishnan, Yufei Huang, Ravi Sandhu
Malware detection in cloud infrastructures using convolutional neural networks
2018 IEEE 11th International Conference on Cloud Computing, CLOUD, IEEE (2018), pp. 162-169
Crossref ↗ View in Scopus ↗ Google Scholar ↗

Ahmed et al., 2009 Ahmed, Faraz, Hameed, Haider, Shafiq, M. Zubair, Farooq, Muddassar, 2009. Using Spatio-Temporal Information in API Calls with Machine Learning Algorithms for Malware Detection. In: Proceedings of the 2nd ACM Workshop on Security and Artificial Intelligence. AISec '09, pp. 55–62.
Google Scholar ↗

Ahmed et al., 2021 Muhammad Ejaz Ahmed, Hyoungshick Kim, Seyit Camtepe, Surya Nepal
Peeler: Profiling kernel-level events to detect ransomware
European Symposium on Research in Computer Security, Springer (2021), pp. 240-260
View in Scopus ↗ Google Scholar ↗

Bansal et al., 2023 Ayoosh Bansal, Anant Kandikuppa, Monowar Hasan, Chien-Ying Chen, Adam Bates, Sibin Mohan
System auditing for real-time systems
ACM Trans. Priv. Secur., 26 (4) (2023)
Google Scholar ↗

Bao et al., 2018 Liang Bao, Qian Li, Peiyao Lu, Jie Lu, Tongxiao Ruan, Ke Zhang
Execution anomaly detection in large-scale systems through console log analysis
J. Syst. Softw., 143 (2018), pp. 172-186
Publisher: Elsevier

View PDF View article View in Scopus ↗ Google Scholar ↗

Berlin et al., 2015 Berlin, Konstantin, Slater, David, Saxe, Joshua, 2015. Malicious behavior detection using windows audit logs. In: Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security. pp. 35–44.

Google Scholar ↗

Bertero et al., 2017 Christophe Bertero, Matthieu Roy, Carla Sauvanaud, Gilles Trédan

Experience report: Log mining using natural language processing and application to anomaly detection

2017 IEEE 28th International Symposium on Software Reliability Engineering, ISSRE, IEEE (2017), pp. 351-360

Crossref ↗ View in Scopus ↗ Google Scholar ↗

Burnap et al., 2018 Pete Burnap, Richard French, Frederick Turner, Kevin Jones

Malware classification using self organising feature maps and machine activity data

Comput. Secur., 73 (2018), pp. 399-410

Publisher: Elsevier

View PDF View article View in Scopus ↗ Google Scholar ↗

Canzanese et al., 2015 Canzanese, R., Mancoridis, S., Kam, M., 2015. System Call-Based Detection of Malicious Processes. In: 2015 IEEE International Conference on Software Quality, Reliability and Security. pp. 119–124.

Google Scholar ↗

Chen et al., 2023 Chen, Xiaohui, Cui, Lei, Wen, Hui, Li, Zhi, Zhu, Hongsong, Hao, Zhiyu, Sun, Limin, 2023. MalAder: Decision-Based Black-Box Attack Against API Sequence Based Malware Detectors. In: 2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks. DSN, pp. 165–178.

Google Scholar ↗

Christodorescu et al., 2005 Christodorescu, M., Jha, S., Seshia, S.A., Song, D., Bryant, R.E., 2005. Semantics-aware malware detection. In: 2005 IEEE Symposium on Security and Privacy. S P'05, pp. 32–46.

Google Scholar ↗

Deng et al., 2023 Liting Deng, Hui Wen, Mingfeng Xin, Hong Li, Zhiwen Pan, Limin Sun

Enimanal: Augmented cross-architecture IoT malware analysis using

graph neural networks

Comput. Secur., 132 (2023), Article 103323

View PDF View article View in Scopus ↗ Google Scholar ↗

Du and Li, 2016 Min Du, Feifei Li

Spell: Streaming parsing of system event logs

2016 IEEE 16th International Conference on Data Mining, ICDM, IEEE (2016), pp. 859-864

Crossref ↗ View in Scopus ↗ Google Scholar ↗

Du et al., 2017 Du, Min, Li, Feifei, Zheng, Guineng, Srikumar, Vivek, 2017. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 1285–1298.

Google Scholar ↗

Fadadu et al., 2019 Fenil Fadadu, Anand Handa, Nitesh Kumar, Sandeep Kumar Shukla

Evading API call sequence based malware classifiers

Information and Communications Security (2019), pp. 18-33

Google Scholar ↗

Guo et al., 2021 Guo, Haixuan, Yuan, Shuhan, Wu, Xintao, 2021. LogBERT: Log Anomaly Detection via BERT. In: 2021 International Joint Conference on Neural Networks. IJCNN, pp. 1–8.

Google Scholar ↗

Han et al., 2019 Weijie Han, Jingfeng Xue, Yong Wang, Zhenyan Liu, Zixiao Kong

MalInsight: A systematic profiling based malware detection framework

J. Netw. Comput. Appl., 125 (2019), pp. 236-250

View PDF View article View in Scopus ↗ Google Scholar ↗

Hendler et al., 2018 Hendler, Danny, Kels, Shay, Rubin, Amir, 2018. Detecting malicious powershell commands using deep neural networks. In: Proceedings of the 2018 on Asia Conference on Computer and Communications Security. pp. 187–197.

Google Scholar ↗

Huang et al., 2019 Lianzhe Huang, Dehong Ma, Sujian Li, Xiaodong Zhang, Houfeng Wang

Text level graph neural network for text classification

(2019)

arXiv preprint arXiv:1910.02356 ↗

Google Scholar ↗

Jindal et al., 2019 Jindal, Chani, Salls, Christopher, Aghakhani, Hojjat, Long, Keith, Kruegel, Christopher, Vigna, Giovanni, 2019. Neurlux: Dynamic Malware Analysis without Feature Engineering. In: Proceedings of the 35th Annual Computer Security Applications Conference. ACSAC '19, pp. 444–455.

Google Scholar ↗

Kalash et al., 2018 Kalash, Mahmoud, Rochan, Mrigank, Mohammed, Noman, Bruce, Neil D.B., Wang, Yang, Iqbal, Farkhund, 2018. Malware classification with deep convolutional neural networks. In: 2018 9th IFIP International Conference on New Technologies, Mobility and Security. NTMS, pp. 1–5.

Google Scholar ↗

Kipf and Welling, 2016 Thomas N. Kipf, Max Welling

Semi-supervised classification with graph convolutional networks

(2016)

abs/1609.02907

Google Scholar ↗

Kong and Yan, 2013 Kong, Deguang, Yan, Guanhua, 2013. 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. pp. 1357–1365.

Google Scholar ↗

Lanzi et al., 2010 Lanzi, Andrea, Balzarotti, Davide, Kruegel, Christopher, Christodorescu, Mihai, Kirda, Engin, 2010. AccessMiner: Using System-Centric Models for Malware Protection. In: Proceedings of the 17th ACM Conference on Computer and Communications Security. CCS '10, pp. 399–412.

Google Scholar ↗

Li et al., 2019 Teng Li, Jianfeng Ma, Qingqi Pei, Yulong Shen, Chi Lin, Siqi Ma, Mohammad S Obaidat

AClog: Attack chain construction based on log correlation

2019 IEEE Global Communications Conference, GLOBECOM, IEEE (2019), pp. 1-6

Google Scholar ↗

Li et al., 2022 Shanxi Li, Qingguo Zhou, Rui Zhou, Qingquan Lv

Intelligent malware detection based on graph convolutional network

J. Supercomput., 78 (2022)

Google Scholar ↗

Liu et al., 2019 Liu, Fucheng, Wen, Yu, Zhang, Dongxue, Jiang, Xihe, Xing, Xinyu, Meng, Dan, 2019. Log2vec: A heterogeneous graph embedding based approach for detecting cyber threats within enterprise. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 1777–1794.

Google Scholar ↗

Ma et al., 2019 Xin Ma, Shize Guo, Haiying Li, Zhisong Pan, Junyang Qiu, Yu Ding, Feiqiong Chen

How to make attention mechanisms more practical in malware classification

IEEE Access, 7 (2019), pp. 155270-155280

Crossref ↗ View in Scopus ↗ Google Scholar ↗

Ma et al., 2015 Ma, Shiqing, Lee, Kyu Hyung, Kim, Chung Hwan, Rhee, Junghwan, Zhang, Xiangyu, Xu, Dongyan, 2015. Accurate, low cost and instrumentation-free security audit logging for windows. In: Proceedings of the 31st Annual Computer Security Applications Conference. pp. 401–410.

Google Scholar ↗

Van der Maaten and Hinton, 2008 Laurens Van der Maaten, Geoffrey Hinton

Visualizing data using t-SNE

J. Mach. Learn. Res., 9 (11) (2008)

Google Scholar ↗

Moskovitch et al., 2008 Robert Moskovitch, Clint Feher, Nir Tzachar, Eugene Berger, Marina Gitelman, Shlomi Dolev, Yuval Elovici

Unknown malcode detection using OPCODE representation

Intelligence and Security Informatics (2008), pp. 204-215

Crossref ↗ View in Scopus ↗ Google Scholar ↗

Naeem et al., 2020 Hamad Naeem, Farhan Ullah, Muhammad Rashid Naeem, Shehzad Khalid, Danish Vasan, Sohail Jabbar, Saqib Saeed

Malware detection in industrial internet of things based on hybrid image visualization and deep learning model

Ad Hoc Netw., 105 (2020), Article 102154

View PDF View article View in Scopus ↗ Google Scholar ↗

Nandi et al., 2016 Nandi, Animesh, Mandal, Atri, Atreja, Shubham, Dasgupta, Gargi B., Bhattacharya, Subhrajit, 2016. Anomaly detection using program control flow graph mining from execution logs. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 215–224.

Google Scholar ↗

Narouei et al., 2015 Masoud Narouei, Mansour Ahmadi, Giorgio Giacinto, Hassan Takabi, Ashkan Sami

DLLMiner: structural mining for malware detection

Secur. Commun. Netw., 8 (18) (2015), pp. 3311-3322

Crossref ↗ View in Scopus ↗ Google Scholar ↗

Ni et al., 2018 Sang Ni, Quan Qian, Rui Zhang

Malware identification using visualization images and deep learning

Comput. Secur., 77 (2018)

Google Scholar ↗

Oliveira and Sassi, 2019 Angelo Oliveira, R. Sassi

Behavioral malware detection using deep graph convolutional neural networks

(2019)

TechRxiv 2019

Google Scholar ↗

Rhode et al., 2018 Matilda Rhode, Pete Burnap, Kevin Jones

Early-stage malware prediction using recurrent neural networks

Comput. Secur., 77 (2018), pp. 578-594

Publisher: Elsevier

View PDF View article View in Scopus ↗ Google Scholar ↗

Ring et al., 2021 Markus Ring, Daniel Schlör, Sarah Wunderlich, Dieter Landes, Andreas Hotho

Malware detection on windows audit logs using LSTMs

Comput. Secur., 109 (2021), Article 102389

Publisher: Elsevier

View PDF View article View in Scopus ↗ Google Scholar ↗

Roy and Chen, 2021 Krishna Chandra Roy, Qian Chen

Deepran: Attention-based bilstm and crf for ransomware early detection and classification

Inf. Syst. Front., 23 (2) (2021), pp. 299-315
Publisher: Springer

Crossref ↗ View in Scopus ↗ Google Scholar ↗

Rusch et al., 2023 T. Konstantin Rusch, Michael M. Bronstein, Siddhartha Mishra
A survey on oversmoothing in graph neural networks
(2023)

Google Scholar ↗

S.L and CD, 2019 S.D. S.L, J. CD
Windows malware detector using convolutional neural network based on visualization images
IEEE Trans. Emerg. Top. Comput. (2019)

Google Scholar ↗

Sun and Qian, 2018 G. Sun, Q. Qian
Deep learning and visualization for identifying malware families
IEEE Trans. Dependable Secur. Comput. (2018)

Google Scholar ↗

Verma and Bridges, 2018 Miki E. Verma, Robert A. Bridges
Defining a metric space of host logs and operational use cases
2018 IEEE International Conference on Big Data, Big Data, IEEE (2018), pp. 5068-5077

Crossref ↗ View in Scopus ↗ Google Scholar ↗

Xiaofeng et al., 2019 Lu Xiaofeng, Jiang Fangshuo, Zhou Xiao, Yi Shengwei, Sha Jing, Pietro Lio
ASSCA: API sequence and statistics features combined architecture for malware detection
Comput. Netw., 157 (2019), pp. 99-111

View PDF View article View in Scopus ↗ Google Scholar ↗

Xiong et al., 2022 Chunlin Xiong, Tiantian Zhu, Weihao Dong, Linqi Ruan, Runqing Yang, Yueqiang Cheng, Yan Chen, Shuai Cheng, Xutong Chen
Conan: A practical real-time APT detection system with high accuracy and efficiency
IEEE Trans. Dependable Secur. Comput., 19 (1) (2022), pp. 551-565

Crossref ↗ View in Scopus ↗ Google Scholar ↗

Yakura et al., 2018 Yakura, Hiromu, Shinozaki, Shinnosuke, Nishimura, Reon, Oyama,

Yoshihiro, Sakuma, Jun, 2018. Malware Analysis of Imaged Binary Samples by Convolutional Neural Network with Attention Mechanism. In: Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy. pp. 127–134.

Google Scholar ↗

Yao et al., 2019 Yao, Liang, Mao, Chengsheng, Luo, Yuan, 2019. Graph convolutional networks for text classification. In: Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 33. pp. 7370–7377, Issue: 01.

Google Scholar ↗

Yen et al., 2019 Steven Yen, Melody Moh, Teng-Sheng Moh

Causalconvlstm: Semi-supervised log anomaly detection through sequence modeling

2019 18th IEEE International Conference on Machine Learning and Applications, ICMLA, IEEE (2019), pp. 1334-1341

Crossref ↗ View in Scopus ↗ Google Scholar ↗

Yuan et al., 2019 Yue Yuan, Han Anu, Wenchang Shi, Bin Liang, Bo Qin

Learning-based anomaly cause tracing with synthetic analysis of logs from multiple cloud service components

2019 IEEE 43rd Annual Computer Software and Applications Conference, Vol. 1, COMPSAC, IEEE (2019), pp. 66-71

Crossref ↗ View in Scopus ↗ Google Scholar ↗

Zhang et al., 2020 Yingjun Zhang, Shangqi Liu, Mu Yang, Haixia Zhang, Kezhen Huang

Survey on anomaly detection technology based on logs

Chin. J. Netword Inf. Secur., 6 (6) (2020), pp. 1-12

View PDF View article Google Scholar ↗

Zhang et al., 2019 Zhang, Xu, Xu, Yong, Lin, Qingwei, Qiao, Bo, Zhang, Hongyu, Dang, Yingnong, Xie, Chunyu, Yang, Xinsheng, Cheng, Qian, Li, Ze, et al., 2019. Robust log-based anomaly detection on unstable log data. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 807–817.

Google Scholar ↗

Zhou et al., 2023 Bo Zhou, Hai Huang, Jun Xia, Donghai Tian

A novel malware detection method based on API embedding and API

parameters
J. Supercomput. (2023)
Google Scholar ↗

Cited by (0)

Yewei Zhen received her B.S. degree in Capital University of Economics and Business in 2021. She is currently working towards the master degree in School of Computer Science, Beijing Institute of Technology. Her research interests include software security and artificial intelligence.

Donghai Tian is an Associate Professor in School of Computer Science, Beijing Institute of Technology, China. His research interest lies in security issues in computer systems and networks, including areas ranging from operating system security, software security and network security, to virtualization technologies. He is a key member of Beijing Key Laboratory of Software Security Engineering Technique. From 2009 to 2011, he was a visiting student in Pennsylvania State University, USA. He received his Ph.D. degree from Beijing Institute of Technology, China in 2012.

Xiaohu Fu received his B.S. degree in Beijing Institute of Technology in 2021, and the M.S. degree in Beijing Institute of Technology in 2023. His research interests include malware detection and software security.

Changzhen Hu received his Ph.D. degree from Beijing Institute of Technology, China in 1996. He holds the Professor, Beijing Institute of Technology, China. He is also the director of Beijing Key Laboratory of Software Security Engineering Technique. His research interest includes information security, computer networks, and pattern recognition. He is the author of over 200 refereed papers in conferences and journals.

1 https://www.itu.int/itu-d/reports/statistics/facts-figures-2022/ ↗.

2 https://en.wikipedia.org/wiki/Entropy_(information_theory) ↗.

3 https://en.wikipedia.org/wiki/tf-idf ↗.

4 https://www.hybrid-analysis.com/ ↗.

View Abstract

© 2025 Elsevier Ltd. All rights are reserved, including those for text and data mining, AI training, and similar

technologies.

ELSEVIER

All content on this site: Copyright © 2025 Elsevier B.V., its licensors, and contributors. All rights are reserved, including those for text and data mining, AI training, and similar technologies. For all open access content, the relevant licensing terms apply.

RELX™