COSC 5P30 - Project Report

Madeline Janecek
mj17th@brocku.ca
Student #: 6436620

Project Definition

Introduction

System calls are the lowest level of communication between the user and kernel levels, and as such, the data generated from system calls can be an incredibly valuable resource for identifying abnormal execution behaviour (Khraisat et al. 2019). Several open-source tracing tools, such as LTTng (Desnoyers and Dagenais 2008), are capable of recording when these system calls occur while imposing minimal overhead. However, the primary drawback of system call analysis is that, depending on the system, thousands of system calls may execute every second. Consequently, the collected sequence data is too large and inherently complex for manual evaluation. Accordingly, numerous studies have set out to evaluate the effectiveness of different machine learning methods for automating the detection of anomalies within system call sequences (Khraisat et al. 2019; Cornelissen et al. 2009) This project sets out to explore if utilizing graph neural networks (GNN) can improve upon existing approaches.

Motivation

Many of the aforementioned anomaly detection techniques convert the sequences into a vector format indicating the number of times each system call occurred within a given time period (Cavalcanti, Inacio, and Freire 2021; Kohyarnejadfard et al. 2021). This process is necessary as most standard machine learning models expect fixed-sized vectors as input. However, this vectorization process results in the loss of valuable information, such as the ordering of system calls within the sequence. By ignoring this critical aspect of the system call sequences, the resulting vectors may not be fully capable of capturing the relationships and dependencies among system calls that may be relevant in detecting anomalous behaviours. Therefore, alternative and more effective methods are still needed to fully utilize system call sequences.

Problem Definition

The main objective of this project was to leverage the expressive nature of graphs to enhance existing system call anomaly detection techniques, using a similar method to the log anomaly detection approach described in (Zhang et al. 2022a). Firstly, a set of normal system call sequences are converted into a graph format that is representative of the events and the transitions between them. These graphs are then passed on a GNN model, and the resulting graph embeddings are used to train a standard machine learning anomaly detection algorithm. Currents results show that this approach can not effectively identify anomalous system call sequences, however, the approach may be improved upon in future works.

Background and Related Works

GNN Based Anomaly Detection

Graph Neural Networks (GNNs) are a class of neural network models that are capable of capturing both the structure and node features of a graph. GNNs allow for analyzing data that is more naturally represented as a graph, such as molecular structures, social networks, and so forth. In recent years, several studies have explored the effectiveness of GNN models in various anomaly detection tasks across different domains. For example, in (Deng and Hooi 2021), the authors convert high-dimensional time series samples into graphs and train a GNN model to classify samples as either normal or anomalous. The sucess seen in these types of works demonstrate the potential of GNNs in detecting anomalies in complex data.

Although the use of GNN models has not been explored for identifying anomalies in system call data, they have shown promise in identifying system anomalies in higher-level log data. For instance, DeepTraLog (Zhang et al. 2022a) is a gated graph neural network (GGNN)-based anomaly detection approach that identifies anomalous logs in distributed microservice systems. Another approach, PU-TraceAD (Zhang et al. 2022c), similarly leverages GNNs to detect anomalies in distributed systems, however it uses positive-unlabeled (PU) learning to bypass the need for large labeled training datasets. Furthermore, in (Xie, Zhang, and Babar 2022) the authors present LogGD, an approach that

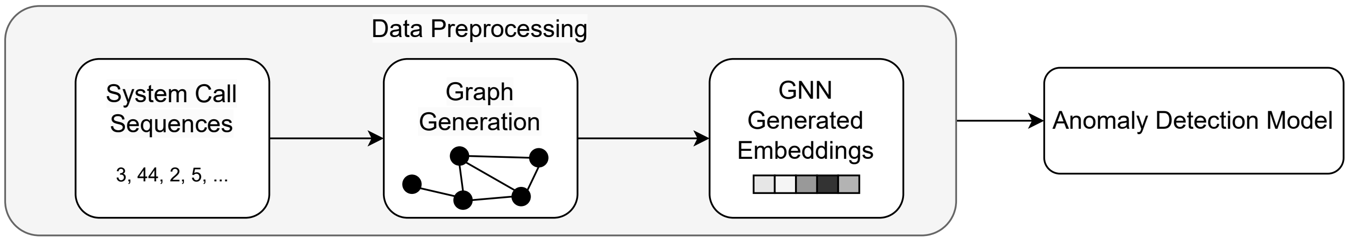

Figure 1: Visual overview of the proposed approach.

uses a Graph Transformer Neural Network to detect anomalous logs, and show how it outperforms state of the art quantitative and sequence-based methods. These studies show that GNNs may effectively identify anomalies in complex software systems, and suggest that they may be used with system call sequences.

System Call Tracing

System call tracing is a well-established technique used to monitor and record the interactions between software applications and the underlying operating system. Tracing tools, such as LTTng (Desnoyers and Dagenais 2008), intercept the system calls made by an application and then record details such as the system call type, timestamp, arguments, and so forth. What system calls are being called, how often they are being called, and their executions' duration can all provide valuable insights that may be used to identify performance bottlenecks (Ezzati-Jivan et al. 2020), localize the root cause of performance bugs (Kohyarnejadfard et al. 2021), identify breaches in security (Khraisat et al. 2019), and so forth.

System Call Anomaly Detection

The concept of identifying anomalies in raw system calls to implement a host-based intrusion detection system (HIDS) was first proposed in (Forrest et al. 1996). Since then, the use of system calls has been adopted in various security-related scenarios (Forrest et al. 1996; Khraisat et al. 2019; Zhang et al. 2020; Cavalcanti, Inacio, and Freire 2021) as well as for performance analysis purposes (Kohyarnejadfard et al. 2021; Fournier et al. 2021).

The central premise behind these works is based on the idea that a normally running system generates consistent sequences of system calls. If there is a performance issue or malicious attack, the resulting system calls will differ from the expected norm. Differences may include the presence of a normally unused system call, an unlikely sequence of system calls, abnormally long system calls, and so forth. By identifying these deviations, security and performance analysts can gain valuable insights into the behaviour of their system.

Some past system call anomaly detection methods have taken a graph-based approach where feature vectors describing graph representations of the system call sequences are used to train classification models. For instance, in (Surendran and Thomas Kallivayalil 2022) the authors create graphs where nodes represent system calls and edges indicate transitions between those calls. They then utilize centrality measures taken from those graphs as input for various machine learning classifiers. Similarly, in (Grimmer et al. 2018) the authors convert n-gram system call sequences into graphs and then employ feature engineering to identify anomalies. These approaches showcase the descriptive nature of system call sequences graphs, thereby indicating the potential of a GNN-based approach.

Approach

A visual overview of the proposed approach, which is described in more detail in the following sections, is shown in Figure 1. The process begins by taking the system call sequences and converting them into a graph format. These transition graphs are meant to capture the events in the sequence, as well as the transitions between events. Then, a Graph Convolutional Network (GCN) (Zhang et al. 2022b) is employed to convert the graph into a vector format that can be used for machine learning. Using a set of these graph embeddings, a one-class Support Vector Machines (SVM) model (Hu, Hu, and Du 2019) is trained to learn the normal execution behavior of the system. Finally, this trained model is used to classify new executions as either normal or anomalous based on their graph embeddings.

Graph Creation

For each system call sequence, a transition graph $G = (V, E)$ is generated to represent the system call events and the transitions between them. The set of all possible system calls is denoted as S. Each transition graph has $|S|$ nodes, and each node $v_i \in V$ corresponds to one specific system call $s_i \in S$. Let a system call sequence be denoted as $C = \{c_1, c_2, ..., c_{n-1}, c_n\}$, where each event $c_i \in C$ also corresponds to a specific system call $s_i \in S$. Multiple events within C may be associated with the same system call (i.e. a system call may have been called multiple times throughout the execution).

Each of the edges within a system call transition graph are indicative of transitions within the original sequence. In other words, an edge between the nodes v_a and v_b is present if the subsequence $C_{a,b} = \{c_a, c_b\}$ is found within C. The weight of each edge $w_{a,b}$ is set to the number of times the corresponding subsequence $C_{a,b}$ is found within

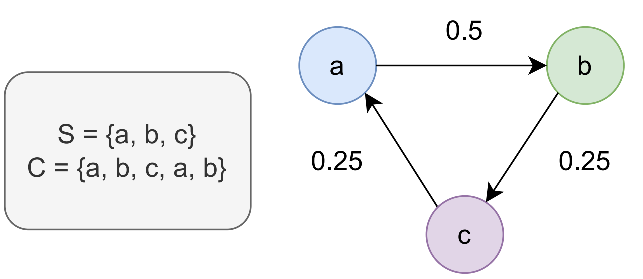

Figure 2: A simplified example illustrating the transition graph generation process.

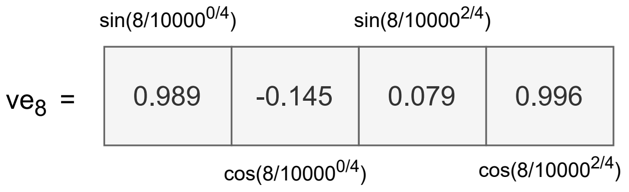

Figure 3: Four dimensional ($d = 4$) encoding of the 8th system call ($y = 8$).

the main sequence C, divided by the total number of transitions within the sequence (i.e. $|C| - 1$). A visual overview of this part of the graph generation process is shown in Figure 2.

When it comes to the transition graph's node features, a node $v_y \in V$ is given a d dimensional encoding of its corresponding system call, which will be denoted as ve_y. The encoding technique used here was first proposed in (Vaswani et al. 2017), and has since been used in other system call analysis works (Fournier et al. 2021). The method consists of alternating between a sine and cosine functions to compute the vector's features. That is, the x^{th} element in the d dimensional encoding of the y^{th} system call is calculated using Equation 1 if x is even, and it is calculated using Equation 2 if x is odd. An visual example of this encoding process is shown in Figure 3.

$$ve_{y,x} = sin\left(\frac{y}{10000^{x/d}}\right) \tag{1}$$

$$ve_{y,x} = cos\left(\frac{y}{10000^{(x-1)/d}}\right) \tag{2}$$

GNN Model

The GNN model used in this project consists of three Graph Convolutional Network (GCN) layers (Zhang et al. 2022b), and Rectified Linear Unit (ReLU) activation functions were applied after each layer. Experiments were conducted to evaluate the efficacy of other combinations of 1 to 5 GCN layers with different output channel sizes, however no notable improvement in performance was observed with the other options. The output channel numbers of these convolutional layers are $64 \rightarrow 64 \rightarrow f$, where f is the number of dimensions accepted by the anomaly detection model. Empirically, the optimal value for f in this context was found to be 3.

At the end, a single representation of the system call transition graph is made by averaging the node features across the node dimension. That is, for every element d_i in the graph's d dimensional feature vector, the sum of each nodes' i^{th} feature are divided by the total number of nodes. This pooling technique was found to outperform other common approaches, such as just computing the sum of the node features. These graph embeddings are what is passed on to the anomaly detection model.

Anomaly Detection Model

One-class Support Vector Machines (SVM) are a class of machine learning models that are used for anomaly detection (Hu, Hu, and Du 2019). Given a set of training data, the one-class SVM is designed to identify patterns within the set, which it then uses to determine whether or not a new test sample is an outlier. More specifically, the one-class SVM uses a kernel function to determine an hypersphere decision boundary. In essence, its goal is to identify the minimum radius r such that all the training data points lie within or on the hypersphere's boundary. This decision boundary then serves as a representation of the normal data distribution, which can be used to classify any data points that fall outside of the boundary as anomalous (see Figure 4).

For the purposes of this project, the one-class SVM model is trained on a set of graph embeddings generated from normal executions. Upon determining the optimal hypersphere, the model is then given new graph embeddings and tasked with labeling them as normal, or as exhibiting unseen behaviour. This means that the definition of a normal

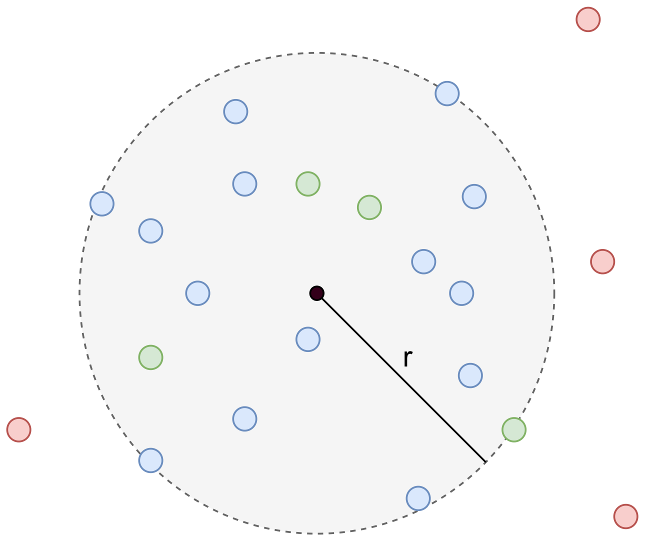

Figure 4: A one-class SVM's decision boundary (grey) determined using the training set points (blue), which is then used to differentiate between normal (green) and anomalous (red) points.

or anomalous execution is entirely dependent on the training data. If the training data is not properly filtered, then harmful anomalies may go undetected. Alternatively, if the training data is not comprehensive, benign behaviour may be flagged as an anomaly.

Results

To evaluate the proposed anomaly detection approach, it was implemented using Python 3.9.16, Torch 1.13.0, and PyTorch Geometric 2.3.0. All tests were conducted in a Google Research's Colaboratory Pro runtime environment with over 12 GBs of system RAM. All of the test input sequences, source code, and results have been made publicly available on GitHub[1].

Data

The data used to evaluate the proposed anomaly detection approach was taken from the Australian Defence Force Academy - Linux Dataset (ADFA-LD) (Creech and Hu 2013). Despite being generated in 2013, this dataset continues to be a widely-used benchmark in modern system call analysis research (Zhang et al. 2020; Lu and Teng 2021; Soliman, Sobh, and Bahaa-Eldin 2021; Shin and Kim 2020). The dataset is comprised of several system call sequences recorded on Ubuntu 11.04, each representing the execution of an application. Each system call is assigned a number, and as such the sequences are represented as a stream of integers. This means that any information pertaining to the events' duration, arguments, return values, and so forth is unavailable.

The system call sequences that comprise the ADFA-LD dataset are divided into two main categories. Firstly, there are 5,206 sequences from normal applications running on the system. Additionally, there are 746 sequences exhibiting anomalous behavior caused by malware attacks. The anomalous sequences cover six types of real-world attacks, namely Adduser, Hydra-FTP, Hydra-SSH, Java-Meterpreter, Meterpreter, and Webshell. A brief description of each attack type is provided in Table 1. To create a balanced test dataset, 746 normal sequences were set aside to evaluate the model, and the remaining 4,460 sequences were used for training the one-class SVM.

Transition Graphs

Visualizations of both normal and anomalous system call transition graphs made from the ADFA-LD system call sequences are shown in Figure 5. Manual inspection of the normal and anomalous graphs shows that they exhibit differences in the types of connected nodes as well as in overall structure. However, these differences are complex and nuanced, illustrating the challenges of manual analysis and reinforcing the need for automated anomaly detection techniques.

Anomaly Detection Accuracy

The primary method used to evaluate the anomaly detection model's efficacy was to calculate its accuracy (Equation 3), precision (Equation 4), and recall (Equation 5). Each of these metrics are computed using on the number of correctly identified anomalies (TP), incorrectly identified anomalies (FN), correctly identified normal sequences (TN), and incorrectly identified normal sequences (FP).

$$Accuracy = \frac{(TP + TN)}{(TP + FP + TN + FN)} \tag{3}$$

$$Precision = \frac{TP}{(TP + FP)} \tag{4}$$

$$Recall = \frac{TP}{(TP + FN)} \tag{5}$$

As it stands, the system call anomaly detection approach can not effectively differentiate between normal and anomalous system call sequences. Overall, the model achieved a 63.472% accuracy, a 61.254% precision, and 73.324% recall. Considering that this is frames as a binary classification task, these results are only moderately better than what could be achieved with random guessing. Furthermore, the results are significantly lower than other comparable system call anomaly detection techniques. The model's accuracy, as well as the accuracy reported in other comparable works, is shown in Table 2.

Attack	Description
Adduser	Adding new superuser using a malicious client-side executable
Hydra-FTP	Bruteforce password guessing on FTP port
Hydra-SSH	Bruteforce password guessing on SSH port
Java-Meterpreter	Java-based exploit to access interactive shell
Meterpreter	Linux-based exploit to access interactive shell
Webshell	PHP-based remote file inclusion vulnerability

Table 1: Real-world attacks that were recorded and included in the ADFA-LD dataset.

Model	Accuracy (/%)
Project	63.472
Early Detection Model (Zhang et al. 2020)	97.05
SVM (Shin and Kim 2020)	82.6869
Logistic Regression (Shin and Kim 2020)	78.9025
KNN (Shin and Kim 2020)	84.7682

Table 2: Accuracy of the model compared to that of other works that us the ADFA-LD dataset.

[1]https://github.com/janecekm/GNN-Anomaly-Detection

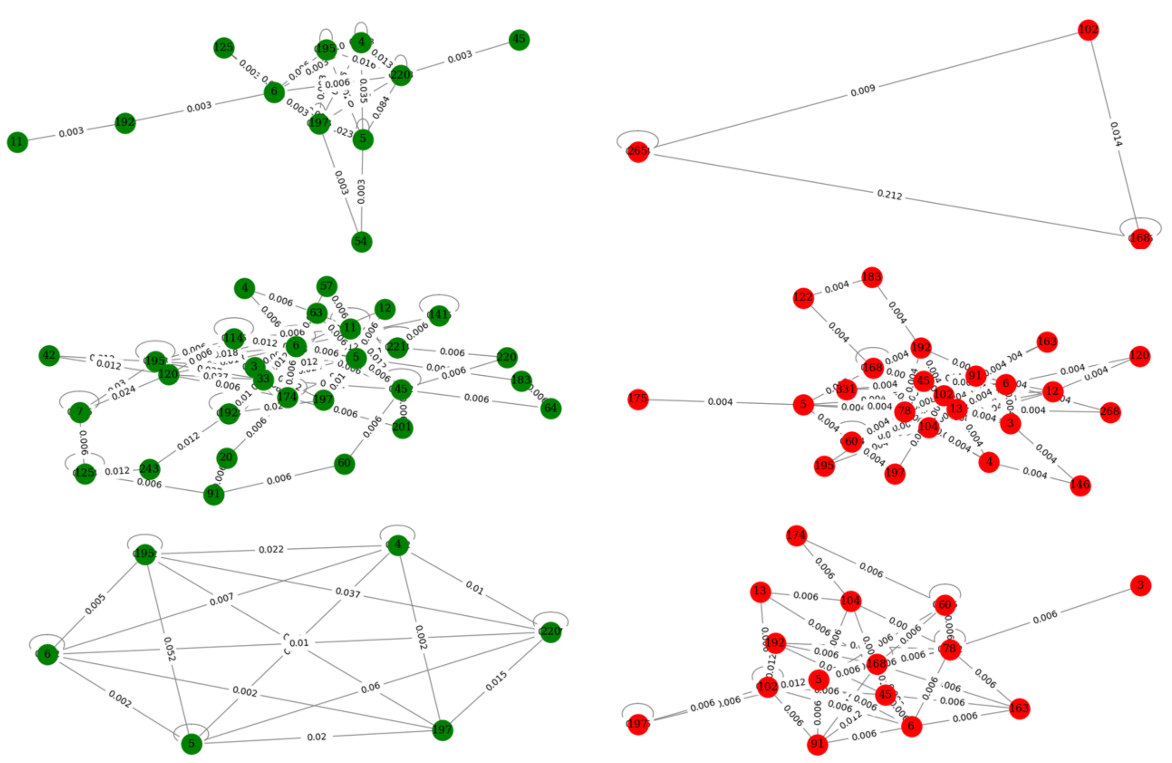

Figure 5: Random samples of both normal (green) and anomalous (red) system call transition graphs.

Runtime Performance

The average execution runtime for each task in the proposed methodology is shown in Table 3. The majority of the computational time is spent on processing the training data and training the one-class SVM. However, since the model can be reused multiple times, the training time is only required when setting up or updating the model. The time it takes to process and classify a new execution, which is the actual time sensitive portion of the proposed method, takes only a fraction of a second.

These results demonstrate that, when only considering efficiency, the proposed method is capable of performing anomaly detection in real time without high computational overhead. This is especially important for practical applications of anomaly detection systems, where the results need to allow for a timely response to potentially critical anomalies.

Conclusion

In conclusion, the current methodology is not sufficiently accurate to perform system call anomaly detection. As it stands, the model's 61.254% accuracy would lead to too many false alarms and unnoticed anomalies. As such, alternative methods need to be explored to improve the existing framework's efficacy.

Task	Average Time (s)
Graph Generation	0.003 per graph
Graph Embedding	0.003 per graph
One-class SVM Training	0.959 per model
Anomaly Classification	0.0005 per graph

Table 3: The average execution time for each task in the proposed methodology.

Extensions

There are several opportunities for enhancing the proposed method in future work. For instance, the current data preprocessing technique does not appear to generate graph embeddings that are easily discernible by machine learning classification models. As such, further research will include testing if the approach's accuracy would be improved through the use of graph autoencoders (GAEs) (Lin et al. 2023). Furthermore, this project has focused solely on system calls sequences, however it is worth noting that there is more data that can be gleaned from trace data. For instance, the duration of system calls can be integral in identifying time-out bugs (He, Dai, and Gu 2018). Or in (Fournier et al. 2021), the authors show how exploring system call arguments can lead to improved anomaly detection. Future work may include finding a way to incorporate additional into the graph representations, which could improve the method's ability to find performance anomalies.

This work may also be extended by applying it to a specific use case. For instance, early anomaly detection is when only a fraction of the sequence is available for anomaly detection, allowing for real-time responses to anomalous execution behaviour (Zhang et al. 2020). Based on its promising efficiency, this method may be well-suited for early anomaly detection.

References

Cavalcanti, M.; Inacio, P.; and Freire, M. 2021. Performance evaluation of container-level anomaly-based intrusion detection systems for multi-tenant applications using machine learning algorithms. In *Proceedings of the 16th International Conference on Availability, Reliability and Security*, ARES 21. New York, NY, USA: Association for Computing Machinery.

Cornelissen, B.; Zaidman, A.; Deursen, A.; Moonen, L.; and Koschke, R. 2009. A systematic survey of program comprehension through dynamic analysis. *Software Engineering, IEEE Transactions on* 35:684 – 702.

Creech, G., and Hu, J. 2013. Generation of a new ids test dataset: Time to retire the kdd collection. In *2013 IEEE Wireless Communications and Networking Conference (WCNC)*, 4487–4492.

Deng, A., and Hooi, B. 2021. Graph neural network-based anomaly detection in multivariate time series. In *Proceedings of the AAAI conference on artificial intelligence*, volume 35, 4027–4035.

Desnoyers, M., and Dagenais, M. 2008. LTTng: Tracing across execution layers, from the hypervisor to user-space. In *Linux Symposium*, 101.

Ezzati-Jivan, N.; Fournier, Q.; Dagenais, M. R.; and Hamou-Lhadj, A. 2020. DepGraph: Localizing performance bottlenecks in multi-core applications using waiting dependency graphs and software tracing. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 149–159.

Forrest, S.; Hofmeyr, S.; Somayaji, A.; and Longstaff, T. 1996. A sense of self for unix processes. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, 120–128.

Fournier, Q.; Aloise, D.; Azhari, S. V.; and Tetreault, F. 2021. On improving deep learning trace analysis with system call arguments. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 120–130.

Grimmer, M.; Röhling, M.; Kricke, M.; Franczyk, B.; and Rahm, E. 2018. Intrusion detection on system call graphs.

He, J.; Dai, T.; and Gu, X. 2018. Tscope: Automatic timeout bug identification for server systems. In *2018 IEEE International Conference on Autonomic Computing (ICAC)*, 1–10.

Hu, Z.; Hu, Z.; and Du, X. 2019. One-class support vector machines with a bias constraint and its application in system reliability prediction. *AI EDAM* 33(3):346–358.

Khraisat, A.; Gondal, I.; Vamplew, P.; and Kamruzzaman, J. 2019. Survey of intrusion detection systems: techniques, datasets and challenges. *Cybersecurity* 2.

Kohyarnejadfard, I.; Aloise, D.; Dagenais, M. R.; and Shakeri, M. 2021. A framework for detecting system performance anomalies using tracing data analysis. *Entropy* 23(8).

Lin, M.; Wen, K.; Zhu, X.; Zhao, H.; and Sun, X. 2023. Graph autoencoder with preserving node attribute similarity. *Entropy* 25:567.

Lu, Y., and Teng, S. 2021. Application of sequence embedding in host-based intrusion detection system. In *2021 IEEE 24th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, 434–439.

Shin, Y., and Kim, K. 2020. Comparison of anomaly detection accuracy of host-based intrusion detection systems based on different machine learning algorithms. *International Journal of Advanced Computer Science and Applications* 11.

Soliman, K.; Sobh, M. A.; and Bahaa-Eldin, A. M. 2021. Survey of machine learning hids techniques. In *2021 16th International Conference on Computer Engineering and Systems (ICCES)*, 1–5.

Surendran, R., and Thomas Kallivayalil, T. 2022. Detection of malware applications from centrality measures of syscall graph. *Concurrency and Computation: Practice and Experience* 34.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.; Kaiser, L.; and Polosukhin, I. 2017. Attention is all you need.

Xie, Y.; Zhang, H.; and Babar, M. A. 2022. Loggd: Detecting anomalies from system logs by graph neural networks. *arXiv preprint arXiv:2209.07869*.

Zhang, X.; Niyaz, Q.; Jahan, F.; and Sun, W. 2020. Early detection of host-based intrusions in linux environment. In *2020 IEEE International Conference on Electro Information Technology (EIT)*, 475–479.

Zhang, C.; Peng, X.; Sha, C.; Zhang, K.; Fu, Z.; Wu, X.; Lin, Q.; and Zhang, D. 2022a. Deeptralog: Trace-log combined microservice anomaly detection through graph-based deep learning. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 623–634.

Zhang, H.; Lu, G.; Zhan, M.; and Zhang, B. 2022b. Semi-supervised classification of graph convolutional networks with laplacian rank constraints. *Neural Processing Letters* 54:1–12.

Zhang, K.; Zhang, C.; Peng, X.; and Sha, C. 2022c. Putracead: Trace anomaly detection with partial labels based on gnn and pu learning. In *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*, 239–250.