Anomaly Detection in Sequential Data: A Deep Learning-Based Approach

Jayesh Soni
Florida International University, jsoni002@fiu.edu

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

ANOMALY DETECTION IN SEQUENTIAL DATA:

A DEEP LEARNING-BASED APPROACH

A dissertation submitted in partial fulfillment of

the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Jayesh Soni

2022

To: Dean John L. Volakis
College of Engineering and Computing

This dissertation, written by Jayesh Soni, and entitled Anomaly Detection in Sequential Data: A Deep Learning-Based Approach, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

Hadi Amini

Leonardo Bobadilla

Ajeet Kaushik

Ananda Mondal

Himanshu Upadhyay

Nagarajan Prabakar, Major Professor

Date of Defense: June 20th, 2022

The dissertation of Jayesh Soni is approved.

Dean John L. Volakis
College of Engineering and Computing

Andres G. Gil
Vice President for Research and Economic Development and
Dean of the University Graduate School

Florida International University, 2022

DEDICATION

To my family members for their supports and sacrifices

ACKNOWLEDGMENTS

First and foremost, I owe my deepest gratitude to my supervisor Professor Nagarajan Prabakar for his generous supervision, guidance, and encouragement during my Ph.D. He was always patient enough to discuss anything with enthusiasm and guided me in the right direction. I am also deeply grateful to Professor Himanshu Upadhyay for supporting, mentoring, and helping in all the stages. I would also like to thank Professor Leonardo Bobadilla, Professor Hadi Amini, Professor Ananda Mondal, and Professor Ajeet Kaushik for being on my doctoral committee. They have provided valuable suggestions. This work is supported by DoD-TRMC. Many thanks to TRMC for providing me the financial support. I want to thank Mr. Clint Miller for always supporting me in solving the issues with the computational servers in the lab. I would also like to thank my lab-mates and all of my co-authors with whom I have worked together. It was my great pleasure to work with them. I am also thankful for the administrative and advising staff. I am also thankful to the conference organizers, publishing editors, and reviewers for providing me the constructive and valuable feedback and facilitating the publication of my research. Finally, I am grateful to my parents. Without their inspiration, it would not be possible to finish my study.

ABSTRACT OF THE DISSERTATION

ANOMALY DETECTION IN SEQUENTIAL DATA: A DEEP LEARNING-BASED APPROACH

by

Jayesh Soni

Florida International University, 2022

Miami, Florida

Professor Nagarajan Prabakar, Major Professor

Anomaly Detection has been researched in various domains with several applications in intrusion detection, fraud detection, system health management, and bio-informatics. Conventional anomaly detection methods analyze each data instance independently (univariate or multivariate) and ignore the sequential characteristics of the data. Often, anomalies in the sequential data can be detected when the individual data instances are analyzed by grouping them into a sequence and hence cannot be detected by a conventional way of anomaly detection. Currently: (1) Deep learning-based algorithms are widely used for anomaly detection purposes. However, significant computational overhead time is incurred during the training process due to static constant batch size and learning rate parameters for each epoch, (2) the threshold to decide whether an event is normal or malicious is often set as static. This can drastically increase the false alarm rate if the threshold is set low or decrease the True Alarm rate if it is set to a remarkably high value, (3) Real-life data is messy. It is impossible to learn the data features by training just one algorithm. Therefore, several one-class-based algorithms need to be trained. The final output is the ensemble of the output from all the algorithms. The prediction accuracy can be increased by giving a proper weight to each algorithm's output. By extending the state-of-the-art techniques in learning-based algorithms, this dissertation provides the following solutions: (i) To address (1), we propose a hybrid, dynamic batch size and learning rate tuning algorithm that reduces the overall training time of the neural network. (ii) As a solution for (2), we present an adaptive thresholding algorithm that reduces high false alarm rates. (iii) To overcome (3), we propose a multilevel hybrid ensemble anomaly detection framework that increases the anomaly detection rate of the high dimensional dataset.

TABLE OF CONTENTS

CHAPTER PAGE

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Security breaches due to ransomware, virus, Trojans, etc., have been reportedly increasing in recent years [1]. A security vulnerability has been reported on a continuous basis which can be seen as a failure by the cybersecurity analysts towards mitigating such attacks. Malware also known as malicious code can be described as "a code written and executed intentionally to harm the system by either adding, modifying or deleting some of its parameters" [2]. A malware can either be a standalone program or attached to the known program.

There are numerous reasons for the creation of malware. Some malware is developed as a concept to avoid future vulnerability. Such types of malware do not cause any harmful attacks on the systems. Some other types of malware created by cyber attackers are solely for stealing private information, infecting the system's main code, etc. There are many sensitive data stored in various current systems and numerous quantities. These give high opportunities to cyber attackers to gain profit illegally out of such legitimate systems. Since the early 2000s, the increase in malware has been exponential. With the migration of traditional office-based work to remote work, high-level targeted attacks have been performed against companies' critical infrastructure.

To identify such malware, anti-virus software provides solutions in two main methods: *signature-based*, which utilizes the already known malware database to detect the new malware, and *anomaly-based*, which makes use of the normal patterns behavior of the program to discriminate between malicious and legitimate program [3]. Signature-based methods can detect only the known malware, whereas anomaly-based methods can detect any malware. Signature-based methods need a labeled dataset of benign and malware code and thus are inefficient in identifying new malware in current world scenarios since most datasets are highly imbalanced. Anomaly-based methods need only a benign class of dataset, and anything that deviates from the normal class is considered anomalous. Thus anomaly-based detection methods are highly efficient in identifying new unknown malware.

Machine Learning and Deep Learning-based anomaly detection algorithms have shown promising results [4, 5, 6, 7]. These learning-based methods have high capabilities in learning the feature representations of complex data. Graph-based data, Spatial-temporal data, and high dimensional data are some examples of complex data. Deep learning for anomaly detection utilizes neural networks to learn the explicit feature representations for detecting the anomaly. In a wide variety of applications, such learning-based anomaly detection methods have also outperformed the traditional anomaly detection methods.

There are numerous ways to develop models that can capture the behavior of the process. One possible approach is to effectively utilize the sequence of system call sequences [8]. The important observation is the underlying fact that for a malicious code to cause harmful damage to the system, it has to interact with the operating system through system calls. If a particular sequence of system calls deviates from the normal expected behavior, we can assume an attack has happened. Thus it is essential to capture every single system call made by a process to its operating system during its execution for analyzing the whole traces of the system call. The anomaly detection methods have to take these sequences of system calls to learn the normal behavior of the process and should be able to detect the abnormality in case the process is injected with a malicious code.

A particular malicious code has to specify the target process to infect it with the malware. Thus, to further enhance the anomaly detection rate, the executive process (_EPROCESS) data structure can be considered. _EPROCESS is a kernel memory structure that contains various distinct attributes pertaining to the process. Every individual operating system process is represented by _EPROCESS. Each _EPROCESS structure has a *Process Environment Block* (PEB). The *Dynamic Link Library* (DLLs) loaded by the process is stored using three doubly-linked lists by PEB. *MemoryOrderList, LoadOrderList and InitOrderList* are such three linked lists. Each list holds the DLLs for a particular process differently. The *MemoryOrderList* uses the virtual memory address space of the loaded DLL. The *LoadOrderList* stores the DLLs in the order they were loaded in the process. The *InitOrderList* uses the order of the execution of the main function of each DLL.

Thus ensemble analysis of the sequence of system calls and different process attributes is the key to efficient anomaly detection.

1.1 Challenges

Although the malware attacks are supposed to be identified by the underlying anti-malware tools and software as part of their services, there are some challenges as follows:

(1) There are various learning-based models trained to detect whether there is an anomaly in the process or not. However, there is a lack of research for contextual point anomaly detection in the current literature.

The learning-based models [9, 10, 11] employed in the existing anomaly detection approaches predict the whole sequence of system calls as benign or malicious. However, to identify the real purpose of the calling actions behavior, an appropriate infected subsequence of system calls needs to be identified. As an existing example, *Sequence Time-Delay Embedding* (STIDE) [12], an extension of Time-Delay Embedding (TIDE) [8], uses the three-tier system to detect the anomalous events. The first two-stage of the three-tier approach utilizes the original input trace and applies the sliding windows of length *k* to generate substrings of fixed length as features, followed by database construction of the features for the training purpose. The third stage uses the static threshold count on the number of mismatches to decide whether a test sequence is anomalous or not. However, such methods lack generalization capabilities and need large storage capacities. Furthermore, training such models is a resource-intensive task that grows linearly with the length of the number of training sequences.

(2) Currently, deep learning algorithms are trained for anomaly detection. However, significant computational overhead time is incurred during the training process due to static constant batch size and learning rate parameters for each epoch.

Most of the real-world analyses in the spatial and temporal context, such as text analysis [13], intrusion detection [14], click fraud detection [15], and sensor anomaly detection [16], are big data and are highly imbalanced. Training the neural network such as *Long Short Term Memory* (LSTM) for a dataset using traditional methods requires enormous computational resources and is time-consuming. Furthermore, many hyper-parameters need to be tuned. One of the hyper-parameter related to memory issues is the batch size. The number of data rows is used to calculate the gradient and further update the neural network's weights in each epoch. Currently, the batch size value is kept constant throughout the training period. However, changing the batch size adaptively can reduce the training time and further improve the utilization of the memory resources efficiently. The second important hyper-parameter is the learning rate, allowing the model to converge to global optima. Currently, adapting these two hyper-parameters is heavily researched in the computer vision area where *Convolution Neural Network* (CNN) is employed [17]. Several techniques are available [18, 19, 20, 21] to adaptively tune those two parameters. AdaBatch [22] is one of the most popular adaptive batch size techniques. Several variants of AdaBatch have been proposed in the recent past to improve the training time and maintain accuracy. Such approaches are applied to the image dataset [23, 24, 25]. However, very few studies have been performed in the recent past on the time series applications which employ the LSTM algorithm. One such study is *the Dynamic Adaptive-Tuning Engine* (DATE) [26]. However, they use constraints such as *Sqrt Scaling* and *Linear Scaling*, which are helpful but do not always prove highly efficient when using specific metrics and optimizers in the anomaly detection area since different optimizers require different initialization of the hyperparameters.

(3) *One class-based algorithm has shown high accuracy in detecting the anomaly. However, the threshold to decide whether an event is normal or malicious is often set as static. This can drastically increase the false alarm rate if the threshold is set low or decrease the True Alarm rate if it is set to a remarkably high value.*

Most of the existing anomaly detection work focuses on prediction than detection. One of the most common issues of fixed thresholds is a low detection rate or high false alarm rate. Setting the threshold to an optimal value in the detection method is an active research area. Parametric and Non-Parametric techniques are being utilized to solve this problem. Parametric techniques such as calculating the Gaussian distribution by employing maximum likelihood estimation [27], double window scoring method [28], and p-value scoring [29] work on the assumption that the distribution of the data is known. This is the limitation of such parametric-based techniques. Non-parametric techniques such as distance-based [30] and residual evaluation do not need to know the data distribution. Dynamic Thresholding can be improved by performing some modifications (Anomaly Pruning [31]) or by adding some extensions (USAD [32]).

(4) Real-life data is messy. It is impossible to learn the data features by training just one algorithm. Therefore, several one-class-based algorithms need to be trained. The final output is the ensemble of the output from all the algorithms. The prediction accuracy can be increased by giving a proper weight to each algorithm's output.

Ensemble approaches are heavily studied in supervised learning, where we have training data with the target label. Ensembles for anomaly detection (unsupervised learning) are an emerging topic. *Normalization* and *Combination* are two of the issues common in the ensembles of unsupervised algorithms. Feature bagging [33] ranks the anomaly detection algorithms from the highest anomaly detection rate to the lowest detection rate. The major disadvantage of this approach is that it loses information about the relative difference between the outlier scores. This can be enhanced by considering the rank and the probability distribution of the value. The choice of function needed to combine the output score is also equally important. *Maximum Function, Averaging Function, Damped Averaging,* and *Pruned averaging* are used. Currently, these combination function is dependent on the individual algorithm. This can be extended by developing an algorithm that first uses the normalization of the scores and then applies the combination approach to produce an optimal anomaly score.

1.2 The objective of this Dissertation

To address the above-said challenges, in this dissertation, we propose Machine Learning and Deep Learning-based solutions to improve the performance of the existing anomaly detection methods. The following serve as the objectives of this dissertation:

- Proposing a hybrid adaptive Batch Size with a Learning Rate tuning algorithm for training the neural network in an optimized way.
- To improve the existing state-of-the-art thresholding methods and reduce the false alarm rate.
- Application of weighted ensemble approach as a hybrid solution for efficient anomaly detection.

1.3 Contributions

This dissertation presents solutions to the anomaly detection problem by extending the state-of-the-art techniques in unsupervised learning methods.

1.3.1 SysCallNet: Behavioral Analysis of System Calls using Deep Learning-based Algorithms for Anomaly Detection

To partly address the challenge (1), we propose a two-dimensional framework to analyze the behavior of the system calls for anomaly detection. We analyzed two types of behavior: *Temporal* and *Non-Temporal* behavior. We trained sequential deep learning-based algorithms for temporal behavior, namely Long Short Term Memory (LSTM). For training the model, only a normal class of data was supplied. Next, the non-temporal behavior is analyzed independently by evaluating frequency and commonality behavior. We applied Cosine Similarity for analyzing frequency behavior and Jaccard Similarity for analyzing the commonality behavior.

1.3.2 P-BoSC: An Extension of Bag of System Call Technique for Detection of Anomalous Points

To fully address the challenge (1), prior works show that detecting the particular window of system calls has not been commonly used in combination with *Bag of System Calls* (BoSC). Hence, we propose an extension called Point-Bag of System Calls (P-BoSC), a Natural Language Processing-based technique to detect the anomaly and output the anomalous window. It is a hybrid method containing two steps. In the first step, we train the cosine similarity algorithm to learn the normal behavior of the data. Next, in the second step, if the output is anomalous while testing, the sequence is given input to *PBoSC*. *PBoSC* algorithm analyzes the frequency behavior and detects an anomalous sequence window.

1.3.3 DynaB: An Enhanced and Dynamic Batch Size Tuning for LSTM Neural Network

Concerning challenge (2), we present a *dynamic tuning algorithm* that can change the batch size dynamically. The proposed algorithm consists of four stages: Gradient Warmup, Loss derivation, calculating the weighted loss with the historical batch size, and updating the batch size. The proposed work's objectives are, firstly, to model the time series sequence data on LSTM Network by relying only on the system call sequences, without the need for too many attributes. Secondly, warm up the gradient for the first two batches to derive the loss using the optimizers. Then, we evaluate the loss in terms of the number of the sequence of system calls predicted correctly. We evaluate *Root Mean Square Error (RMSE)*, *Mean Absolute Error (MAE), and Mean Absolute Percentage Error (MAPE)* as a wide variety of loss functions. Next, using the historical batch size value with their incorporated loss, we dynamically decide on the new batch size rather than keeping the batch size constant. We go one depth further to achieve more granularity for calculating the loss gradient where instead of at each epoch level, it is calculated at each batch level. We achieve three significant benefits: *Reduced training time*, *Optimized memory usage*, and *Loss reduction* at an early epoch stage.

1.3.4 DynaB-LR: A Hybrid Algorithm for Dynamic Batch Size and Learning Rate tuning for an optimized training of Neural Network

As a solution to challenge (2), we propose a hybrid algorithm, *Dynamic Batch Size and Learning Rate (DynaB-LR)*, that tunes the batch size and learning rate dynamically and consecutively. It is fast and can be applied to any time series-based dataset. At the same time, it accounts for the learning algorithm obtaining the optimal value of these hyper-parameters without the need for the user to manually input the number, unlike most of the other established algorithms such as [34]. The algorithm works in four steps. First, we simultaneously perform a gradual warmup for batch size and learning rate for the first two epochs. Then, during the warmup procedure, the loss gradient is calculated using the optimizer. In the third epoch, we dynamically change the batch size based on the loss calculation. In the next epoch, the learning rate is updated. We continue to update batch size and learning rate in alternate epochs till the model is trained successfully. Thus, this approach of dynamically tuning batch size and the learning rate results in optimizing the training time of the neural network.

1.3.5 Machine Learning-based Cyber Threat Anomaly Detection in Virtualized Application Processes

To address the challenge (4), we developed a two-step hybrid framework. Host-based systems frequently depend on various attributes of a process to describe the normal behavior of any process. Multiple malicious vectors can be launched on a process with different characteristics to infect it. First, we analyze *ProcessList* data structure and create Principal Component Analysis (PCA) features known as Eigen traces used for training multiple one-class anomaly detection models. These multiple models allow different attributes of process data to be assessed from numerous and diverse standpoints. As the anomaly scores of these models vary significantly, combining the scores to a single value is often challenging. Therefore, we apply a majority voting approach for the final anomaly score as the second step. This final score measures the occurrence of a malicious event. We demonstrate the implementation of the proposed two-step approach using four different one-class classifiers: *Mahalanobis Classifier*, *One-Class Support Vector Machine (OCSVM)*, *Isolation Forest*, and *Dendrogram based Agglomerative Clustering*.

1.3.6 AdaThres: An Adaptive Thresholding Method to Mitigate the False Alarms

To address the challenge (3), we develop an adaptive thresholding algorithm that can mitigate the issue of high FPR. The proposed algorithm applies three scoring mechanisms. They are *Anomaly Pruning*, *Sequence Scoring*, and *Adaptive Thresholding*. The model is trained on sequential data. *Anomaly Pruning* gives a score to an individual data point. It either rejects or accepts the data points to be considered for Sequence Scoring. This *Sequence Scoring* will give a score to an individual sequence. Finally, an *Adaptive Thresholding* is applied to the cumulative score of all the sequences to detect the anomalous nature of the analyzed data. Multiple experiments have been conducted using various optimizers to access our proposed approach. Using the proposed approach, we train a deep learning-based LSTM algorithm widely adopted for sequential data. Furthermore, we validate it with three different datasets of various sizes.

1.3.7 EA-Net: A Hybrid and Ensemble Multi-Level Approach for Robust Anomaly Detection

As a solution to challenge (4), we design a multi-level hybrid ensemble anomaly detection approach. At the First Level, we train several weak classifiers (weak one class classifiers). Next, we utilize deep learning-based AutoEncoder to reduce the dimension of the dataset. These are the two sets of hybrid features. Next, different one-class classifiers have their strength and limitations. Thus, we propose an adaptive weightage approach that gives the weight to each classifier. Next, this input is passed to the second level. At this level, we have a deep neural network that learns the patterns of the dataset and generates an adaptive dynamic threshold to discriminate the input feature as an anomaly or benign. The major benefit of this approach is the *reduced training time* and *high anomaly detection rate*.

CHAPTER 2

SYSCALL-NET: BEHAVIORAL ANALYSIS OF SYSTEM CALLS USING DEEP LEARNING-BASED ALGORITHMS FOR ANOMALY DETECTION

With the advent of technology, sophisticated malware presents a significant threat to computer security. This work proposes anomaly detection techniques that learn three different behaviors of windows system-call sequences. We apply *Long-Short-Term-Memory (LSTM)* for temporal behavior, *Cosine Similarity* for frequency distribution behavior, and *Jaccard Similarity* for commonality behavior. The proposed framework monitors the processes in a hypervisor-based environment to detect compromised virtual machines. System call sequences of normal and malware-infected processes were extracted with memory forensic techniques. Our proposed anomaly detection techniques learned the above three behavior of the system call sequences with 99% accuracy.

2.1 Introduction

Virtualization is popular nowadays in distributed systems due to its usage and applicability. Vast resource sharing and load balancing across multiple nodes are the significant advantages of virtualization. With the invention of virtualization technologies, hypervisor-based methods have evolved to scan virtual machines (VMs) and identify threats. Every day, new malware is becoming more sophisticated and robust such that traditional malware detection techniques are incapable of detecting it and thus fail in protecting the VMs.

© 2019. Reprinted, with permission, from J Soni, et al., Behavioral Analysis of System Call Sequences using LSTM Seq-Seq, Cosine Similarity and Jaccard Similarity for Real-time Anomaly Detection. In Proceedings of the 6th IEEE International Conference on Computational Science and Computational Intelligence (CSCI), Las Vegas, Nevada, pp. 214-219, Dec 5-7, 2019. DOI: https://doi.org/10.1109/CSCI49370.2019.00043

This complexity of malware becomes a cyber threat to organizations. To overcome this problem, hypervisor-based malware detection techniques have evolved and are outperforming compared to guest-based systems. *Virtual Machine Introspection (VMI)* is the most effective host-based malware detection technique to monitor and analyze cyber threats on virtual machines [35, 36, 37].

Anti-viruses and security patches are a few existing security techniques available to prevent malicious attacks. Regardless of these techniques, there is still a possibility of unknown attacks on the VM due to the delay in installing the latest updates of the anti-virus software and security patches. Potential malware attacks are identified by studying the characteristics of the program execution, which is known as behavioral analysis [38]. System call sequence analysis [8, 39] leads to several behavioral analyses for malware detection.

2.1.1 Summary of Contribution

We propose a two-dimensional framework to analyze the behavior of the system calls for anomaly detection. We analyzed two types of behavior: Temporal and Non-Temporal behavior. For temporal behavior, we trained sequential deep learning-based algorithms, namely Long Short Term Memory (LSTM). For training the model, only a normal class of data was supplied. Next, the non-temporal behavior is analyzed independently by evaluating frequency and commonality behavior. We applied Cosine Similarity for analyzing frequency behavior and Jaccard Similarity for analyzing the commonality behavior.

2.1.2 Organization of the Chapter

Section 2.2 discusses the state-of-the-art techniques related to the current approach. In section 2.3, we present an overview of the system call extraction. The feature extraction technique for the system call sequence is discussed in section 2.4. We provide an in-depth understanding of the proposed anomaly detection algorithms in section 2.5. The experimental setup is described in section 2.6. Next, in section 2.7, the results are discussed. Finally, we present our conclusions in section 2.8.

2.2 Related Work

Host-based malware detection in any production system is crucial for the security of its internal hardware and software components. Such host-based frameworks use the knowledge of existing malware to detect malicious activities. There are two types of malware analysis techniques: *static* and *dynamic*.

The static method analyzes a source file without executing the code [40]. In paper [41], Ye et al. developed an Intelligent Malware Detection System (IMDS), which uses an association mining algorithm to obtain import function information. In paper [42], Masud et al. used the byte level code, containing five different static features extracted from assembly instructions. Alarifi and Wolthusen [43] take sequences from a virtual machine and then train the model using a Hidden Markov Model (HMM). Their HMM-based method gave fewer detection rates since it required fewer training samples. Wang et al. [44] use the probability score and threshold value.

However, techniques such as polymorphism, encryption, and advanced unknown malware are not traceable by static-based anomaly detection methods due to their dynamic characteristics. To overcome this limitation, we study the behavioral analysis of the process. The basic idea is to analyze the execution sequence of the process.

Neural networks are extensively used for anomaly detection [45, 46]. Recently deep learning-based techniques such as LSTM have been used for improving the anomaly detection rate [47, 48, 49, 50]. However, they used a feature-based supervised classifier that required pre-labeled malicious data, which inherently limits the detection of any unknown attacks [51, 52, 53]. Moreover, their approach needs specific feature engineering to generate meaningful feature representations for the supervised classification problem [54, 55, 56, 57].

Our work proposes three unsupervised anomaly detection approaches by learning different behaviors of system call sequences. Our approach utilizes a technique that trains on benign (normal) data and checks for unusual activities that differ from normal behavior. Such procedures identify unknown attacks on the system.

2.3 Overview of System Call Extraction Using VMI

The proposed framework consists of four platforms, namely *Virtualization, Advanced Cyber Analytics*, *Test Control Center*, and *Malware Repository*. The following subsections outline the details of the individual modules.

Virtualization Platform: This module uses the *VMI API* to introspect and perform memory analysis. The extracted low-level data from the hypervisor-based virtual machine memory is transferred to the agent listener. Introspection interface with hypervisors to add, delete, or control virtual machines. Security agent with LibVMI library performs introspection to extract data from the virtual machine and transfer it to the agent listener. Lastly, the data is directed to a database server.

Cyber Analytics Platform: In this module, we fetch data from the database server for data preprocessing and analysis. The module comprises of LSTM encoder-decoder algorithm to train models on the processed data.

Test control center Platform: The operator in this module can control different operations of the whole framework. The operator can create, delete, stop, or pause the VM. Further, the operator can manage VMs by running benign and malware applications.

Malware Repository Platform: This repository is the database for different multifunctional malware for Windows and Linux.

The security agent extracts system call traces of Processes Under Examination (PUE) from the proposed framework. Agent listener stores the collected information in the database through a socket established by introspection. System call sequence data is preprocessed using feature engineering techniques and analyzed with anomaly detection algorithms.

We generate malicious data sets of system call sequences through custom DLL injections. These custom malware hooks into the functions of the process, where the malware modifies system call sequences by adding, deleting, or modifying system calls. The detection algorithms need to identify this anomaly in system call sequences.

2.4 Feature Extraction Technique for System Call Sequence

System call sequences of processes under examination are collected from a hypervisor. We consider a sequence as a document, and each system call as a word.

Windows operating system has a total of *450* unique standard system calls. Each system call in the sequence is mapped to the corresponding standard system call index that ranges from *0 to 449*. A sample mapping is shown below.

SystemCallSequence= [NtQueryVolumeInformationFile, NtQueryVolumeInformationFile, NtQueryInformationFile, NtSetInformationFile, NtDelayExecution, NtDelayExecution, NtWriteFile, NtClose, NtCreateFile, ...]

The corresponding mapping is as below:

[73, 73, 17, 39, 52, 52, 8, 15, 85…]

Next, we create a bag of system calls as a vector of *450* dimensions. The value of the i^{th} cell in the vector implies the frequency of the i^{th} standard system call in the entire system call sequence.

2.5 Detection Algorithms Based on Behavioral Aspects

In this section, we discuss an overview of anomaly detection algorithms, namely *LSTM Seq-Seq*, *Cosine Similarity*, and *Jaccard Similarity*, as shown in Figure 2.1. These algorithms work well even for high-dimensional data and large training examples. They have low runtime computational complexity, crucial for anomaly detection systems. These algorithms work in an unsupervised learning mode without any explicit training labels during the training phase.

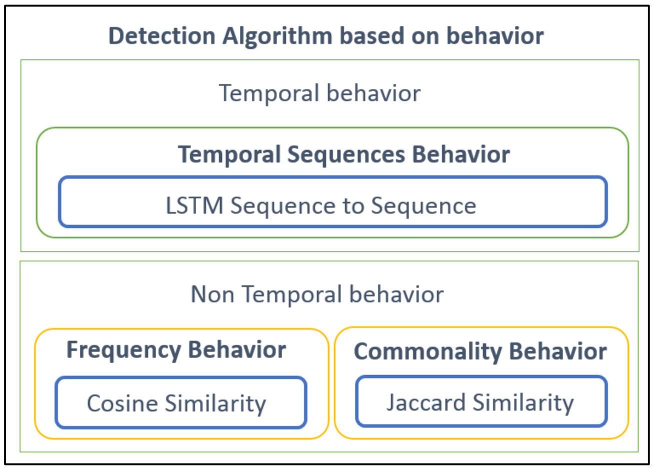
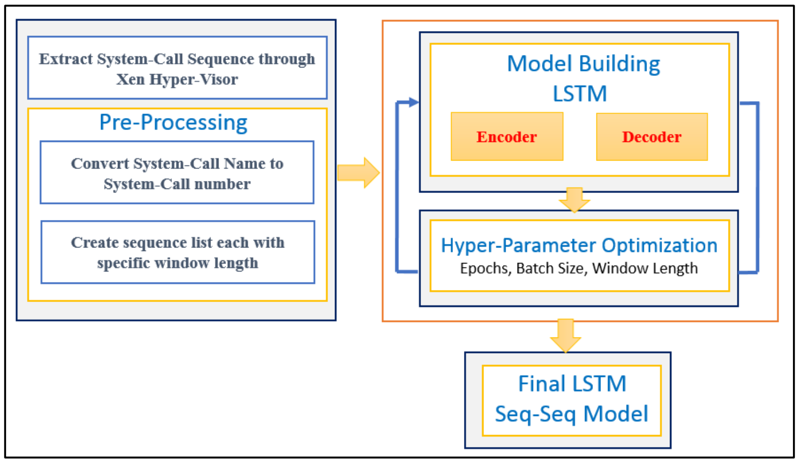


Figure 2.1: Behavior Detection Algorithm

2.5.1 Temporal aspects: Long Short Term Memory (LSTM) Seq-Seq

We train the *LSTM Seq-Seq model* on system calls by considering the system call sequences. Figure 2.2 depicts a high-level view of our LSTM training method. First, we preprocess the raw system call sequence and then train the LSTM model by optimizing its hyperparameters (Number of epochs, Batch size, and Sequence length) for higher accuracy and reduced runtime computational complexity. For the training of the LSTM model, we consider system call sequences as sentences where each system call in a sequence corresponds to a word in the sentence. We implemented sequence to sequence architecture with this approach, where we feed the first few system call sequences as input, and the trained LSTM model predicts the following system call sequence.



Extract System-Call Sequence through Xen Hyper-Visor
Pre-Processing
Convert System-Call Name to System-Call number
Create sequence list each with specific window length
Model Building LSTM
Encoder
Decoder
Hyper-Parameter Optimization
Epochs, Batch Size, Window Length
Final LSTM Seq-Seq Model

Figure 2.2: Overview of LSTM Model Building

In NLP, we require a vocabulary to convert a sentence into its numerical vector. Similarly, we have the vocabulary for the Windows system calls. Let us define S as the system call sequence generated by the hypervisor during program execution. We convert this sequence into a sequence of numerical values in the range *0 to n-1*, where *n* is the total number of unique system calls of the OS environment. The training set consists of *m* benign system call sequences represented as *m* training vectors. Since all system call sequences have the same number of system calls, all training vectors have a fixed number of numeric values (say *k* values). These numeric values correspond to the OS system call indices. For the training set of *m* system call sequences, we represent it as a set of *m* training vectors:

$$T_r = T_{r1}, T_{r2}, T_{r3}, \dots, T_{rm} \tag{2.1}$$

and represent the test system call sequence as T_e where T_e and each T_{ri} have k numeric values. The neural network of the LSTM encoder generates the hidden state and the output with the forward propagation operation as below:

$$h_t = (S^{HX} x_t + S^{HH} h_{t-1}) \tag{2.2}$$

$$y_t = (S^{YH} h_t) \tag{2.3}$$

The hidden state h_t is the encoded information, and vector *c* is known as the context vector, which is used in the decoder part. The equations for the weight update of an LSTM cell are as follows:

$$(g_t, b_{t-1}, s_{j-1}) \rightarrow (b_t, s_t) \tag{2.4}$$
$$y_t = \sigma\,(T_{xi}x_t + T_{hi}p_{t-1} + d_i) \tag{2.5}$$
$$f_t = \sigma\,(T_{xf}x_t + T_{hf}p_{t-1} + d_f) \tag{2.6}$$
$$o_t = \sigma\,(T_{xo}x_t + T_{ho}p_{t-1} + d_o) \tag{2.7}$$
$$g_t = \ tanh(T_{xc}x_t \ + \ T_{hc}p_{t-1} \ + \ d_c) \tag{2.8}$$
$$c_t = f_t c_{t-1} + \ i_t g_t) \tag{2.9}$$
$$k_t = o_t * tanh(c_t) \tag{2.10}$$

In the above equations, y_t, f_t, o_t, and c_t are the input, forget, output gates, and memory cell activation vectors, respectively, σ is a sigmoidal function, and *tanh* is the hyperbolic tangent function. The target sequence is generated by the decoder part of the architecture that uses the following conditional probability equation.

$$p(y_1, y_2, \ldots, y_T \mid x_1, x_2, \ldots, x_{i-1}) = \ \prod_{t=1}^{T'} \ p(\, y_T \mid y_1, y_2, \ldots, y_{t-1}) \tag{2.11}$$

The above conditional probability needs to be modified as below to include the attention mechanism.

$$p(\, y_i \mid y_1, y_2, \ldots, y_{i-1}, X) = g(y_{i-1}, s_i, c_i) \tag{2.12}$$

Where c_i is the context vector used by the attention mechanism during the training period.

$$c_i = \sum_{j=1}^{T_x} a_{ij} \,.\, h_j \tag{2.13}$$

Where a_{ij} is the coefficient of the i_{th} hidden state at time step j.

2.5.2 Frequency aspects: Cosine Similarity

Cosine similarity measures the cosine angle between two numerical vectors. The following Euclidean distance method is used:

$$A.B = \|A\| \, \|B\| \, cos\theta \tag{2.14}$$

Cosine similarity between two n-dimensional vectors *A* and *B* is calculated as *cos(θ)*:

$$\text{Similarity} = \cos(\theta) = \frac{A.B}{\|A\| \, \|B\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}} \tag{2.15}$$

where A_i and B_i are the features of the vectors.

We developed the following anomaly detection algorithm that uses the cosine similarity to detect anomalies at a particular time window in the process under examination.

Algorithm 1 Anomaly Detection Algorithm using Cosine Similarity

```
Input: Normal and Test System Call Sequence
Output: TestSeq, Similarity_Value
 1: for system call sequence of normal and malicious process do
 2:     Convert System Call Name to System Call Number using Mapping Table
 2:     $BOW_{BasePr}$ = Bag-of-words for normal process
 3:     $BOW_{TestPr}$= Bag-of-words for malicious process
 4:     $Base_{length}$ = length(Sequence list of Normal Process)
 5:     $Test_{length}$ = length(Sequence list of Test Process)
 6:     $Min_{length}$ = minimum ($Base_{length}$ , $Test_{length}$)
 7:     for $i$ = 0 to $window_{length}$ do
 8:         $Base_{Seq}$ = $BOW_{BasePr}$ $[i : i + window_{length}]$
 9:         $Test_{Seq}$ = $BOW_{TestPr}$ $[i : i + window_{length}]$
10:         $Similarity_{Value}$ = ($Cosine\_Similarity$($Base_{Seq}$ , $Test_{Seq}$))
```

```
11:         if $Similarity_{Value}$ < 0.99 then
12:             Test Sequence is Anomalous
13:         end if
14:     end for
15: end for
```

We used different fixed window lengths and compute the cosine similarity for each window length. The test process is considered anomalous if a particular window has a similarity value of less than 0.99. From an example of the above algorithm shown in Figure 2.3, we observe that among the cosine similarities between normal and malicious sequences of three windows, the second window has a low similarity value since malicious test vectors have infected it.

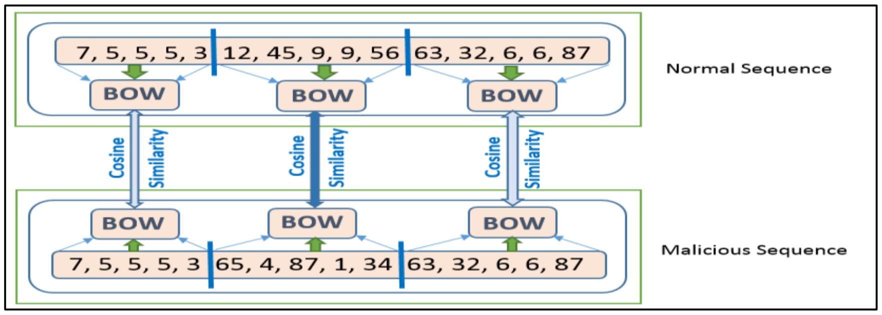


Figure 2.3: Anomalous window detection

2.5.3 Commonality aspects: Jaccard Similarity

Jaccard similarity is the ratio of the length of the intersection of two sets to the length of the union of the same two sets. Given two sets, X and Y, its Jaccard similarity is calculated as:

$$J(X,Y) = \frac{|X \cap Y|}{|X \cup Y|} \tag{2.16}$$

We developed the following anomaly detection algorithm that uses the Jaccard similarity to detect anomalies in the process under examination on Windows VM.

Algorithm 2 Anomaly Detection Algorithm using Jaccard Similarity

Input: Normal and Test System Call Sequence
Output: Similarity_Value

```
1:  Convert System Call Name to System Call Number using Mapping Table
2:  $Base_{UniqueSC}$ = {Unique System Call of Normal Process}
3:  $Test_{UniqueSC}$ = {Unique System Call of Test Process}
4:  $Intersection_{list}$ = $Base_{UniqueSC} \cap Test_{UniqueSC}$
5:  $Union_{list}$ = $Base_{UniqueSC} \cup Test_{UniqueSC}$
6:  $Similarity_{Value}$ = length ($\frac{Intersection_{list}}{Union_{list}}$)
7:  if $Similarity_{Value} < 0.99$ then
8:      Anomalous
9:  end if
```

Jaccard Anomaly Detection algorithm works well irrespective of the size of the system call sequence. Since its result depends only on the unique system calls of normal and test data, the ratio of the intersection of unique system calls to the union of unique system calls of these two datasets is not affected by the variability of the length of system call sequences.

2.6 Experimental Setup

The proposed framework runs on *Xen* 4.12 hypervisor with *libvert* 5.4.0 library to manage virtual machines. In the current implementation of this framework, the modules Introspector and Security Agent extract and process the system call information. System call traces were collected from inspecting the VM process under examination. System call features are extracted with LibVMI library in combination with rekall profiles of Google. This rekall profile is a file in JSON that comprises memory mappings and offsets of Windows data structures and other resources. The above-specified modules, i.e., Introspector and Security agent, are written in Go Language to process the request and extract the system call traces from VM with LibVMI functions. The LibVMI library handles introspection requests. The Libvirt library allows to create, start, and stop of virtual machines of Windows.

2.7 Results

The following subsections discuss the results of the LSTM Seq-Seq, Cosine Similarity, and Jaccard similarity algorithm.

2.7.1 Result analysis of LSTM Seq-Seq Model

We obtained system call sequences through the hypervisor and implemented LSTM Seq-Seq architecture to detect anomalies by considering the temporal ordering of system calls. We tune up the following hyper-parameters for the LSTM model:

- *Sequence length*: It is the length of the last processed system call sequence, used as input to predict the adjacent next set of system call sequences.
- *Epochs*: Number of iterations, a model passes through the entire data for training.
- *Batch-size*: Number of system call sequences passed to the LSTM network in a single iteration.

To evaluate the model, we define accuracy as:

$$Accuracy = \frac{number\ of\ correct\ system\ call\ sequence\ predictions}{total\ number\ of\ system\ call\ sequence\ predictions\ performed} \tag{2.17}$$

We trained the LSTM model with an Adam optimizer and categorical cross-entropy as a loss metric. We found that a batch size of 256 gives higher accuracy.

Figure 2.4 and Figure 2.5 show the accuracy and loss for different sequence lengths with batch sizes of 256 and epochs of 100. We found that the model trained with a sequence length of ten has the lowest loss and highest accuracy as listed in Table 2.1 and 2.2 . Our trained model gives 97% accuracy with a loss of 0.08.

Table 2.1: Accuracy with Windows Size

Window Size	Accuracy
3	0.94
10	0.97
15	0.96

Table 2.2: Loss with Windows Size

Window Size	Loss
3	0.20
10	0.08
15	0.10

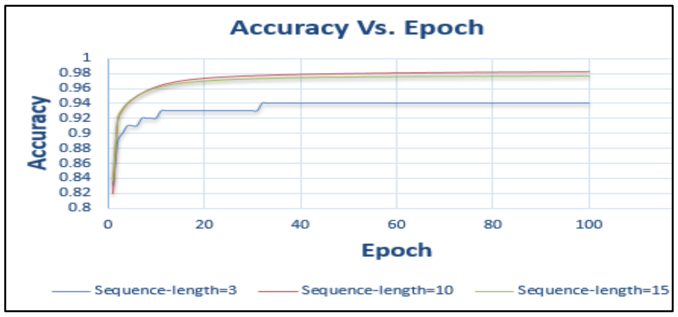


Figure 2. 4: Accuracy with Epoch for Individual Sequence Length

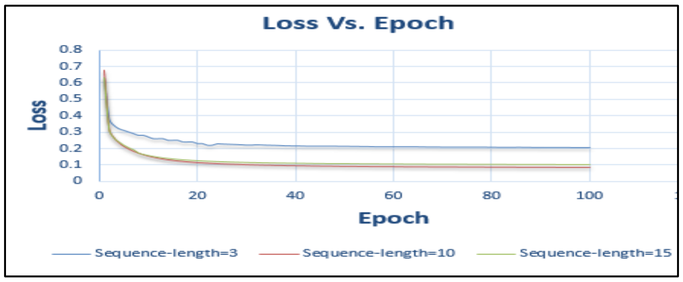


Figure 2. 5 : Loss with Epoch for Individual Sequence Length

From Figure 2.6, we observe that the time required for a neural network to train for one epoch increases with increasing sequence length.

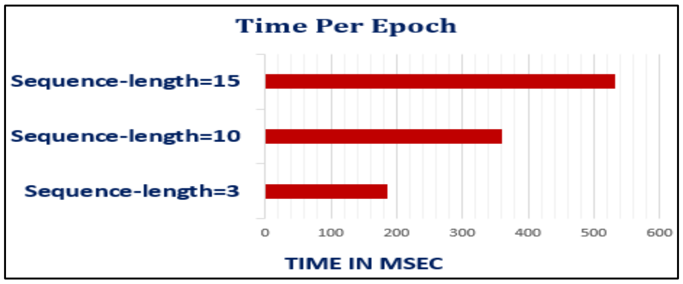


Figure 2.6: Time per Epoch for Individual Sequence Length

Our optimized LSTM model is trained with the benign data of a PUE. We find that the model gives 93% accuracy for benign test data and 0.02% for malicious test data during the testing phase.

2.7.2 Result analysis of Cosine Similarity anomaly detection algorithm

We trained a Cosine Similarity algorithm with the sequence size of 300K (maximum sequence length of an application) system calls. Figure 2.7 depicts the seven most frequently occurring system calls (*NtDelayExecution*, *NtQueryVolumeInformationFile*, *NtClose*, *NtCreateFile*, *NtQueryInformationFile*, *NtWriteFile*, *NtSetInformationFile*) of benign PUE (benign data) and corresponding system call frequency of the malicious PUE(malicious data).

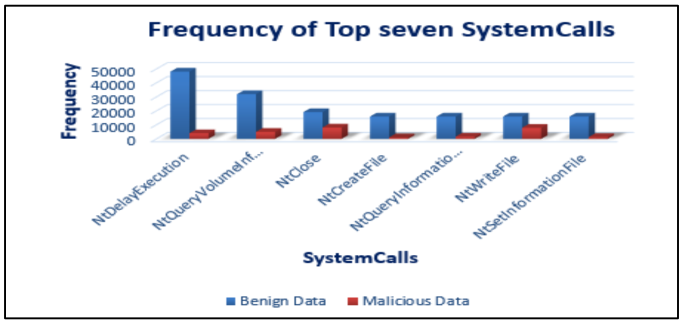


Figure 2.7: Top Seven SystemCalls in Benign and Malicious Data

We tested this algorithm in the following two scenarios. Since this algorithm is based on the frequency of the system call, it requires both benign and malicious data to be of the same length. We divided the sequence into windows of system calls, each with a length of 1000 system calls. This feature allows detecting anomalies at any particular time frame.

First Case: We compared two different benign system call sequences, each of size 300k system calls.

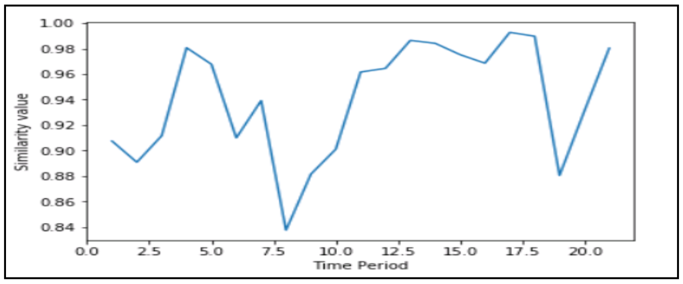


Figure 2.8: Cosine Similarity for Benign Data

From Figure 2.8, we observe that the cosine similarity between two different benign sequences ranges from 0.84 to 1.00. It means the two sequences are 84% to 100% similar.

Second Case: We compared a benign system call sequence with a malicious system call sequence generated through malicious test vectors.

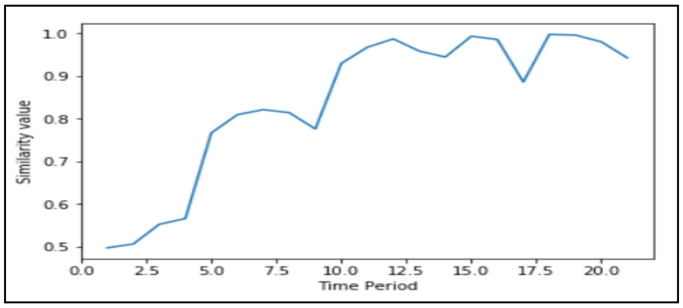


Figure 2.9: Cosine Similarity for Malicious Data

Malicious test vectors were injected for a short duration only at the beginning of the data collection. From Figure 2.9, we notice that the Cosine Similarity algorithm can easily capture the malicious activities by giving low similarity for that earlier time frame that was influenced by the malicious test vectors.

2.7.3 Result Analysis of Jaccard Similarity anomaly detection algorithm

The Jaccard Similarity anomaly detection algorithm considers the uniqueness of system call sequences rather than the frequency count of the system calls.

From Figure 2.10, we confirm that Jaccard similarity works well irrelevant of the size of both benign and malicious sequences. Additionally, it gives low similarity for malicious sequences and high value for benign sequences.

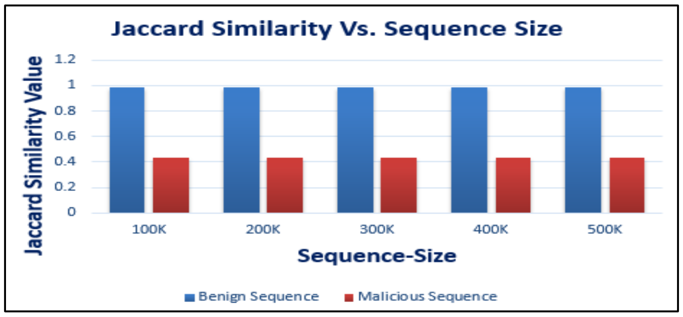


Figure 2.10: Jaccard Similarity

2.8 Conclusion

In this chapter, we presented the implementation of anomaly detection techniques by considering temporal, frequency, and commonality behaviors of system-call sequences. We applied LSTM sequence-to-sequence for temporal behavior, Cosine Similarity for frequency behavior, and Jaccard Similarity for commonality behavior to detect anomalies in a process under examination using system-call sequences of the process. These detection algorithms perform well in detecting anomalies from previously unseen data and across multiple machine configurations.

Among these three anomaly detection algorithms, Jaccard Similarity captures the least number of characteristics of the system call sequence while the algorithm's complexity is straightforward. Hence, its similarity result is very primitive in detecting anomalies. This algorithm is suitable only for coarse-level analysis.

LSTM seq-to-seq trains with several overlapped subsequences of the input system call sequence from the other two anomaly detection algorithms. Therefore, it provides a fine-grained detection of anomalies. On the other hand, it requires a long training period to learn the temporal behavior of the system call sequence. The Cosine Similarity algorithm detects anomalies reasonably well with improved accuracy than the Jaccard Similarity algorithm but is less fine-grained than the LSTM seq-to-seq algorithm.

From our experiments, we recommend using Cosine Similarity in the first phase of anomaly detection to find out the susceptible anomaly window time frames. For further fine-grained insights into these susceptible anomaly windows, apply LSTM seq-to-seq algorithm. This approach will reduce the overall analysis time and improve anomaly detection accuracy.

CHAPTER 3

P-BOSC: AN EXTENSION OF BAG OF SYSTEM CALL TECHNIQUE FOR DETECTION OF ANOMALOUS POINTS

With the rapid growth in technology, the impact of malware on the crucial system of computers has increased at an alarming rate. In this chapter, we utilized the *Natural Language Processing* technique, namely *Bag of Words*, and proposed an anomaly detection method. Each system call is considered as a single word. The proposed approach trains the model to learn the normal behavior of the processes running on virtual machines. We utilize virtual memory introspection to extract the sequence of system calls for the normal processes and malware-infected processes. Through data processing techniques such as filtering and redundancy removal, the extracted sequences are processed. The proposed algorithm employs cosine similarity for anomaly detection purposes. As an extension to this, we also proposed a novel window detection algorithm to detect the anomalous sequence window. Through experimental results, we achieved an anomaly detection rate of 99%.

3.1 Introduction

In today's world, there is an ever-growing usage of the distributed systems. Virtualization is one of the major components of such a system. Applicability is one of its popular reason. Virtualization protects the resources of the running systems, performs load balancing operations, and manages the sharing of the resources. Due to these reasons, numerous hypervisor-based approaches have been developed to scan virtual machines for threat detection. Currently, malware has become very sophisticated such that the current detection methods fail to detect it, and ultimately the virtual machines are compromised. Thus, several cyber threats need to be mitigated to protect expensive resources and sensitive data by numerous industries and organizations. The virtual memory introspection technique is most widely used at the hypervisor level to capture the running states of virtual machines, which is also used for memory forensic analysis [34, 35, 36].

The frequency-based approach is used by Kang et al. [58]. The sequence of system calls S is represented as a list $\{ M_1, M_2 . . . M_n \}$ in this approach, where the count of distinct system calls are represented by n and M_i represents the total count of an i^{th} system call in a single sequence.

Natural Language Processing based *Bag-Of-Words* (BoW) is very popular for text analysis purposes. In our context, we examine the richness of *BoW* by considering every system call as a single word for sequence analysis purposes. Based on this, we proposed an anomaly detection algorithm that can detect a window of the sequence which is malicious. The experimental results depict that analysis of system call sequences for detecting anomalies in the processes running on the hypervisor provides accurate detection.

The rest of the chapter is organized in the following way. Related work to the area of anomaly detection using system calls is discussed in section 3.2. The system overview is explained in section 3.3. Extraction of the features and pre-processing techniques are discussed in section 3.4. In section 3.5, the proposed window anomaly detection algorithm is described in detail. Section 3.6 gives an overview of the environmental setup. Experimental results with in-depth discussion are performed in section 3.7, with the conclusion in section 3.8.

3.2 Related Work

In a production-based system, to secure the software and its different components, it is imperative to classify and detect the malware. Two different types of analysis are widely researched in this area. They are *static analysis* and *dynamic analysis*. With the increase in threat through robust malware, there is a considerable growth of research in the area of anomaly detection.

The source files are analyzed directly without any execution in the static method of analysis [40]. Ye et al. [41] generate different association-based rules by employing an *association mining algorithm*. This leads to the development of *Intelligent Malicious code Detection System (IMDS)*, which generates the information pertaining to the import function. Finally, a rule-based classification algorithm was used for malware detection. Assembly-based instructions were used by Masud et al. [42], which were then transformed to bytecodes of *4-gram*. Furthermore, five unique static features were generated using feature engineering techniques. Finally, two machine learning-based classification algorithms, namely *decision trees* and *support vector machines*, were trained to classify malware. Nonetheless, there are some limitations to the static analysis approach. They are polymorphism, encryption, and many others. The analysis of the behavior of the application is termed dynamic analysis. Its overall idea is to examine the process during its execution [38]. This overcomes the limitations of static approaches.

Hidden Markov Models (HMMs) based classifiers are being utilized by many researchers currently for anomaly detection using system call sequences [12, 46]. To improve the precision rate and achieve a high detection rate, each author proposes a distinct variety of methods and techniques. HMMs trained by Alarifi and Wolthusen [43] utilize the sequences of system calls generated from VMs. Since their approach requires only a few examples to train HMM model, the detection rate was very low. Threshold Value followed by the probabilistic scoring of the sequence is analyzed by Yeung et al. [59]. Multi-layer anomaly detection model based on the sliding window technique is proposed by Hoang et al. [60]. Detection of an anomaly in the operations performed at user-level privileges is experimented with through HMMs by Cho et al. [14]. An extensive comparative analysis of *HMMs*, *RIPPER* [43], and *STIDE* [8] is performed by Warrender et al. [9]. Every method has its unique characteristics in terms of performance measurements. The storage requirement is very high for considerable sequence length during the training of HMMs. Furthermore, the need for heavy computational power increases with an increase in the multiple passes of the data. Modeling based on time series has been performed [56,71]. In another line of work, the distinct process of the kernel modules is being analyzed with the extraction of a sequence of system calls called *kernel state modeling* (KSM). First, it calculates the total count of states in the malicious sequences, followed by probability computation. Next, the comparative analysis is performed with the normal traces to analyze the deviation. For the UNM dataset, the detection rate of KSM is higher than HMMs. Various embedding based on the neural network is being utilized for the data with one dimension to extract multiple features [57, 61, 51, 48]. For multi-dimensional datasets, authors in [63, 64, 65, 66] developed learning-based approaches for the extraction of the features.

3.3 System Overview

In this section, we discussed the implementation of the proposed framework. This includes the approach of extraction of system call sequences using a technique based on VMI with its analysis. Various traces of system calls during a process execution are collected, and the anomalous behavior of such a process on a VM is analyzed. The proposed framework consists of *Architecture*, *Operational Methodology*, and the *development of custom malware*.

3.3.1 Architecture

The architecture of the proposed anomaly detection framework comprises four unique components: *Virtualization*, *Data Analysis*, *Malware Log*, and *Control Center*. At a high level, it performs the introspection of memory to extract the features, perform advanced analysis and finally visualize the results.

Individual modules with their sub-components are explained in the following sub-sections.

Virtualization: In this module, *VMI API* is employed on *VMs* for memory forensics and smart memory introspection. *Introspect* and *CyberAgent* are two sub-components of this module.

Introspect: It scans the VM's (operating on a hypervisor) memory to extract the essential data from low-level. This data is exported to the listener for analysis purposes. There is an interface between hypervisor and introspect. This allows complete control of the various states of the VMs. Some of the states are Run, Stop, and Shut-Down.

CyberAgent: It utilizes the *LibVMI* library to initiate the introspection of the memory. The primary objective is to mine the different data structure of memory from the virtual machines and transfer it to the analytics for analysis. This agent has a wide variety of capabilities. Such as initiation of the scans on the processes, extraction of invariant structures, and monitoring changes in the file.

Data Analysis: In this module, we have several learning-based algorithms utilized to train the model. Next, the prediction is performed on the test dataset using the trained model. Here, the data extracted during the normal behavior of the process is viewed as baseline data, and malware-infected process data is considered malicious or test vector data. Introspect module is used to extract these datasets and store them in the database for analysis purposes.

Malware Log: It comprises a numerous variety of malicious vectors which are used to manipulate the data structures at the kernel level. It contains malware for both Windows and Linux.

Control Center: This module provides an extensive user interface for a user to control and manage the whole framework architecture. Several VMs operations such as create, delete, and so on can be handled from this module. Furthermore, it can be used to execute the benign and malicious test vectors on the running VMs. The visualization of the results obtained through the data analysis module can be easily monitored by the user.

3.3.2 Proposed Methodology

The traces of the system calls are collected using the introspect and cyber agent module, which scans the memory of the running VMs. The listener collects the recorded information and stores it in the database. Furthermore, various one-class-based anomaly detection algorithms are trained using the collected data. The user can send a request to the introspect to control the VMs during the extraction of the system call traces.

To manipulate the sequences of the system calls, custom test vectors are developed by employing the method of DLL injection. Numerous distinct system calls with a particular frequency range are injected into the processes by creating a file hidden on the disk.

3.4 Feature Engineering

A method based on angle similarity is employed to analyze the behavior of the process. We utilize the total frequency of each system call in the sequences of a particular rather than considering the temporal nature of the sequences. In this work, we developed a technique based on the angle similarity, which is heavily used in NLP text classification. Here, each system calls sequence is emphasized as a single document, and its unique system call is considered a single word. The benign sequence of system calls is extracted during the normal behavior of the process running on a Xen Hypervisor. Table 3.1 depicts the short sample sequence of extracted system calls.

Table 3.1: Sample Sequence

System Call Data
NtOpenKey
NtQueryKey
NtSetInformationFile

NtQueryValueKey
NtQueryKey
NtQueryKey
NtClose
: :

For analysis using the proposed algorithm, every individual system call needs to be converted into its corresponding numeric mapping number. In windows, the total amount of system calls that are unique is *450*. Thus, each system call name will be converted into an integer number between *0 to 449*. A sample conversion is described in Table 3.2.

Table 3.2: Numeric Mapping of System Calls

NtOpenKey	**NtQuery Key**	**NtSetInfor mationFile**	**NtQuery ValueKey**	**NtQuery Key**	**NtQuery Key**	**NtClose**
18	22	39	23	22	22	15

Next, each sequence is converted to *450*-dimensional vector data. This approach is called *Bag of System Calls*, where the value in each cell represents the total number of system calls of the i^{th} column. Table 3.3 shows the transformed version of the extracted sequence of system calls.

Table 3.3: Vector of Frequencies

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**…**	**448**	**449**
1	0	0	0	6	10	17	0	…	0	0

3.5 Proposed Algorithm

The proposed anomaly detection algorithm uses the transformed format of the benign and test vector processes and calculates the cosine similarity between them. This similarity computes the cosine angle.

As shown in equation 3.1, the Euclidean dot product is utilized to calculate the cosine angle between two vectors.

$$A.B = |A||B| \cos\theta \tag{3.1}$$

The cosine similarity value for two *m* dimensional vectors *X* and *Y* is computed using *cos(θ)* function as shown below:

$$similarity = \cos(\theta) = \frac{X.Y}{\|X\|\|Y\|} = \frac{\sum_{i=1}^{m} X_i Y_i}{\sqrt{\sum_{i=1}^{m} X_i^2}\sqrt{\sum_{i=1}^{m} Y_i^2}} \tag{3.2}$$

3.5.1 Anomaly Detection Algorithm

This section explains the proposed algorithm that analyzes the process running on VM on a particular hypervisor to detect the anomalies.

Algorithm 1 takes as an input the sequence of system calls of the normal and the test vector processes and the system call mapping table. It gives processes that are anomalous as an output.

```
Algorithm 1 Anomaly Detection Algorithm
Input: System Call Sequence of Baseline and Test Processes, Mapping Table (M)
Output: Anomalous Process
  1: for sequences of baseline process $B(P_i) = (P_1, P_2, \dots P_n)$, do
```

```
 2:     Transform System Call Name to Unique Number
 3:     $BOSc_{BasePr}$ = Bag-of-SystemCalls for normal process
 4: end for
 5: for test processes $T(P_i) = (P_1, P_2, ... P_n)$ do
 6:     $C_{Pr}$ = $B(P_i) \cap T(P_i)$
 7:      Transform System Call Name to Unique Number for test process in $C_{Pr}$
 8:     $BOSc_{C_{Pr}}$ = Bag-of-words for test process
 9: end for
10: for combined test processes $C_{Pr}(P_i) = (P_1, P_2, ... P_n)$ do
11:     if $BOSc_{C_{Pr}}$ $\neq$ M (t) then
12:         $C_{Pr}(P_i)$ is anomalous
13:     else
14:         $Similarity_{Value}$ = ($Cosine\_Similarity$($BOSc_{BasePr}$ ,$BOSc_{C_{Pr}}$))
15:         if $Similarity_{Value}$ < 0.99 then
16:             $C_{Pr}(P_i)$  is Anomalous
17:         end if
18:     end if
18: end for
```

The algorithm converts the sequence of all normal processes into the Bag of system calls. The same transformation is applied to all the test processes. Next, the cosine similarity is checked between each normal and test process. If the cosine similarity value is less than 0.99, it is considered anomalous and returns as an output.

3.5.2 Anomalous Window Detection Algorithm

This section discusses the proposed algorithm that can detect the anomalous window in a sequence. This allows us to check what part of the sequence the attack occurs.

Algorithm 2 Anomalous Window Detection Algorithm

```
Input: Anomalous Process, sequence length
Output: Anomalous Window
 1: for Anomalous Processes $A(P_k)$ and Base Process $B(P_k)$ do
 2:     for k in range (len (BoSC[$A(P_k)$] - $Sequence_{length}$] do
 3:         Sequence($B(P_k)$) = BoSC[$B(P_i)$] [$k : k + Sequence_{length}$]
 4:         Sequence($A(P_k)$) = BoSC[$B(P_i)$] [$k : k + Sequence_{length}$]
 5:         $Similarity_{Value}$ = $Cosine\_Similarity$(Seq($B(P_k)$)  ,Seq($A(P_k)$))
 6:         if $Similarity_{Value}$ < 0.99 then
 7:             Sequence($A(P_k)$) is Anomalous
 8:         end if
 9:     end for
10: end for
```

Algorithm2 takes the anomalous process detected by Algorithm1 and the sequence length. Next, it divides the sequences of anomalous and base processes into a batch of sequences followed by their transformation into a Bag of System Calls. Finally, the cosine similarity value is computed between the individual batches of the base and anomalous process sequences. If the cosine similarity value is < 0.99, then that batch of the sequence is considered anomalous. Figure 3.1 shows the sample example.

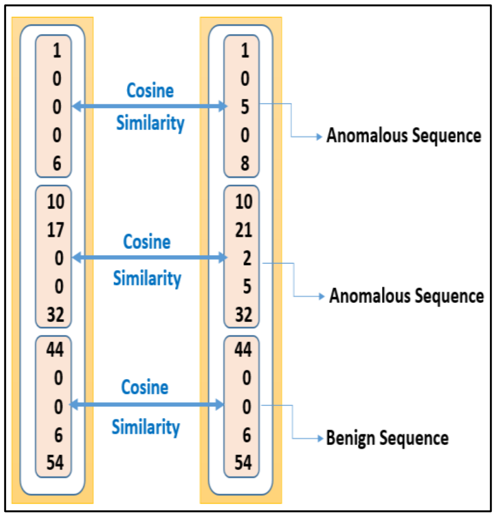


Figure 3. 1: Cosine Similarity Between Batches

3.6 Environmental Setup

Xen hypervisor is used in the development of the proposed framework. *Libvirt* library is used for controlling virtual machines. Next, the *DRAKVUF* library is used to get the virtual addresses of the memory during the execution of the processes. We have *CyberAgent* and Introspect modules in the current operation of the proposed framework. They use *Google's recall profile* with the *DRAKVUF* and employ the *LibVMI* library for extracting the traces of the system calls. Furthermore, it is used to control and manage the VMs running on Windows. The recall profile contains several kernel features of the data structures pertaining to the windows and is in the *JSON* format. *GO* language is used to develop those two modules. *Microsoft VS.Net* is used to develop the main application, containing user-specific *API* calls for initiating the communication. Next, the extracted data is stored in the database server using the agent. Lastly, the data is studied using advanced learning-based algorithms. Figure 3.2 depicts the overall framework.

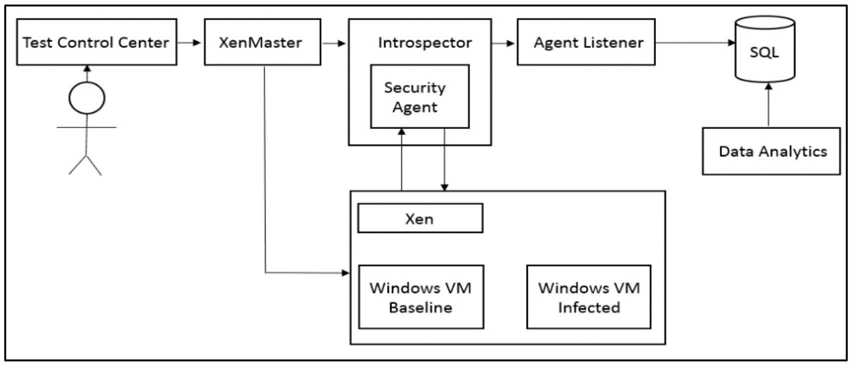


Figure 3.2: Environmental Framework

3.7 Results Analysis

The experimental results from the anomaly detection algorithm are conversed in this section.

3.7.1 Anomalous Process Detection Algorithm

The proposed algorithm is evaluated with the sequence of *1.5M* system calls. We have *450* unique system calls in the operating systems running under windows. The *top 5* unique system calls with their counts for the benign and anomalous processes are depicted in Figure 3.3 and 3.4, respectively. We notice the difference between the frequencies of the system calls of the normal process and the malicious process. Table 3.4 depicts the cosine similarity value of individual processes. From the experimental results, we deduct that the similarity value is lower for the malicious process than the normal processes.

Table 3.4: CS Evaluated Results

Application Type	Normal	Malicious
Cosine Score	0.99	0.20

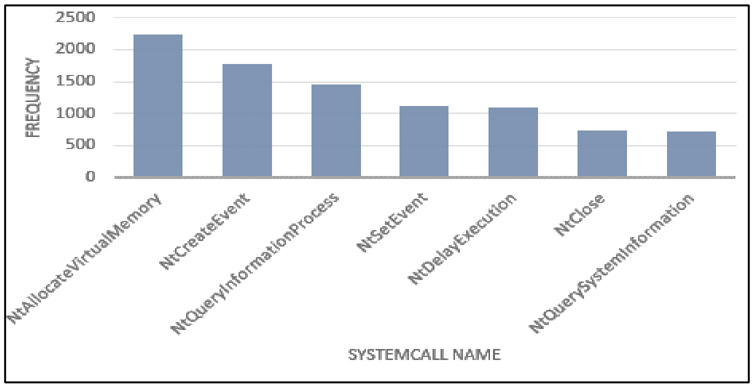


Figure 3.3: Top System Calls with highest frequency in Baseline Data

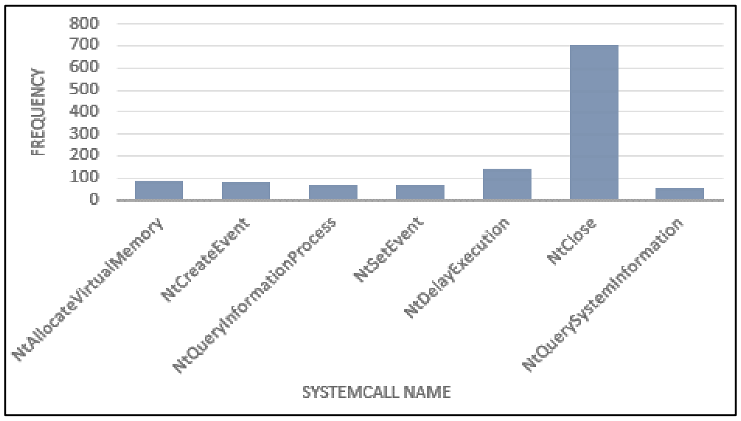


Figure 3.4: Top System Calls with highest frequency in Test Data

Furthermore, the proposed anomaly detection algorithm is not impacted by the arbitrary length of the sequence. This representation is depicted in figure 3.5. We perceive the similar cosine similarity value regardless of the sequence length.

3.7.2 Anomalous Window Detection Algorithm

We discuss the experimental results of the anomalous window detection algorithm in this section. Sequence length is one of the input parameters of this algorithm. Thus, we perform the experiments with varying sequence lengths. Based on Figure 3.6, we find that the sequence length with five system calls provides the optimal detection rate.

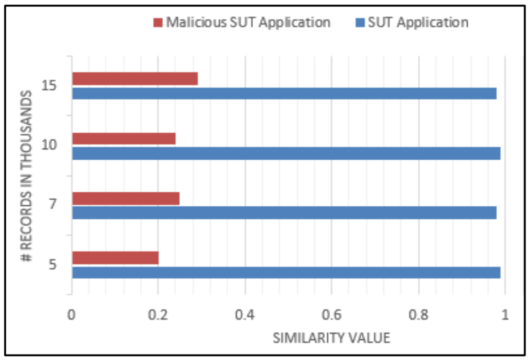


Figure 3.5: Cosine Similarity between Normal and Malicious Process

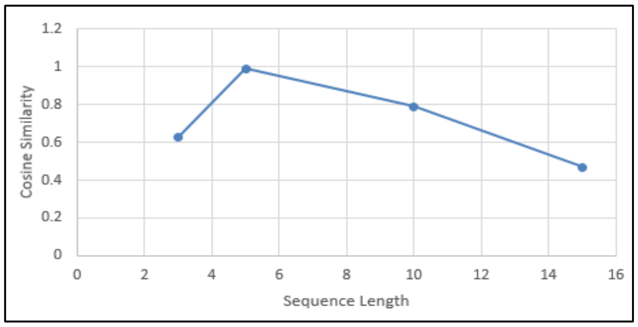


Figure 3.6: Cosine Similarity with Sequence Length

We also evaluate the proposed algorithm by running the processes with different scan times. Based on the experimental results showed in Figure 3.7, we observed that the cosine similarity value is regularly consistent for the sequence length of *5*, even when the scan times are different.

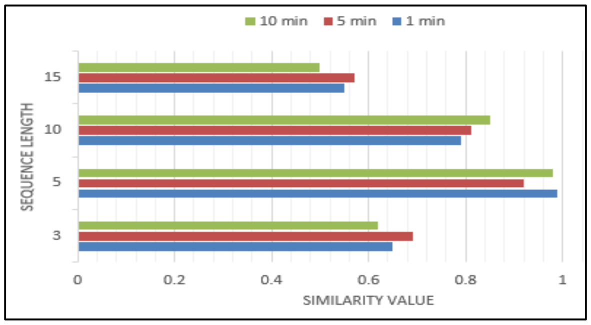


Figure 3.7: Cosine Similarity with varying scan duration

3.8 Conclusion

Various intrusion detection-based algorithms are developed with the hypothesis that the normal system routines differ vastly from the malicious or abnormal routines. The normal behavior of the program is learned by the anomaly detection algorithms. One behavior is the count of the occurrence of the system call executed by the process during its execution. Such sequences change with the interference of the malware, and thus it is one of the vital data structures for anomaly detection. In this chapter, we propose two algorithms for anomaly detection. These algorithms use natural language processing-based analysis to detect an anomaly in the process. Based on the frequency behavior of the sequence, the cosine similarity value is calculated by transforming the sequence of system calls to Bag of System calls. The first algorithm detects whether the process is anomalous or not, and the second algorithm identifies the specific window in the anomalous process. Based on the experimental results, we achieved an anomaly detection rate of 99%.

CHAPTER 4

DYNA-B: AN ENHANCED AND DYNAMIC BATCH SIZE TUNING FOR LSTM NEURAL NETWORK

In a wide variety of domains, it has been experimented that deep neural networks can be trained with increasingly large batch sizes without the loss of efficiency. However, such massive data parallelism differs from domain to domain. It is challenging computationally to train large deep neural networks on big datasets. To tackle these issues, there has been a surge in interest in utilizing large batch size values during the optimization part. A large batch size allows the training of deep neural networks faster. This enables developers and researchers to perform distributed processing. On the other hand, such utilization of large batch size during training possesses a very well-known problem called 'generalization gap' inducing the degradation in the performance across multiple datasets. There is minimal understanding of finding the optimal batch size value.

To address this problem, we present an adaptive tuning algorithm that can change the batch size adaptively. The proposed algorithm consists of four stages: *Gradient Warmup*, *Loss derivation*, *calculating the weighted loss* with the historical batch size, and finally *updating the batch size*. We showcase its superior performance compared to the traditional constant batch size approach. We make the comparison with multiple system call datasets with varying sizes.

4.1 Introduction

For large-scale empirical risk minimization, mini-batch stochastic gradient descent with its variants is the standard for the training of deep neural networks. These approaches are utilized with constant batch size values to evaluate empirical value. The determination of the variance in the estimates of the gradients is performed by the *stochastic gradient descent* (SGD) optimization algorithm. The behavior of this algorithm is impacted heavily by the batch size value. Also, during the process of optimization, the variance changes with constant batch size. This results in instability and non-convergence of the model to an optimal rate.

SGD and its variants are heavily used for the training purpose of the deep learning models. Such models need a large amount of data for training, and also such networks are oversized by design. Thus, the training dataset is divided into a series of fixed-size batches as an implementation. During the training, every batch is processed in sequence in every epoch. The individual samples of a single batch can be processed and trained parallelly [66, 67].

Currently, during the training of the neural network, the user typically chooses a static batch size *b*, which remains constant throughout the process. However, there are two crucial conflicts with this approach. Firstly, a small batch size value is essential because it allows the model to converge to the global optimal value. Secondly, a large batch size value improves the utilization of computational resources efficiency. Therefore, it is vital to have a trade-off between the values of batch size as being small or large during the training of the model.

4.1.1 Summary of contribution

We developed an automated batch size learning algorithm that dynamically changes the value of the batch size during training the model. We also developed a brute force method to compare the efficacy of the developed algorithm. Three different datasets are being used to evaluate the proposed algorithm. Each dataset selected for the evaluation is of varying sizes.

4.1.2 Organization of the chapter

In section 4.2, we discuss the related work in the area of batch size. Next, in section 4.3, we define and explain the problem formulation. Next, Gradient Descent and its variants are defined in section 4.4. The datasets used for the experiment purpose are described in section 4.5. In section 4.6, we discuss the metrics used to evaluate the model. The brute force algorithm implementation is described in section 4.7. The proposed dynamic batch size algorithm is explained in depth in section 4.8. The experimental results are discussed in section 4.9 with the conclusion in section 4.10.

4.2 Related Work

Dynamic updating of batch size has attracted significant attention recently. The variance of the gradient is utilized by Friedlander et al. [68] to derive a series of decreasing bounds. Their proposed approach converges faster and proves that an increase in batch size can replace the variability in the learning rate parameter. To prove it experimentally, they increase the batch size value to a constant factor (pre-specified) in each iteration without using a gradient estimate.

The closest to our work is performed by De et al. [69]. They use the estimate of the variance in the gradient. Defazio et al. [70] proposed SAGA, which aims to utilize the information of the gradient from the previous iteration to reduce the stochastic gradient's variance. Furthermore, their work examines the convergence behavior in terms of theoretical and empirical aspects of the convex optimization problems. Daneshmand et al. [71] combine various variance reduction methods with varying the sample size of the batch value. The limitation is that the sample size must be predefined before the training of the neural network model and is not dynamic at runtime.

However, for a particular dataset and a model, there is very little information as to how to set the batch size value. Also, how the value differs with different datasets and models. Researchers and developers simply experiment with varying batch sizes and see which value works the best. The downside of this approach is that it requires lots of experiments that need computational power and require careful tuning of the algorithm.

Our work performs the dynamic tuning of the batch size value in a two-step process: First, based on the loss generated after every batch of the input data passed, and secondly, perform the exponentially weighted sum of the historical loss to calculate the value for the subsequent batch size.

4.3 Problem Formulation

User processes interact with the operating system in the kernel mode. During the execution of the program, a piece of code is compiled. To execute the code, system calls are made. In this scenario, a sequence of system calls is a behavior of process interaction. If an intruder wants to manipulate the program, the sequence of the system calls will change.

Thus, to capture an anomaly in the process, it is essential to learn its sequence of system calls during the normal behavior. Various learning-based algorithms can learn the sequential data. We employed a deep learning-based Long Short Term Memory algorithm since it learns and captures long-term dependencies. Many hyperparameters need to be tuned. Below are a few of those hyper-parameters.

1) *Number of layers*: Total number of hidden layers.
2) *Number of nodes*: Total number of LSTM nodes in each layer.
3) *Epoch*: Total number of times, a dataset is passed to the model.
4) *Batch size*: Total number of data points, the model uses as a group to compute the loss and then update the weights.

Figure 4.1 shows the problem outline. The sequence of system calls is converted to a numeric format. LSTM model learns the function *F* of the mapping input sequence to the output sequence.

Let us say hypothetically that we have a sequence of *90* system calls. Then, we first convert it into input and output sequences. Each input and output have a length of *10* (called window size). *Window size* is the number of data points that are processed together at any instant. Window size cannot exceed the batch size. So, the total number of input-output sequences will be

$$\frac{Total\ number\ of\ system\ calls}{WindowSize} - 1 \tag{4.1}$$

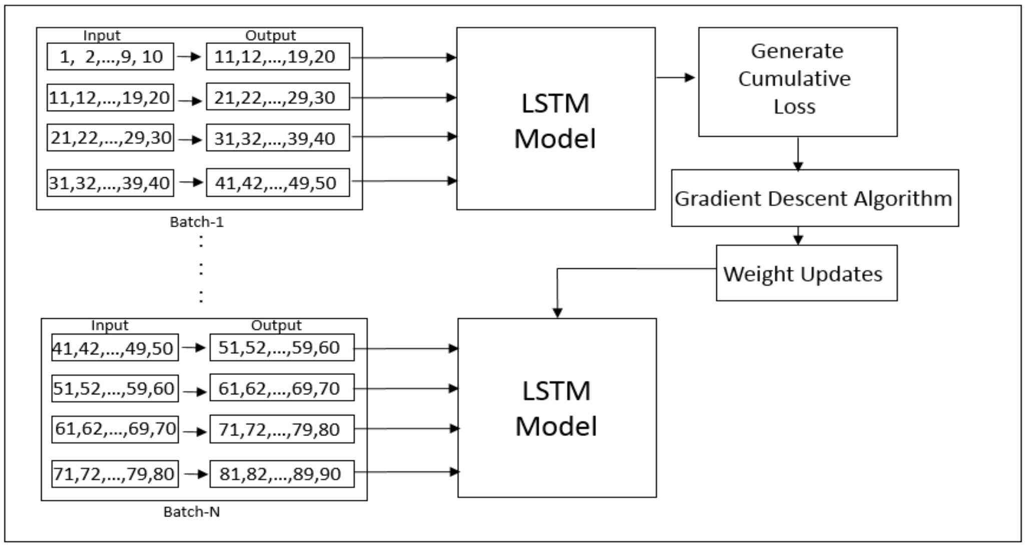


Figure 4.1: BatchSize Training Problem

Now, if the batch size is set to *2*, then in each batch, we will have four input-output sequences. A batch of data will be given to the LSTM model, which learns the mapping function. Based on generated loss, it will update the weights using a gradient descent algorithm. The LSTM model uses these updated weights for processing the next batch of data, and the process continues until we reach the last batch. Currently, the main bottleneck is that the batch size value has to be set before the training of the model. Once we set the batch size value, it is impossible to change during the training process.

Henceforth, to solve the problem mentioned above, we propose an iterative algorithm that can dynamically update the batch size value. This reduces the training time plus convergence to the global optima at a faster rate.

4.4 Gradient Descent and its variants

Gradient Descent plays a vital role in updating the weights of the neural network parameters. It is a way to minimize the objective function $J(\theta)$ by updating the model's parameters in the direction opposite to the gradient calculation $\nabla_\theta J(\theta)$. There are three different types of gradient descent. They are as follows:

Batch Gradient Descent (BGD): It uses the entire dataset to compute the gradient of the cost function. Therefore, the weights of the model parameters (θ) are updated only one time. It is not feasible for the big dataset.

$$\theta = \theta - \eta . \nabla_\theta J(\theta) \tag{4.2}$$

where η is the learning rate.

Stochastic Gradient Descent (SGD): It updates the weights of the model parameters after every individual data point. With such frequent updates, the model tends to overfit the training data.

$$\theta = \theta - \eta . \nabla_\theta J(\theta;\, x^i;\, y^i) \tag{4.3}$$

where x^i is the single input sequence, and y^i is the single output sequence.

Mini-Batch Gradient Descent (MBGD): It is an intermediate of *BGD* and *MBGD* where the weights of the model parameters are updated after every *n* input-output sequence. This way, the model can converge to optimal minima.

$$\theta = \theta - \eta . \nabla_\theta J(\theta;\, x^{(i:i+n)};\, y^{(i:i+n)}) \tag{4.4}$$

where *n* is the size of the mini-batch.

4.4.1 Optimizers

Nesterov Accelerated Gradient: It provides momentum to move the parameters θ in the right direction. The gradient is calculated by approximating the future position of the parameters.

$$v_t = \gamma v_{t-1} + \eta . \nabla_\theta J(\theta - \gamma v_{t-1}) \tag{4.5}$$

$$\theta = \theta - v_t \tag{4.6}$$

where γ is the momentum term with the default value of 0.9.

Adagrad: It is well suited for sparse data due to following reasons. First, for the parameters with infrequent features, it performs larger updates, whereas, for the parameters linked with the features occurring frequently, it performs smaller updates. The following equation updates the weights:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \varepsilon}} \odot g_t \tag{4.7}$$

where the sum of the squares of the previous gradients is G_t, ε is the smoothing factor, and g_t is the gradient at time step t.

Adadelta: It reduces the impact of decreasing the learning rate monotonically. It uses the fixed-size window to accumulate past gradients instead of considering all the gradients. It updates the weights based on the following update rule.

$$\Delta\theta_t - \frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t \tag{4.8}$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t \tag{4.9}$$

where $RMS[\Delta\theta]_{t-1}$ is the root mean square error of the updates of the parameters.

RMSprop: It computes the exponentially decaying average of the gradients and divides the learning rate with that value. The update rule is:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \varepsilon}} \odot g_t \tag{4.10}$$

where $E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1{g^2}_t$

Adam: It updates the weights by calculating the mean and uncentered variance of the gradients. The weight update rule is:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t + \varepsilon}} m_t \tag{4.11}$$

where $m_t = \frac{m_{t-1}}{1 - \beta_1^t}$ and β_1 and β_2 are the decay rates.

$$v_t = \frac{v_{t-1}}{1 - \beta_2^t} \tag{4.12}$$

Adamax: The adam optimizer uses the l_2 norm to calculate the v_t parameter. Furthermore, these two values are inversely proportional to each other. Adamax improves Adam by using $the\ l_p$ norm. The update rule is:

$$\theta_{t+1} = \theta_t - \frac{\eta}{u_t} m_t \tag{4.13}$$

Where u_t is the revised version of v_t.

Nadam: It is the combination of *Adam* and *NAG*. It performs accurate steps towards the optimal direction by performing parameter updates before gradient calculation. The weight update rule is:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t} + \varepsilon} \left(\beta_1 m_t + \frac{(1-\beta_1) g_t}{1 - \beta_1^t}\right. \tag{4.14}$$

4.5 Dataset Description

The following system call sequence datasets are used for experiment purposes. Each dataset is unique and has a varying size of sequences.

Joint Mission Planning System (JMPS) Dataset: The Joint Mission Planning System, otherwise known as JMPS, is an application that aggregates different target data, platform data, and land elevation data for the user to use in mapping route plans for missions. It offers route planning capabilities for aerial, land, and sea missions to choose respective platforms/vehicles and entities and their characteristics concerning their route, other entities, and themselves. A typical mission plan on JMPS might involve multiple user-made or imported aircraft routes belonging to ally, enemy, or other parties for a multitude of platforms, each with their radar and effective weapon ranges flying at different altitudes and speeds as to avoid detection or collision from some adversary or other entity (ground radars, mobile Surface-to-Air Missiles, jammers, etc.). This flexible layout of entities is displayed to the user as an order of battle in which the user plans around accordingly to fulfill some target or objective.

Test vectors injected into JMPS will thus slow down the application, achieve unauthorized access to data, and modify plan contents to the favor of some other actor as the plan report is being generated and sent. A test vector could inject many system threads to slow down the application, effectively behaving as a form of *denial-of-service* attack. Another test vector could attach itself to the process in kernel space and relay reads of process files to a desired outlet, compromising confidentiality. In the same way, a similar yet less general test vector can write specific *JMPS* data to process files that aid in altering a route so that an aircraft misses its target by however many units desired.

Table 4.1 shows the total number of system calls in this dataset.

Table 4.1: JMPS Dataset Information

Data Type	**Length**	**Category**
Training	1.6 Million	Normal
Testing	2 Million	Malware Infected

Australian Defense Force Academy Linux (ADFA-LD) Dataset: The host used to generate the sequence of system call data is configured with a Linux server. It captures the sequence of system calls during the normal operations of the process. Furthermore, several malware such as *Hydra-SSH*, *Hydra-FTP*, *Webshell*, *Add user, Java-Meterpreter* and *Meter-preter* are injected during the ongoing operation of a legitimate process. Table 4.2 and 4.3 shows the data distribution for normal and malware-injected traces respectively.

Table 4.2: ADFA-LD Dataset Distribution for Normal Category

Types of Trace	**Number of Traces**	**Category**
Training	833	Normal
Validation	4373	Normal

Table 4.3: ADFA-LD Dataset Distribution for Attack Category

Types of Trace	**Number of Traces**	**Category**
Hydra-SSH	148	Attack
Hydra-FTP	162	Attack
Webshell	118	Attack
Adduser	91	Attack
Java-Meterpreter	125	Attack
Meterpreter	75	Attack

University of New Mexico (UNM) Dataset: Each trace is generated by running one program. It has three types of files. The *Sun file types* contain information on “*synthetic sendmail CERT*”, “*synthetic sendmail*”, "*live lpr MIT*" and "*live lpr UNM*". The second type is *Linux*, and it contains information about *DoS*, *inet*, *ps*, *login*, and *live named*. Lastly, the third one is from the *new Linux*, which has information about *xlock* and *synthetic ftp*. The experiments are performed on *sun file types*. Table 4.4 shows the data distribution.

Table 4.4: UNM Dataset Distribution

Program Name	**Normal Traces**	**Malicious Traces**
UNM Live lpr	1232	1001
Live lpr MIT	2704	1001
Synthetic sendmail	7	10
Synthetic sendmail CERT	2	6

4.6 Evaluation Metrics

The following metrics evaluate the proposed algorithm experiments on the datasets discussed in section 4.5.

Mean Absolute Error (MAE): These metric estimates mean of the absolute difference between actual input sequences of system calls to the predicted sequences of system calls.

$$MAE = \frac{1}{N}\sum | Y - \overline{Y} | \tag{4.15}$$

Where *N* is the total number of sequences, *Y* is the actual input sequence, and $\overline{Y}$ is the input sequence predicted by the trained model.

Mean Squared Error (MSE): It acts the same as *MAE*. The only difference is that it computes the squared deviations between the actual and the predicted sequences instead of taking the absolute value.

$$MSE = \frac{1}{N}\sum \left(Y - \overline{Y} \right)^2 \tag{4.16}$$

Root Mean Squared Error (RMSE): It applies the square root function to *MSE*.

$$\sqrt{\frac{1}{N}\sum \left(Y - \overline{Y} \right)^2} \tag{4.17}$$

Mean Absolute Percentage Error (MAPE): It measures the accuracy of the prediction value.

$$MAPE = \frac{100\%}{n}\sum_{t=1}^{n} \frac{Y_t - \overline{Y_t}}{|Y_t|} \tag{4.18}$$

4.7 Algorithm Analysis

This section discusses the brute force approach to finding the optimal batch size value. This approach is naïve and needs lots of computational time.

Brute force approach

The brute force approach is described in *Algorithm 1*. It inputs the sequences of system calls and outputs the optimal batch size value. It starts by finding the initial interval range of batch size value. Next, for each batch size value in that interval range, we train the LSTM model and select the batch size with minimum loss as the local optimal batch size. Using this local optimal batch size, we narrow the initial interval range of the batch size to less range of values.

Algorithm 1 Brute Force Algorithm for finding Optimal Batch Size Value

```
Input: Normal System Call Sequences
Output: Optimal Batch Size Value
 1: N = Total length of Sequence
 2: X = $\sqrt{N}$
 3: OptimalBatchSize = NULL
 4: GlobalMinLoss = NULL
 5: IntervalList = [1… X... 2X]
 6: $Current_{IntervalStep}$ = $floor(\sqrt{2X})$
 7: BatchList = []
 8: for i in range (1, 2X, $Current_{IntervalStep}$ + 1) do
 9:     Batchlist.append(i)
10: end for
10: LossList = []
11: for j in range ($Batchlist_{length}$) do
12:     Loss = Train LSTM algorithm with Batchlist[j] and output the final loss
13:     LossList.append(Loss)
14: end for
15: MinLoss = Min(LossList)
16: $Current_{BestBatchSize}$ = BatchSize with MinLoss
17: if (MinLoss < GlobalMinLoss) then
18:     OptimalBatchSize = $Current_{BestBatchSize}$
19: end if
20: start = $Current_{BestBatchSize}$ - $Current_{IntervalStep}$ + 1
21: end = $Current_{BestBatchSize}$ + $Current_{IntervalStep}$ – 1
22: NextIntervalStep = $floor(\sqrt{2 * (Current_{IntervalStep} - 1)})$
23: BatchList = []
24: for k in range (start, end, NextIntervalStep + 1) do
25:     BatchList.append(k)
26: end for
```

27: Repeat step *11* to *26* until BatchList=NULL

Table 4.5 depicts the different number of the batch size that has been searched by *Algorithm 1* to find the optimal batch size value for all the three datasets. The LSTM model is trained with a default learning rate (i.e., 0.001).

Table 4.5: Static BatchSize Algorithm Results

Dataset	**Size in Length**	**Number of BatchSize Searched**
JMPS	1.5 Million	76
ADFA-LD	300,000	54
UNM	500,000	66

The model is trained with various optimizers listed in the *section 4.4*. We found out that *Adam*, *Nadam*, and *Adamax* give the optimal loss value based on the experimental result. Table 4.6 shows the optimal batch size value found using the *brute force algorithm* with a default learning rate of *0.001* for each dataset with the respective metrics and total training time in terms of Epoch.

Table 4.6: Static BatchSize Algorithm Results Analysis

Dataset	**Optimal BatchSize**	**Minimum Loss**	**Total #Epoch**
JMPS	596	0.193	92
ADFA-LD	312	0.125	65
UNM	402	0.263	72

We further applied the brute force algorithm by trying different learning rate values. The learning rate value experimented are: *{1, 0.1, 0.01, 0.0001, 0.00001, 0.000001}*. We pick one learning rate value from the list and run the brute force algorithm each time. Table 4.7 shows the experimental results. For two of the datasets, the result improved compared to the default learning rate.

Table 4.7: Static BatchSize Algorithm Results Analysis with Multiple LR

Dataset	**Optimal BatchSize**	**Best Learning Rate**	**Minimum Loss**	**Total #Epoch**
JMPS	501	0.00001	0.0732	88
ADFA-LD	312	0.001	0.125	65
UNM	437	0.00001	0.171	74

We found out that the learning rate of *0.00001* was optimal for *JMPS* and *UNM* datasets. Figure 4.2, 4.3 and 4.4 shows the decreasing loss value for JMPS, ADFA-LD, and UNM datasets respectively.

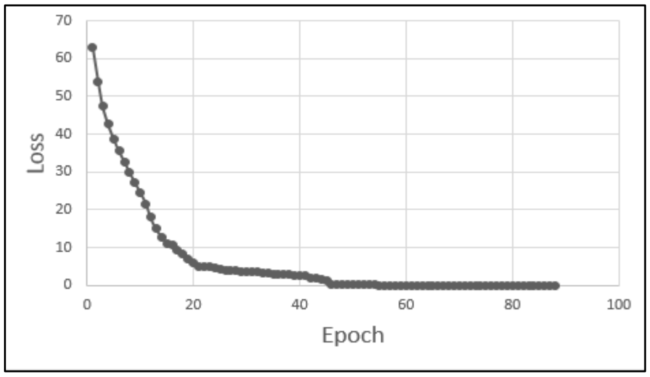


Figure 4.2: Brute Force BS for JMPS

For the JMPS dataset, the lowest loss achieved was *0.0732* at epoch *88* with an optimal batch size value of *501* and a learning rate of *0.00001*.

For the ADFA-LD dataset, the lowest loss achieved was *0.125* at epoch *65* with an optimal batch size value of *312* and a learning rate of *0.001*.

For the UNM dataset, the lowest loss achieved was *0.171* at epoch *74* with an optimal batch size value of *437* and a learning rate of *0.00001*.

To improve the training time in terms of the total number of epochs, we propose dynamic batch size value selection, which is discussed in the next section.

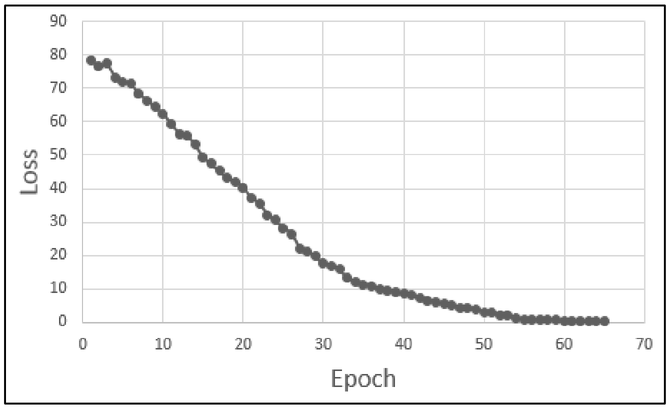


Figure 4.3: Brute Force BS for ADFA-LD

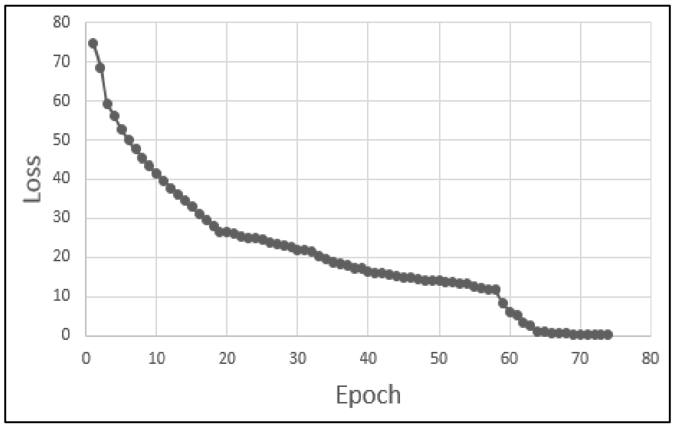


Figure 4.4: Brute Force BS for UNM

4.8 Proposed Dynamic BatchSize Algorithm

Algorithm 2 Dynamic BatchSize Algorithm

Input: Normal System Call Sequences

Output: Optimized Model

```
1: S= Normal System Call Sequences
2: N = Total length of Sequence
3: $Dynamic_{BS}$ = [ ]
4: $Local_{LL}$ = [ ]
5: $Global_{LL}$ = [ ]
6: J = 0
7: $Remaining_{DataPoints}$ = N
```

```
 8: for $i$ = 1 to 2 do
 9:    BS=$floor(\sqrt{N})$
10:    $Dynamic_{BS}$.append(BS)
11:    $Current_{BatchSet}$ = S [$j$ : $j$ + $BS$ + $1$]
12:    $Loss_{Value}$ = TrainOnBatch($Current_{BatchSet}$)
13:    $Local_{LL}$.append($Loss_{Value}$)
14:    $Global_{LL}$.append($Loss_{Value}$)
15:    J = BS
16:    $Remaining_{DataPoints}$ = $Remaining_{DataPoints}$ - J
17:    $New_{BS}$ = J
18:    $Current_{DataIndex}$ = 2 * J + 1
19: end for
20: $DeltaBS = [\ ]$
21: $Initial_{DeltaBS}$ = 0
22: while ($Current_{DataIndex}$ < N) do
23:    $Current_{BatchSize}$ = $New_{BS}$
24:    $i$ = len($Global_{LL}$) - 1
25:    $Current_{DeltaBS} = \frac{(Global_{LL}[i-1] - Global_{LL}[i])}{\max(|Global_{LL}[i-1]|,\ |Global_{LL}[i]|)}$
26:    $DeltaBS$.append($Current_{DeltaBS}$)
27:    $Initial_{Weight}$ = 0.50
28:    $DeltaBS_{MultiplyWeightList}$= [ ]
29:    for $k$ in range (len($DeltaBS$), 0, -1) do
30:       $DeltaBS_{Weight}$ = $Initial_{Weight}$ * $DeltaBS$ [$k$ - 1]
31:       $DeltaBS_{WeightList}$.append($DeltaBS_{Weight}$)
32:       $Initial_{Weight} = \frac{Initial_{Weight}}{2}$
33:    end for
34:    $TotalDeltaBS_{Weight}$ = $\sum DeltaBS_{WeightList}$
35:    $New_{BS}$ = ceil($Current_{BS}$ * (1 + $TotalDeltaBS_{Weight}$))
36:    $Dynamic_{BS}$.append($New_{BS}$)
37:    $Remaining_{DataPoints}$ = $Remaining_{DataPoints}$ - $New_{BS}$
38:    $Current_{BatchSet}$ = S[$Current_{DataIndex}$ : $Current_{DataIndex}$ + $New_{BS}$ + 1]
39:    Values= TrainOnBatch($Current_{BatchSet}$)
40:    $Local_{LL}$.append(Values[0])
41:    $Global_{LL}$.append(Values[0])
42:    $Current_{DataIndex}$ = $Current_{DataIndex}$ + $New_{BS}$
43: end while
Repeat step 21 to 40 for multiple epoch until the model converges
```

The proposed adaptive algorithm works in the following way. Initially, for the first epoch, the value for two batch sizes is selected as $\sqrt{N}$. This is called the warm-up phase, where the batch size is selectively constant. Now, the batch size value is calculated adaptively starting from the third batch. It is computed as follows. We compute the difference between the loss generated from the previous batch to the current batch. To normalize this loss difference, we divide it by the loss value, which is the maximum. Next, we apply the weight to each of the previous losses generated with the previous selection of the batch size. As we go higher up to the previous losses, the weight is reduced to half. This is important because, with this approach, we enforce giving higher weightage to the current batch size than that of the previous batch size selection. This weight value is multiplied by their corresponding loss value, and finally, they all are added to produce a single value. Now, to check whether to increase or decrease the batch size value, we add the resultant sum value to 1. Furthermore, we multiply this additive term by the current batch size value to get the final batch size for the next iteration. This new batch size is used to select the chunk of the data for training the model—this way, and the batch size is selected adaptively. The above steps are repeated every time to calculate the batch size value. Once we reach the end of the dataset, it is called one epoch. We repeat the process for further epochs until the model converges.

4.9 Experimental Results

The following are the results from training the LSTM model using the proposed adaptive batch size selection algorithm.

Table 4.8 shows the minimum loss achieved with all the datasets and the number of epochs needed to achieve minimum loss. This dataset shows the experimental results performed by keeping the default learning rate of *0.001*.

Table 4.8: Dynamic BS with Default LR

Dataset	**Minimum Loss**	**Total #Epoch**
JMPS	0.0765	79
ADFA-LD	0.142	56
UNM	0.186	65

Furthermore, we experimented with the proposed algorithm by changing the default learning rate value. Different learning rate experimented are *{1, 0.1, 0.01, 0.0001, 0.00001, 0.000001}*. We pick one learning rate value from the list and run the proposed algorithm each time. Table 4.9 shows the experimental results. For two of the datasets, the result improved compared to the default learning rate of 0.001.

Table 4.9: Dynamic BS with Multiple LR

Dataset	**Best Learning Rate**	**Minimum Loss**	**Total #Epoch**
JMPS	0.00001	0.0766	74
ADFA-LD	0.001	0.142	56
UNM	0.00001	0.181	62

In the proposed algorithm's case, we also found out that the learning rate of *0.00001* was optimal for JMPS and UNM datasets. Figure 4.5, 4.6 and 4.7 shows the decreasing loss value for JMPS, ADFA-LD, and UNM datasets.

For the JMPS dataset, the lowest loss achieved was 0.0766 at epoch 74 with a learning rate of 0.00001.

For the ADFA-LD dataset, the lowest loss achieved was 0.142 at epoch 56 with a learning rate of 0.001.

For the UNM dataset, the lowest loss achieved was 0.181 at epoch 62 with a learning rate of 0.00001.

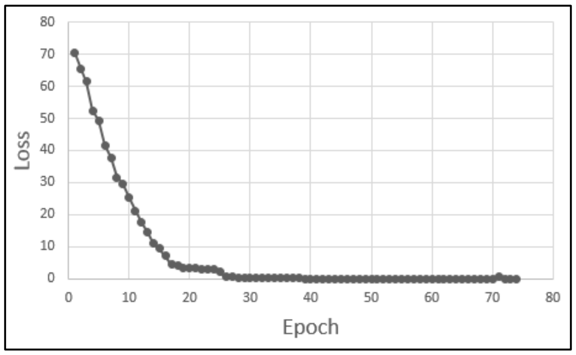


Figure 4.5: Dynamic BS for JMPS

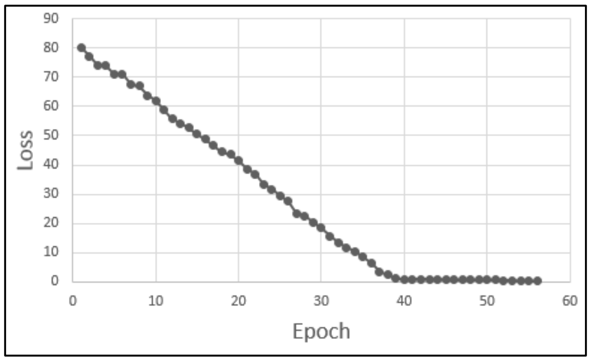


Figure 4. 6: Dynamic BS for ADFA-LD

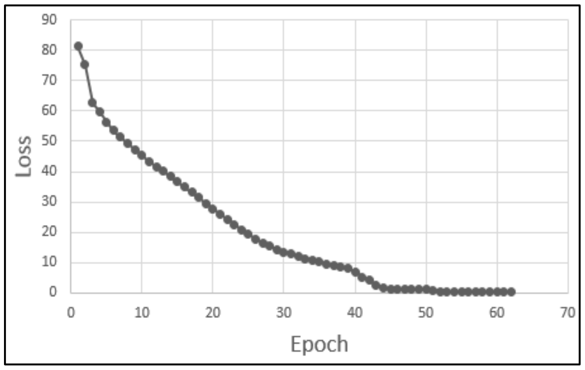


Figure 4.7: Dynamic BS for UNM

For the JMPS dataset, compared to static batch size selection, the number of training epochs reduced to *74*, which is *14* epochs less, with the loss reaching *0.0766*, which is *0.0034* higher.

For the ADFA-LD dataset, compared to static batch size selection, the number of training epochs was reduced to *56*, which is *11* epochs less, with the loss reaching *0.142*, which is *0.017* higher.

For the UNM dataset, compared to static batch size selection, the number of training epochs was reduced to *62*, which is *12* epochs less, with the loss reaching *0.181*, which is *0.010* higher.

4.10 Conclusion

Deep Neural Networks have demonstrated state-of-the-art results in numerous wide-range of application domains. Such neural networks require lots of hyper-parameter tuning to generalize well. Tuning takes a vast amount of computational training time. The training time and generalizability depend on how often the model's weights are adjusted and the total rows of the dataset (called batch size) used to update those weights. Currently, the batch size has to be set before training the model. This limits the model's capability to reach optimal minimum loss at a reduced epoch. Thus, we proposed a dynamic batch size selection algorithm that dynamically updates the batch size value. The update mechanism is based on the historical loss achieved by historical batch size values. We experimented our proposed approach with three different datasets. ADFA-LD and UNM are open sources of benchmark datasets. We found that dynamically updating the batch size value trains the model at a faster epoch rate compared to the static batch size value with experimental results. Furthermore, we notice that changing a default constant learning rate impacts the training time of the model.

CHAPTER 5

DYNAB-LR: A HYBRID ALGORITHM FOR DYNAMIC BATCH SIZE AND LEARNING RATE TUNING FOR AN OPTIMIZED TRAINING OF NEURAL NETWORK

Learning-based algorithms are adopted widely to solve complex problems in the current world. Such a model requires tuning of the hyperparameters to achieve the optimal loss. Learning rate impacts the rate at which the model updates the weights. It is a crucial problem to adjust the learning rate schedule in stochastic gradient methods. If the parameters such as convexity constants are known prior, theoretical schedules can be computed. Nonetheless, these parameters are not known, and most of the current loss function is concave in nature. Thus, we propose a dynamic learning rate schedule that can dynamically update the value of the learning rate. This is an added extension on top of the dynamic update of the batch size. Therefore, we update both the batch size and learning rate dynamically in a consecutive manner. Multiple experiments have been conducted using various optimizers to assess our proposed approach. Using the proposed approach, we train a deep learning-based LSTM algorithm widely adopted for sequential data. Furthermore, we validate it with three different datasets of various sizes and distinct in nature. From the experimental results, we infer that the proposed approach allows the model to train faster and reach the minimum loss at a faster rate.

5.1 Introduction

Deep Neural Networks (DNNs) are powerful and widely used learning-based networks competent enough in computing and learning techniques. In DNNs, the information flows from the input to the hidden layer and finally reaches the output layer. Each neuron in a particular layer computes the weighted sum from the previous layer's output and generates the output, fed as input to the next layer. Training is performed through a backpropagation algorithm. This algorithm uses the stochastic gradient descent method to update the weights. The main power of *DNNs* comes from the backpropagation algorithm. It computes the gradients to update the weights using the chain rule.

A constant learning rate is used in broad research areas that use backpropagation to train the model [72]. However, for the optimal performance of the training, the tuning and design of the learning rate hyper-parameter are essential. The conventional way is to compute the statistical characteristics. *Momentum usage* [73] and *simulated annealing* [74] are some of the optimization methods in this research area. Hayjin, in his research, proposed that if the *learning rate* $< \left(\frac{2}{\lambda_{max}}\right)$, then variance estimation can be used by gradient descent algorithm to converge, where λ_{max} is the maximum value of the eigenvector for a particular input feature vector. The limitation of the above rule is that it fails with the increasing length of the input feature vector. An optimized scheme of the first search and converge was proposed by Darken [75]. In his approach, he reduced the learning rate parameter by increasing the number of iterations. However, this approach requires high computational time and is often not considered a neural learning process. Instead of adjusting the weight parameters, Duchi et al. [76] and Zeiler et al. [77] propose *Adagrad* and *Adadelta*, respectively, to adjust the gradient's direction. Using the first order *Newtonian method*, they adjust the approximate value of its second order. Their approach works well in the early training stages. However, towards the end of the training, it deteriorates, and learning slows down, failing to reach the optimal loss value.

5.1.1 Summary of contribution

We developed an automated learning rate tuning algorithm that dynamically changes the value of the learning rate during the training of the model. The proposed tuning algorithm is an extension of the batch size tuning algorithm. Here, both hyper-parameters, the batch size, and the learning rate are tuned dynamically and consecutively. We also developed a brute force method to compare the efficacy of the developed algorithm. Three different datasets that are sequential in nature are being used to evaluate the proposed algorithm. Each dataset selected for the evaluation is of varying sizes and belongs to different domains.

5.1.2 Organization of the chapter

In section 5.2, we discuss the related work in the area of learning rate updates. Next, in section 5.3, we define and explain the problem formulation. Next, learning rate preliminaries and their variants are defined in section 5.4. Brute force algorithm implementation is described in section 5.5. In section 5.6, we discuss the impact of the window size on the training of the network. The proposed dynamic learning rate tuning algorithm is explained in depth in section 5.7, with the discussion of the experimental results in section 5.8. Finally, we conclude in section 5.9.

5.2 Related Work

Researchers have been actively focusing on stable and fast optimization algorithms. Despite the simplistic nature of the stochastic gradient descent algorithm, it is heavily used in various science and engineering domains. There are numerous rules for setting the value of the learning rate parameter. Given a list of prior statistical assumptions based on a particular loss function *f*, each rule has its way of justifying the convergence of the model. In the stochastic gradient rule, setting the learning rate value differently for different components is advantageous. The learning rate is set to a smaller value for the components with higher gradients and vice-versa. In some cases, such a heuristical approach is justified theoretically. Adaptive gradient methods use the root mean square's reciprocal value to update the learning rate of each parameter. The adaptive gradient's limitation is that it needs data to be sparse since sparse parameters are very informative. To overcome this issue, different modifications of the adaptive gradient have been proposed recently. RMSprop [78], Adam [79], Adadelta [80] and Nadam [81]. However, these approaches have no guaranteed results of convergence. Since the underlying problem of optimization changes due to the biased updates of the gradient, such an adaptive learning method becomes infeasible for the learning rate tuning problem. To mitigate this problem, Vasvani et al. [82] use the low value of learning rate for initial epochs.

Another approach toward the learning rate tuning has been proposed by Needell et al. [83] in another line of work. The authors use Lipschitz constants for setting distinct and constant but different learning rates for various component variables. They compute the loss function as a summation of its component variables by performing sampling during the gradient descent method.

Due to the lack of experimental results, specifically in sequential learning with the deep neural network such as LSTM, there is no guarantee that the model will consistently reduce loss value with the warmup approach. Also, there is no such rule of thumb for conducting data-specific warmup experiments. Thus, researchers and domain experts apply the trial and error approach technique by applying different settings in different applications. This is computationally inefficient and requires a lot of training time.

In this chapter, we propose an algorithm that can adaptively tune the learning rate parameter along with the tuning of the batch size parameter. The proposed approach does not require any manual tuning.

5.3 Problem Formulation

For a user process to interact with the operating system, it has to operate in the kernel mode. A batch of code is compiled during the process runtime. System calls are made to execute a particular line of code. If we arrange all the system calls made by a process during its execution, we can define the normal behavior. This sequence of system calls is temporal and thus can be used to train the learning-based models. We employed a deep learning-based Long Short Term Memory algorithm since it learns and captures long-term dependencies. Many hyperparameters need to be tuned. Below are a few of those hyper-parameters that affect the training time.

1) *Epoch*: Total amount of time a dataset is passed to the model.

2) *Batch size*: The total number of examples the model uses to generate the loss and update the weights further.

3) *Learning Rate*: A constant factor that controls the amount of gradient loss that needs to be applied to the current weight.

Figure 5.1 shows the problem outline. The sequence of system calls is converted to a numeric format. LSTM model learns the function *F* of the mapping input sequence to the output sequence.

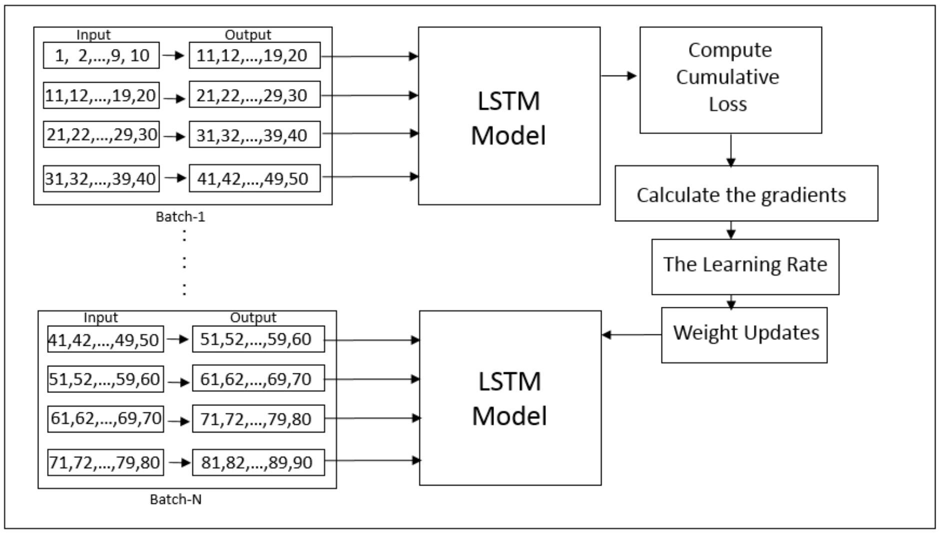


Figure 5.1: Learning Rate Problem Outline

The LSTM model will take an *input/output* sequence of system calls. Each input and output is of identical length. With this, the model learns the normal sequential behavior of the process. Thus, anything that deviates from the normal sequence of system calls is considered an anomaly during testing. First, the model takes the ground-truth input/output sequence and predicts the output sequence. The predicted output sequence is compared with the actual output sequence, and loss is computed. This step is repeated for all the input/output sequences of a particular batch, and the cumulative loss is generated at the end of the batch. Next, the gradient of the cost function is computed based on the generated loss. Then the constant learning rate parameter is then applied to the gradient value to update the weight value for every parameter. This whole process is continued for all the batches. Once all the batch of data is passed to the model, it is called an epoch. The model is trained with multiple epochs to learn and reach the optimal loss value.

Currently, the learning rate parameter has to be defined prior to the training of the model, and it remains constant throughout the training period. This bottleneck limits the model from reaching the optimal loss value at an earlier epoch time.

Henceforth, to solve the problem mentioned above, we propose an iterative algorithm that can dynamically update the learning rate value. This results in reducing the training time plus convergence to the global optima at a faster rate.

5.4 Preliminaries

Learning rate: It is a hyper-parameter that regulates the weight adjustment concerning the gradient computed based on loss. If the value is set to low, the model will take a long time to reach the optimal loss value. This low value may seem legitimate since we do not want to miss the local minimum, but on the other hand, it will take a considerable amount of time to converge if it gets stuck in the plateau region.

The equation for calculating the new weight based on learning rate is as follows:

$$current_{weight} = previous_{weight} - \eta * gradient \tag{5.1}$$

where η is the learning rate.

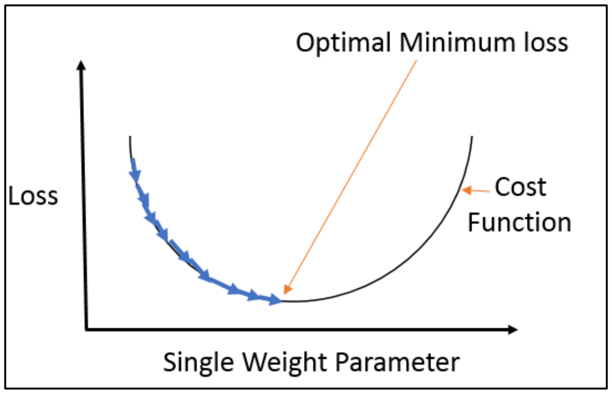


Figure 5. 2: Gradient Descent with small learning rate

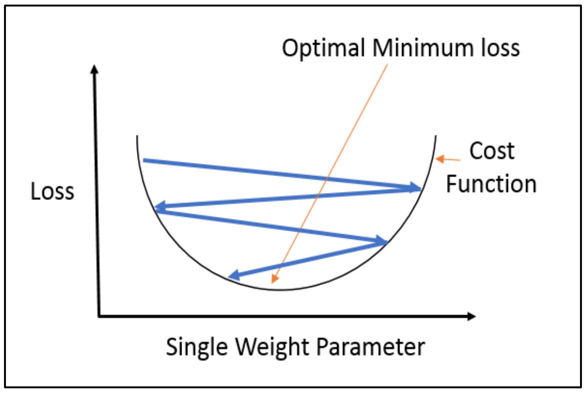


Figure 5.3 : Gradient Descent with large learning rate

If the learning rate parameter is a very small number, the gradient descent will be slow, as pictured in Figure 5.2. If the learning rate parameter is a very large number, the gradient descent can fail to converge as shown in Figure 5.3 and miss the optimal minimum loss value.

The value for the learning rate is set randomly by the user based on their past experiences. It is not easy to get the right value. This parameter affects the convergence slope of the model. Therefore, it is necessary to find the optimal value for it from starting to reach an optimal value with fewer epochs. This, in turn, reduces the training time of the model.

Even though finding the optimal value of the learning rate is a challenging problem, some well-researched approaches are available. Some of the popular techniques are explained in the section below.

5.4.1 Learning Rate Techniques

Decaying Learning Rate*:* With the increase in the number of epochs, the learning rate value decreases in this approach. The decrease rule is:

$$\alpha = \frac{\alpha_0}{1 + (\beta * E)} \tag{5.2}$$

Where α_0 is the initial learning rate, β is the decay rate, and E is the epoch number.

Variants of Decaying Learning Rate:

Exponential Decay: The learning rate decays exponentially throughout epoch time.

$$\alpha = \alpha_0 * \beta^E \tag{5.3}$$

Discrete Staircase: In this approach, the learning rate is decreased in specific discrete steps throughout epoch intervals.

Epoch Number consideration: In this approach, we apply a constant factor and divide it by the square root of the epoch.

$$\alpha = H * \frac{\alpha_0}{\sqrt{E}} \tag{5.4}$$

where H is the constant factor.

Mini-batch approach: It is similar to the above equation, except the mini-batch number is used instead of the epoch number.

$$\alpha = H * \frac{\alpha_0}{\sqrt{M}} \tag{5.5}$$

where M is the mini-batch number. This approach can be used only when a mini-batch gradient descent approach is employed.

Scheduled Drop Learning Rate: In this method, instead of updating the learning rate value in a monotonous fashion, it is decreased at a particular frequency or particular discrete proportional value.

The major limitation of the *scheduled drop learning rate* and the *decaying learning rate* is no evaluation mechanism. The learning rate value is decreased irrespective of the model's convergence to the optimal loss value for a particular cost function.

***Adaptive Learning Rate*:** In this method, the value of the learning rate is dependent on the gradient of the loss function. It will either increase or decrease. If the gradient value is higher, then the value for the learning rate will be smaller. If the gradient value is smaller, then the value for the learning rate will be higher. Thus, based on the curve of the cost function, the learning rate will either accelerates in shallow areas or decelerates in the steeper area.

***Cyclic Learning Rate*:** This method allows the training of the neural network with a learning rate that can be updated in a cyclic way rather than the non-cyclic way, which either decreases at every epoch or remains constant. The learning rate value oscillates between the predefined higher and lowers bound.

The following are the steps for cyclic learning rate:

1. Set the value for the lower bound of the learning rate called *base_lr*.

2. Set the value for the upper bound of the learning rate called *max_lr*.

3. Make the learning rate value goes back and forth between the *lower* and *upper* bound during training based on the increase and decrease in the gradient value of the cost function.

So, at first, the learning rate value will be very small. Then, over some time, it will grow until it reaches the maximum upper bound value. At this point, it will start reducing to a lower value until it hits the minimum base value. This cyclic pattern of increase and decrease continues throughout the training period.

5.5 Algorithm Analysis

In this section, we discuss the *brute force approach* of trying various learning rate values. This approach is naïve and needs lots of computational time.

Brute force approach

The brute force approach is described in Algorithm 1. It inputs the sequences of system calls, learning rate, and batch size value and outputs the trained model. It starts by dividing the dataset into batches with the specified batch size value. Next, for each batch size value in that interval range, we train the LSTM model with the input and output sequence of the corresponding batch. The model predicts the output sequence. Then, the cumulative loss is calculated for a single batch. The gradient is calculated based on this loss, and the learning rate is applied to the gradient. This will update the weight of the model parameter that the next batch of data will use. This whole process continues until the model converges to an optimal loss value.

Algorithm 1 Brute Force Algorithm

Input: Normal System Call Sequences, Batch Size, Learning Rate

Output: Trained Model

```
 1: $Train_{Data}$ = Preprocess the sequence into input and output sequence
 2: for i = 1 to BatchSize do
 3:     $Input_{Sequence}$ = $Train_{Data}$ [i][0]
 4:     $Output_{Sequence}$ = $Train_{Data}$ [i][1]
 5:     Model = LSTM model train on $Input_{Sequence}$ and $Output_{Sequence}$
 6:     $PredictedOutput_{Sequence}$ = Predicted sequence from Model
 7: end for
 8: $Cumulative_{Loss}$ = | $Output_{Sequence}$ – $PredictedOutput_{Sequence}$ |
 9: $\Delta\theta = -\eta * (\frac{\partial Cumulative_{Loss}}{\partial \theta})$
10: $\theta_{i+1} = \theta_{i+1} + \Delta\theta$
11: Repeat step 2 to 9 until for multiple epoch and until model converges
```

The major limitation of this approach is that the value for batch size and learning rate needs to be specified before the training of the model, and it remains constant through the training period. This limits the model to train faster at an earlier epoch.

5.6 Impact of Window Size

The preprocessing step of the model training requires data to be in specific input and output format. This input and output have a specific length which is called *window size*. For instance, if the window size is *5*, the model will take a sequence of *5* system calls as input and the following sequence of *5* system calls as output. We experimented with various window sizes, and Table shows the loss results achieved after running the algorithm for 100 epochs. We can depict that the lower the window size, the faster the model reaches the minimum loss.

Table 5.1: Loss with window size

WindowSize	**Loss**
3	0.29
5	**0.24**
10	0.45
15	0.67
20	0.94

5.7 Proposed Dynamic BatchSize with Dynamic Learning Rate Algorithm

Algorithm 2 Dynamic BatchSize and Learning Rate Algorithm

Input: Normal System Call Sequences, Initial Learning Rate

Output: Optimized Model

```
 1: S= Normal System Call Sequences
 2: N = Total length of Sequence
 3: $Dynamic_{Lr}$ = [ ]
 4: $LocalLr_{Loss}$ = [ ]
 5: $GlobalLr_{Loss}$ = [ ]
 6: J = 0
 7: $Remaining_{DataPoints}$ = N
 8: $BS$= BatchSize obtained after training model for first epoch through Algorithm
 9: for $i$ = 1 to 2 do
10:    $\eta$ =Initial Learning Rate
11:    $Dynamic_{Lr}$.append($\eta$)
12:    $Current_{BatchSet}$ = S [$j$ : $j$ + $BS$ + $1$]
13:    $Loss_{Value}$ = TrainOnBatch($Current_{BatchSet}$)
14:    $LocalLr_{Loss}$.append($Loss_{Value}$)
15:    $GlobalLr_{Loss}$.append($Loss_{Value}$)
16:    J = BS
17:    $Remaining_{DataPoints}$ = $Remaining_{DataPoints}$ - J
18:    $New_{Lr}$ = $\eta$
19:    $Current_{DataIndex}$ = 2 * J + 1
20: end for
21: $DeltaLr$ = [ ]
22: $Initial_{DeltaLr}$ = 0
23: while ($Current_{DataIndex}$ < N) do
```

```
24:  $Current_{BatchSize}$ = BS
25:  $Current_{Lr}$ = $New_{Lr}$
26:  $i$ = len($GlobalLr_{Loss}$) - 1
27:  $Current_{DeltaLr} = \frac{(GlobalLr_{Loss}[i-1] - GlobalLr_{Loss}[i])}{\max(|GlobalLr_{Loss}[i-1]|,\ |GlobalLr_{Loss}[i]|)}$
28:  $DeltaLr$.append($Current_{DeltaBS}$)
29:  $Initial_{Weight}$ = 0.50
30:  $DeltaLr_{MultiplyWeightList}$= [ ]
31:  for $k$ in range (len($DeltaLr$), 0, -1) do
32:      $DeltaLr_{Weight}$ = $Initial_{Weight}$ * $DeltaLr$ [$k$ - 1]
33:      $DeltaLr_{\text{WeightList}}$.append($DeltaLr_{Weight}$)
34:      $Initial_{Weight} = \frac{Initial_{Weight}}{2}$
35:  end for
36:  $TotalDeltaLr_{Weight}$ = $\sum DeltaLr_{\text{WeightList}}$
37:  $New_{Lr}$ = ceil($Current_{Lr}$ * (1 + $TotalDeltaLr_{Weight}$))
38:  $Dynamic_{Lr}$.append($New_{Lr}$)
39:  $Remaining_{DataPoints}$ = $Remaining_{DataPoints}$ - $New_{Lr}$
40:  $Current_{BatchSet}$ = S[$Current_{DataIndex}$ : $Current_{DataIndex}$ + $BS$ + 1]
41:  Values= TrainOnBatch($Current_{BatchSet}$, $New_{Lr}$)
42:  $LocalLr_{Loss}$.append(Values[0])
43:  $GlobalLr_{Loss}$.append(Values[0])
44:  $Current_{DataIndex}$ = $Current_{DataIndex}$ + $BS$
45: end while
Repeat step 23 to 45 for all even number of epoch until the model converges
```

The proposed dynamic algorithm works in the following way. It extends the dynamic batch size algorithm proposed in the previous chapter. Initially, the value for batch size is obtained from the *dynamic batch size algorithm* that trains the model using the specified learning rate for the first epoch. Next, the batch size value remains constant for the second epoch, and the learning rate is dynamically updated as follows. The learning rate is set according to the specified learning rate for the first two batches. This phase is called the *warmup phase*, where the learning rate is selectively constant. Now, the learning rate value is calculated *dynamically* starting from the third batch. It is computed as follows.

We compute the difference between the loss generated from the previous batch to the current batch. To normalize this loss difference, we divide it by the loss value, which is the maximum. Next, we apply the weight to each of the previous losses generated with the previous selection of the learning rate. As we go higher up to the previous losses, the weight is reduced to half. This is important because, with this approach, we enforce higher weightage to the current learning rate than that of the previous selection of the learning rate value. This weight value is multiplied by their corresponding loss value, and finally, they all are added to produce a single value. Now, to check whether to increase or decrease the learning rate value, we add the resultant sum value to 1. Furthermore, we multiply this additive term by the current learning rate value to get the final learning rate for the next iteration. This new learning rate is used to train the model for the next batch of the data—this way, the learning rate is selected dynamically. The above steps are repeated every time to calculate the learning rate value. Once we reach the end of the dataset, it is called one epoch. This process of dynamically updating the batch size and learning rate goes on alternate epochs consecutively until the model converges. This process reduces the training time sub-optimally.

5.8 Experimental Results

The following are results from training the LSTM model using the proposed dynamic learning rate with a dynamic batch size selection algorithm.

Table 5.2 shows the minimum loss achieved with all the datasets with the number of epochs needed to achieve minimum loss. The model is trained by keeping the default learning rate of 0.001.

Table 5.2: Loss with Default Learning Rate

Dataset	Minimum Loss	Total #Epoch
JMPS	0.0760	63
ADFA-LD	0.135	47
UNM	0.178	59

Furthermore, we experimented the proposed algorithm by changing the default learning rate value. Different learning rate experimented are *{1, 0.1, 0.01, 0.0001, 0.00001, 0.000001}*. We pick one learning rate value from the list and run the proposed algorithm each time. Table 5.3 shows the experimental results. For two of the dataset, the result improved compared to the default learning rate of 0.001.

Table 5.3: Loss with Different Learning Rate

Dataset	Best Learning Rate	Minimum Loss	Total #Epoch
JMPS	0.00001	0.0749	52
ADFA-LD	0.001	0.135	47
UNM	0.00001	0.177	53

In the proposed algorithm's case, we also found out that the learning rate of 0.00001 was optimal for JMPS and UNM datasets. Figure 5.4, 5.5 and 5.6 shows the decreasing loss value for JMPS, ADFA-LD, and UNM datasets, respectively, with the increasing number of epochs.

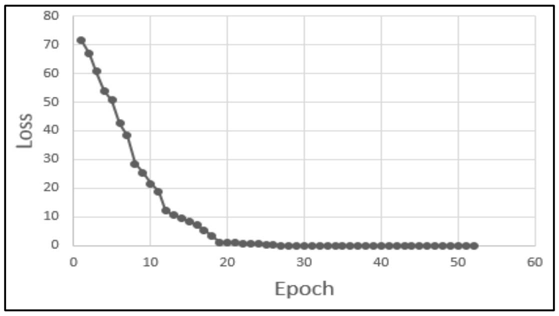


Figure 5.4: Dynamic BS_LR for JMPS

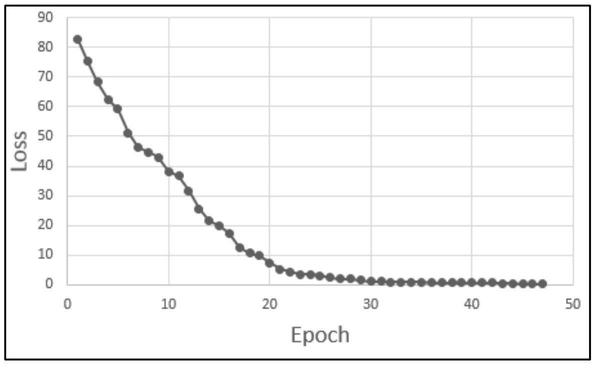


Figure 5.5: Dynamic BS_LR for ADFA-LD

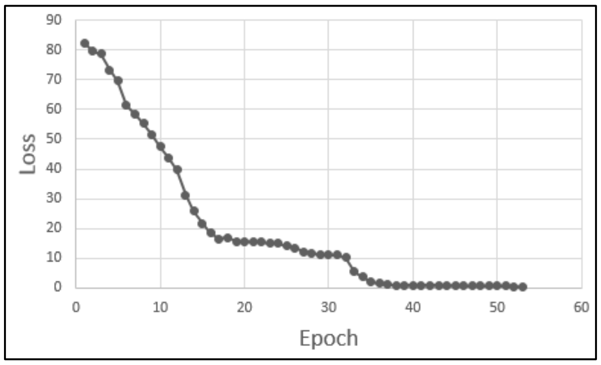


Figure 5.6 Dynamic BS_LR for UNM

For the JMPS dataset, the lowest loss achieved was *0.0749* at epoch *52* with a learning rate of *0.00001*.

For the ADFA-LD dataset, the lowest loss achieved was *0.135* at epoch *47* with a learning rate of *0.001*.

For the UNM dataset, the lowest loss achieved was *0.177* at epoch *53* with a learning rate of *0.00001*.

For the JMPS dataset, compared to dynamic batch size selection, the number of training epochs was reduced to *52*, which is *22* epoch less, with the loss reaching *0.0749*, which is *0.0017* higher.

For the ADFA-LD dataset, compared to dynamic batch size selection, the number of training epochs was reduced to *47*, which is *9* epoch less, with the loss reaching *0.135*, which is *0.007* higher.

For the UNM dataset, compared to dynamic batch size selection, the number of training epochs was reduced to *53*, which is *9* epochs less, with the loss reaching *0.177*, which is *0.004* higher.

5.9 Conclusion

Optimization of Deep Neural Networks using first-order algorithms has been researched in the literature. Stochastic Gradient Descent algorithms are one of the most popular ones. These algorithms need manual tuning and pre-specified values for most of the hyper-parameters. Such specification changes with the different datasets of a wide variety of domains and also with different neural architectures. Despite their wide usage, the generalization capability is still an open issue since the value of the parameters stays constant throughout the training period. This limits the model's capability to reach optimal minimum loss at a reduced epoch. In the previous chapter, we proposed a dynamic batch size selection approach. We extend that approach by incorporating the learning rate parameter, which is also tuned dynamically. Thus, we proposed a dynamic learning rate selection algorithm that dynamically updates the value of the learning rate. The update mechanism is based on the historical loss. The proposed approach updates the batch size and learning rate dynamically and consecutively. We experimented our proposed approach with three different datasets. ADFA-LD and UNM are open-source benchmark datasets. With experimental results, we found out that dynamically updating the batch size with the learning rate value trains the model at a faster epoch rate than the static approach. Furthermore, we notice that changing a default constant learning rate impacts the training time of the model.

CHAPTER 6

MACHINE LEARNING-BASED CYBER THREAT ANOMALY DETECTION IN VIRTUALIZED APPLICATION PROCESSES

Intrusion-based detection systems spot traces of abnormal activities focused on the network and connected resources. Anomaly-based detection systems analyze events of applications for abnormal behavior based on the hypothesis that anomalies signify an indication of malicious events. Host-based systems frequently depend on various attributes of a process to describe the normal behavior of any process. Multiple malicious vectors can be launched on a process with different characteristics to infect it. We propose a two-step approach for host-based anomaly detection. First, we analyze *ProcessList* data structure and create *Principal Component Analysis (PCA)* features known as E*igen traces* used for training multiple one-class anomaly detection models. These multiple models allow different attributes of process data to be assessed from numerous and diverse standpoints. As the anomaly scores of these models vary significantly, combining the scores to a single value is often challenging. Therefore, we apply a majority voting approach for the final anomaly score as the second step. This final score measures the occurrence of a malicious event. In this study, we demonstrate the implementation of the proposed two-step approach using four different one-class classifiers: *Mahalanobis Classifier*, *One-Class Support Vector Machine (OCSVM)*, *Isolation Forest,* and *Dendogram based Agglomerative Clustering*. We show that the proposed anomaly system improves the accuracy of anomaly detection.

6.1 Introduction

Intrusion-based Recognition Systems are a dynamic exploration arena in arrears to a strict and crucial requirement for cybersecurity methods in contradiction to continuously growing global outbreaks on computing organizations. Such attacks are characterized into two diverse types: *1) Known Attacks*: where signatures are available for each attack, and *2) Unknown Attacks*: Attacks which are never been identified before. It is more straightforward to combat known attacks since numerous orientations provide valuable data about their behavioral patterns. Nevertheless, no such signatures exist for unknown attacks; the only information we know about them is that they do not belong to a particular process's regular method of operations.

Recent techniques based on machine learning systems offer a variety of choices to analyze the incoming data for evaluation and show an efficient intrusion detection rate. We have two categories for Intrusion based detection systems: 1) Semi-supervised learning: Training with a labeled dataset for normal class 2) Unsupervised learning: Training with no labeled dataset. Numerous process characteristics make detecting anomalous attacks quite challenging [84]. Foremost is defining a meticulous and accurate borderline for the regular normal class. Most of the time, the usual behavior of a process frequently changes, and so as the malicious vectors. Today's process behavior often may not be the same as tomorrow's.

Data needs to be labeled in the Supervised classification type of machine learning system, whereas it is a crucial compelling problem for anomaly detection-based algorithms. The adversaries executing abnormal behavior on a particular system or process will try to acclimate themselves to scheme a behavior as if it fits regular action, making the detection process complicated. Additionally, in nearly a few circumstances, the irregular behavioral points nearby to the regular borderline will be measured as benign behavior.

The proposed idea is to build normal program behavior models by utilizing process-related data. The crucial reflection is the detail that a particular malicious vector interacts with the underlying system through process attributes to initiate damaging the system. An attack is presumed when the normal behavior of a known process diverges from the expected behavior.

With the ongoing comprehensive research in this area, developing an anomaly detection system is required to diminish false and missed alarms and simultaneously maximize the anomalous detection rate. Therefore, this study presents an unsupervised (one class) learning approach to the dataset extracted from windows processes. We collected the dataset during numerous regular operations and malware vector attack operations. We pre-process the extracted raw data with descriptive statistics and PCA's popular feature extraction technique. Furthermore, various learning-based algorithms, namely, Mahalanobis Classifier, OCSVM, Isolation Forest, and Dendogram-based Agglomerative Clustering, in conjunction with the dynamic thresholding, were used to detect the anomalous behavior. Using an ensemble of the proposed algorithms, we developed an anomaly detection system that tracks the particular applications to recognize malicious behavior.

The rest of the chapter is structured as follows. Section 2 provides the literature review on this work. Section 3 defines the in-depth overview of the proposed methodology to solve the anomaly detection problem. Section 4 discusses the experimental results, and finally, we conclude in section 5 with future work discussion in section 6.

6.2 Related Work

There are abundant methods testified in many of the literature works performed for abnormality detection. Tavallaee et al. [85] accumulated an all-inclusive review of studies on such systems. Out of *276* articles under consideration, they stated that *160* of them applied supervised classification-based machine learning algorithms, *62* of them projected methods developed on statistics, *36* of them utilized clustering-based algorithms, and the last 46 research papers have shown to use various fusion methods. Furthermore, the dataset was based on single host and network traffic data.

Deshpande et al. [86] projected an anomaly detection model based on system call traces for cloud computing settings that signal users of cloud systems in contrast to disturbances within their scheme. They used a framework based on a *Linux OS audit* and extracted data such as system calls and their frequencies with process IDs recorded in a log stored in the database. For all new trace of system calls, their detection system uses the Euclidean distance to relate it with the normal vectors. Their study achieved an accuracy rate of 96% in distinguishing abnormal events on the datasets used in [87] and [91].

Several weaknesses [87] can be exploited in software-defined networking (SDN), which offers numerous proficiencies in preventing and mitigating such attacks. Mahrach et al. [89] projected a technique to detect anomalies using the *SYN cookie* technique at the switch level. Their research examines a particular application's descriptive mean on the data collected throughout a specific period and prompts an alarm when the accrued capacity of extents surpasses a threshold.

Aghaei et al. [90] used SMOTE-based technique to overcome the imbalanced data problem created during the most frequent data patterns extraction. They employed an ensemble approach and developed two different classification models using different classification algorithms [48], namely Naive Bayes, Decision Trees, Random Forest, etc. The first classification model is the binary classification model [57] used to classify the attacks as normal or malicious. The second one is the multiclass classification model used to classify all six different types of attacks. Their study reports a *99.9%* detection rate for binary classification. The average accuracy score reported was only *55%* for all attack classes for the multiclass classification. Serpen et al. [91] used a windowed technique to generate fixed-size feature vectors. Furthermore, they used the features generated from principal component analysis to reduce the dimensionality. Next, they trained and developed the k-nearest neighbor algorithm to classify new test data.

6.3 Proposed Methodology

This section provides an in-depth overview of the proposed anomaly detection framework shown in Figure 6.1 with four stages: *Data Extraction*, *Data Pre-Processing*, *Detection Algorithm Training*, and Finally, *Deployment*.

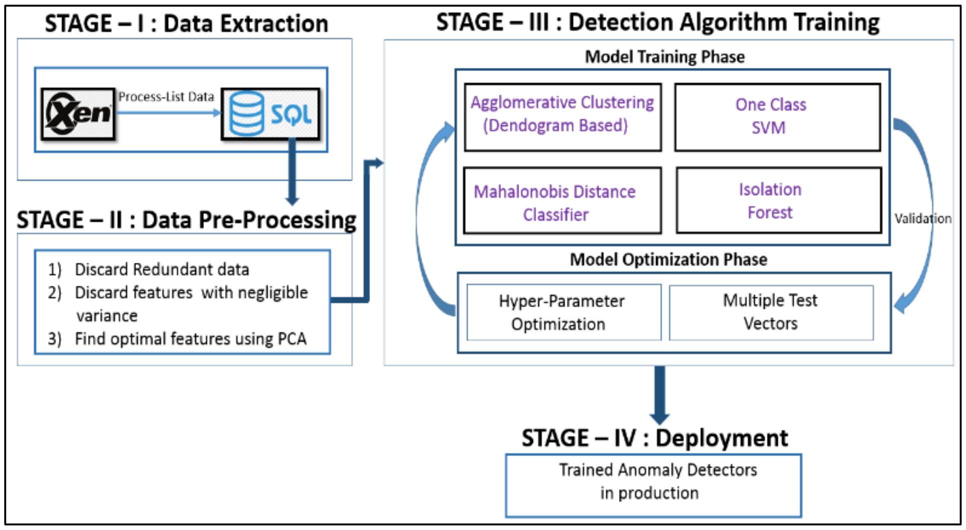

STAGE – I : Data Extraction
Xen
Process-List Data
SQL
STAGE – II : Data Pre-Processing
1) Discard Redundant data
2) Discard features with negligible variance
3) Find optimal features using PCA
STAGE – III : Detection Algorithm Training
Model Training Phase
Agglomerative Clustering (Dendogram Based)
One Class SVM
Mahalonobis Distance Classifier
Isolation Forest
Validation
Model Optimization Phase
Hyper-Parameter Optimization
Multiple Test Vectors
STAGE – IV : Deployment
Trained Anomaly Detectors in production

Figure 6.1: Proposed Anomaly Detection Framework

6.3.1 Stage-I: Data Extraction

Extraction of process list data is made with the help of the Virtual Machine Introspection (VMI) mechanism [92]. Here, the Xen hypervisor inspects the running processes in a virtual machine and extracts different features of a process as described in Table 6.1. The authors devised an architecture to remove process information, as shown in Figure 6.2. There are three significant components present in this architecture: User Interface, Virtual Machine Introspection Layer, and Data Analytics Layer. The user interface can be used to initiate the data extraction process with the help of user-defined application programming interfaces (API). The user interface starts and stops the data extraction process with pre-defined functionality. It can also view and manage virtual machines such as creation, deletion, and other activities. The virtual Machine Introspection Layer is the significant component of the architecture where the actual data extraction of the virtual machine occurs. It has different elements and performs several activities.

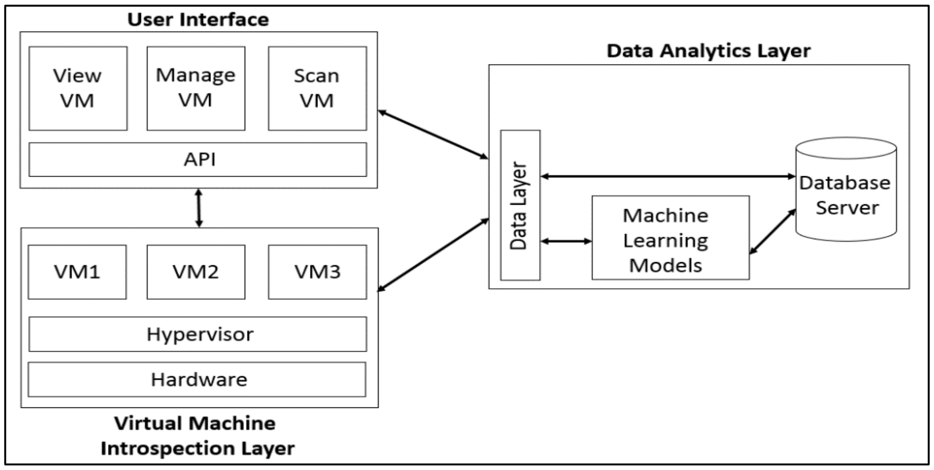


Figure 6.2: Architectural diagram of the Data extraction mechanism

The hypervisor will receive the command from the user interface. It uses the LibVMI library to extract memory offsets of the running processes and push the extracted data to the Data Analytics layer. In the Data Analytics layer, the extracted data is received at the data layer and stored in the Database server for further processing. The machine learning models module in the Data Analytics layer will create different models using different machine learning algorithms. Table 6.2 depicts the benign and malicious dataset samples, and Table 6.3 describes the extracted data set sizes.

Table 6.1: Extracted Features

No.	Features	Description
1	VMName	Name of the Virtual Machine
2	ProcessID	Unique Process Identifier Number
3	ProcessName	Name of the running process
4	NumberOfPrivatePages	Total number of private pages
5	NumberOfLockedPages	Total number of locked pages
6	ModifiedPageCount	Total number of pages modified
7	WorkingSetPage	Set of pages in the virtual address space
8	ActiveThreads	Number of Active Threads of a particular Process
9	ReadOperationCount	Count of read operations
10	WriteOperationCount	Count of write operations
11	OtherOperationCount	Count of I/O operations
12	ReadTransferCount	Amount of data read
13	WriteTransferCount	Amount of data written
14	OtherTransferCount	Amount of data transferred that are not read or written operations

15	KernelTime	Time in kernel mode, in 100 nanosecond units
16	UserTime	Time in user mode
17	BasePriority	Current base priority of a thread
18	DefaultPagePriority	Default Page value set during the process creation
19	DefaultIOPriority	Default IO value set during the process creation
20	StackCount	Define the size of the process stack space
21	PeakVirtualSize	Extreme virtual address space that can be used at any time
22	VirtualSize	Comprises the size of all pages that the process has reserved
23	DisableDynamicCode	When turned on, the process cannot generate dynamic code
24	DisallowStrippedImages	Rejects the reallocation information
25	DisallowWin32kSystemCalls	User mode calls that are disallowed to be serviced by win32k.sys
26	ActiveThreadsHighWatermark	Unused Stack Space
27	CommitCharge	The total amount of virtual memory to be backed by either physical memory
28	CommitChargePeak	The highest amount that the commit charge has reached
29	Cookie	Files with small pieces of data
30	CreateInterruptTime	The time spent by the processor servicing hardware interrupts
31	CreateUnbiasedInterruptTime	The time that the system is in the working state
32	DefaultHardErrorProcessing	Default value to process the error
33	DeviceAsid	Device ID
34	ExitStatus	Status of process Exit
35	Flags	An 8-bit field of 1-bit flags relating to structures in effect for the GPO (Group Policy Object)
36	Flags2	The computer configuration portion of GPO is disabled
37	Flags3	The GPO is disabled
38	ImagePathHash	The full path to the executable file corresponding to each process
39	LastAppStateUpdateTime	Last update time of the process state
40	LastFreezeInterruptTime	Interrupt time during process freeze
41	OwnerProcessID	The process ID of the owning thread
42	PriorityClass	The priority category for the associated process, from which the BasePriority of the process is calculated
43	ReadyTime	The time thread is waiting to use a processor
44	SectionSignatureLevel	The default required signature level for any modules that get loaded into the process
45	SequenceNumber	Track Process Sequence
46	SharedCommitCharge	Total of potential storage space required, which could be in either RAM or the page file
47	SignatureLevel	The validated signature level of the image present in the Image Name field
48	VadCount	It contains a detailed count of a process' allocated memory segments
49	LastThreadExitStatus	Exit status after the last thread has been terminated.

Table 6.2: Sample PS Dataset

		Sample Benign Rows			**Sample Malicious Rows**		
No.	**FeatureName**	**BR-1**	**BR-2**	**BR-3**	**MR-1**	**MR-2**	**MR-3**
1	VMName	VM	VM	VM	VM	VM	VM
2	ProcessID	1900	1900	1900	392	392	392
3	ProcessName	PName	PName	PName	PName	PName	PName
4	NumberOfPrivatePages	5063	4663	6219	11321	11105	8640
5	NumberOfLockedPages	0	0	0	0	0	0
6	ModifiedPageCount	381	377	424	516	495	462
7	WorkingSetPage	929852	929852	929852	51090	51090	51090
8	ActiveThreads	20	19	20	625	622	323
9	ReadOperationCount	0	0	0	0	0	0
10	WriteOperationCount	0	0	0	0	0	0
11	OtherOperationCount	0	0	0	17	17	0
12	ReadTransferCount	0	0	0	0	0	0
13	WriteTransferCount	0	0	0	0	0	0
14	OtherTransferCount	0	0	0	0	0	0
15	KernelTime	0	0	0	0	0	0
16	UserTime	0	0	0	1	1	0
17	BasePriority	8	8	8	8	8	8
18	DefaultPagePriority	10	10	10	10	10	10
19	DefaultIOPriority	4	8	4	4	4	4
20	StackCount	160	152	160	4976	4976	2584
21	PeakVirtualSize	799952896	795136000	799952896	3339759616	3339759616	2082299904
22	VirtualSize	799576064	795136000	798662656	3339759616	3339759616	2082299904
23	DisableDynamicCode	0	0	0	0	0	0
24	DisallowStrippedImages	0	0	0	0	0	0
25	DisallowWin32kSystemCalls	0	0	0	0	0	0
26	ActiveThreadsHighWatermark	20	19	20	622	622	323
27	CommitCharge	10735	10509	12107	20016	20016	16063
28	CommitChargePeak	10195	10635	12107	20016	20016	16063
29	Cookie	4006769173	4006769173	4006769173	118257043	118257043	118257043
30	CreateInterruptTime	888464747444	88846474744	88846474744	12526213484	1256213484	12526213484
31	CreateUnbiasedInterruptTime	888464747444	88846474744	88846474744	12526213484	12526213484	12526213484
32	DefaultHardErrorProcessing	1	1	1	1	1	1
33	DeviceAsid	0	0	0	0	0	0

34	ExitStatus	259	259	259	259	259	259
35	Flags	340626433	340626433	340626433	340626433	340626433	340626433
36	Flags2	33607700	33607700	33607700	33607700	33607700	33607700
37	Flags3	32	32	32	32	32	32
38	ImagePathHash	1554279663	1554279663	1554279663	1554279663	1554279663	1554279663
39	LastAppStateUpdateTime	88846474744	88846474744	88846474744	12526213484	12526213484	12526213484
40	LastFreezeInterruptTime	0	0	0	0	0	0
41	OwnerProcessID	0	0	0	0	0	0
42	PriorityClass	2	2	2	2	2	2
43	ReadyTime	0	0	0	0	0	0
44	SectionSignatureLevel	0	0	0	0	0	0
45	SequenceNumber	628	628	628	589	589	589
46	SharedCommitCharge	1227	1167	1227	1252	1252	1252
47	SignatureLevel	0	0	0	0	0	0
48	VadCount	192	195	194	1418	1418	809
49	LastThreadExitStatus	0	0	0	0	0	0

Table 6.3: Benign and Malicious Dataset Size

Dataset	**Projected Use**	**Class**	**Dimensions**
$\mathbf{D}_{train}$	Train	Normal	19500 * 49
$\mathbf{D}_{test}$	Test	Normal/Malicious	20960 * 49

6.3.2 Stage-II: Data Pre-Processing

The pre-processing of attributes of all processes (benign and malicious) is outlined in Algorithm 1, which comprises the following operations to clean the data:

i) Remove rows with zero variance

ii) Remove redundant rows

iii) Remove highly correlated features

iv) Normalize the dataset

First, the algorithm will create an empty PFD (Processed Feature Dataset). Next, it will calculate the variance of each column, remove the columns with zero variance, and remove the duplicate rows. Furthermore, it will calculate the Pearson correlation between every two pairs of columns and remove columns with correlation values greater than or equal to 95%. Finally, it normalizes the remaining columns and adds them to the PFD.

Zero Variance Features: The following attributes are removed due to zero variance.

NumberOfLockedPages, WorkingSetPage, ReadOperationCount, WriteOperationCount, ReadTransferCount, WriteTransferCount, OtherTransferCount, BasePriority, DefaultPagePriority, DefaultIOPriority, DisableDynamicCode, DisallowStrippedImages, DisallowWin32kSystemCalls, CreateInterruptTime, CreateUnbiasedInterruptTime, DeviceAsid, ExitStatus, Flags3, LastAppStateUpdateTime, LastFreezeInterruptTime, PriorityClass, ReadyTime, SectionSignatureLevel, SequenceNumber, SignatureLevel, LastThreadExitStatus

Highly Correlated Features: The following attributes are removed due to high correlation among themselves.

ModifiiedPageCount, OtherOperationCount, KernelTime, UserTime,Cookie, DefaultHardErrorProcessing, Flags, Flags2, ImagePathHash, OwnerProcessID.

Algorithm 1 Feature Processing Methodology

Input: Dataset (Δ) with benign process attributes, PFD: Empty Processed Feature Dataset
Output: Cleaned data

```
 1: for column $\Psi \in \Delta$ do
 2:     if ($\sigma(\Psi) != 0$ ) then
 3:         $PFD = PFD\ U\ \Psi$
 4:     end
 5: end for
 6: for row $\omega \in \Delta$ do
 7:     if ($\omega$ already exists) then
 8:         $PFD = PFD - \omega$
 9:     end
10: end for
11: for column $\Psi \in \Delta$ in range ($i$ to $n-1$) do
12:     for column $\Psi \in \Delta$ in range ( $j$ to $n$ ) do
```

```
13:         if (Pearson_Correlation ($\Psi_i$ , $\Psi_j$ ) >= 0.95) then
14:             $PFD = PFD - \Psi$
15:         end
16:     end for
17: end for
18: for column $\Psi \in \Delta$ do
19:     $\Psi new = \frac{\Psi i - \min(\Psi)}{max(\Psi) - \min(\Psi)}$
20:     $PFD = PFD \; U \; \Psi_{new} - \Psi$
21: end for
```

```
Algorithm 2 EigenTrace Methodology
Input: PFD from Algorithm1
Output: Reduce Dimensional Data
 1: for column Ψ ϵ PFD do
 2:     $\zeta$ = average trace vector $\Psi$
 3: end for
 4: $P = PFD - \zeta$
 5: $Q = PP^T$
 6: values, vectors = Eig(Q) //Compute Eigen Decomposition of the Covariance matrix Q
 7: if (all values are same) then
 8:     Data already in compressed form
 9: else
10:     Select N(Ψ) or less to compromise the chosen subspace
11:     PrincipalComponents = select (values, vectors)
12:     $P = B^T . A$      //Project Data into the subspace
13: end if
```

Final used attributes: We generate the principal components from the following attributes as described in Algorithm 2.

NumberOfPrivatePages, ActiveThreads, StackCount, PeakVirtualSize, VirtualSize, ActiveThreadsHighWatermark, CommitCharge, CommitChargePeak, SharedCommitCharge, VadCount.

The principal components generated using Algorithm 2 are used to train One-Class Classifiers.

6.3.3 Stage-III: Detection Algorithm Training

The following one-class classifiers are trained and optimized during this experimental stage to detect anomalies in our process list data structure.

Isolation Forest

The Isolation Forest algorithm [94] works based on two fundamental dataset principles: *A.* Over the total distribution of the dataset, the portion of the anomalous data points is low. *B.* The difference between the feature values of the anomalous data point and the normal data point is high. This algorithm uses *iTree,* a binary tree where each node has either zero or two children.

Let us consider M as a node. M can be a node with zero children or a node with two children nodes *(M_l, M_r)*. To build an isolation tree from a given sample data *D = {d1, ..., dn}*, we randomly select a feature *f* and split value *v* to divide *D* recursively until the following conditions occur: (i) Length of D is one, (ii) A particular height is reached (Defined as a hyperparameter) or (iii) D contains a constant value. Anomalies are detected according to their path length. The larger the path length, the higher the chance of an anomalous data point.

Path Length *p(d)* of a particular point *d* is the count of edges *d* must traverse to reach the end node from the root node.

An unsuccessful path length of any binary search tree is given as:

$$C(m) = 2G(m-1) - \frac{2(m-1)}{m} \tag{6.1}$$

where *G(i)* is the harmonic quantity.

F(m) is the average of *g(d)* and use to normalize *g(d)*. The anomaly *A* of an instance *d* is defined as:

$$A(d,m) = 2^{-\frac{\mathrm{Avg}(\mathrm{g(d)})}{\mathrm{F(m)}}} \tag{6.2}$$

where *Avg(g(d))* is the average of *g(d)*.

A = 0.5 when *Avg(g(d)) = F(m)*;

- *A = 1* when *Avg (g(d)) = 0*;
- *A = 0* when *Avg (g(d)) = m – 1*

A particular data point is considered anomalous based on the following criteria:

(a) if $A \approx 1$, then it is an anomaly,

(b) if $A << 0.5$, then it is considered a normal instance, and

(c) if $A \approx 0.5$, there is no distinct anomaly.

One-Class Support Vector Machine (SVM)

OCSVM [95] is a semi-supervised learning-based algorithm that learns a decision boundary to classify a point similar or dissimilar to the training data. It takes training data of one class (normal) as input. This algorithm was developed by Schölkopf et al. Given a training dataset $E = \{e_1,.., e_m\}$, $e_i \in R^d$. The algorithm tries to find a separating boundary with maximum distance. Mathematically, it is defined as follows:

$$\mathit{Arg\text{-}min}_{\mathrm{w},\zeta,\mathrm{p}} \frac{1}{2} ||\ \mathrm{b}\ ||^2 + \frac{1}{un} \sum_{i=1}^{m} \zeta i - o \tag{6.3}$$

where,

$$\big(w, \omega(x_i)\big) \geq \ \mathrm{o} - \zeta\mathrm{i} \text{ and } \zeta\mathrm{i} \geq 0 \tag{6.4}$$

Here m is the total training data points, and ω (•) is a non-linear kernel function. Furthermore, this algorithm uses normal vector b and offset o to learn the decision boundary. The degree of misclassification is calculated by the slack variable ζi. Finally, u, which lies between 0 and 1, determines which points are outside the decision boundary and inside the boundary.

Mahalanobis Classifier

The Mahalanobis classifier [97] is based on a statistical method that measures how far a particular data point is from a normal data points distribution cluster. Given a dataset with R datapoints and C features, the Mahalanobis distance M^2 is calculated as a function of $\bar{V}$ that comprises the mean of each feature and a covariance matrix E.

$$M^2\ (V_R) = (V_R - \bar{V})^T\, E^{-1}(V_R - \bar{V}) \tag{6.5}$$

where $\bar{V}$, E^{-1}, $and\ V_R$ transform each value of every feature column to a standard normal distribution by mean centering, scaling, and rotating, respectively, following a chi-squared distribution.

Dendogram based Agglomerative Clustering

Given N data points to be clustered, the basic process of Agglomerative Clustering [96] is as follows:

- Each data point is considered a cluster; therefore, there will be n clusters for n data points.
- Find the two closest clusters and join them into a single cluster.
- Continue clustering the data points until we reach a single cluster.

In Step 2, there are different ways to find the two closest clusters. They are:

- *Single-linkage*: Distance between two nearest data points of two clusters is calculated.
- *Complete-linkage*: Distance between two farthest data points of two clusters is calculated.
- *Average-linkage*: The two clusters' mean of the data points are computed and used to calculate the inter-cluster distance.

Custom Malware

We developed a custom malware that adds malicious values to the attributes of the process using the DLL injection method. It creates additional threads by creating a new file that hooks into the write system call of the process.

6.4 Experiments and Results

This section discusses the experimental results for all four one-class classifiers. All the classifiers are trained on the normal class of data, and the normal (benign) and anomalous scores are evaluated.

6.4.1 Mahalanobis Classifier

Mahalanobis Distance Metric distinguishes the malicious point from the normal point by calculating the Mahalanobis distance, an extended version of Euclidean distance. From Figure 6.3, we notice that when the normal data is tested again, the distance range is around *4000*, whereas, in the malicious data, the distance range is about *30000*.

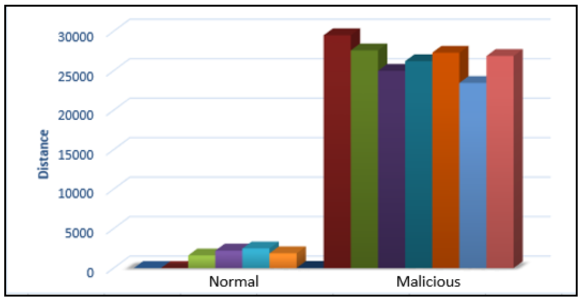


Figure 6.3: Mahalanobis Distance Metric Results

6.4.2 Isolation Forest

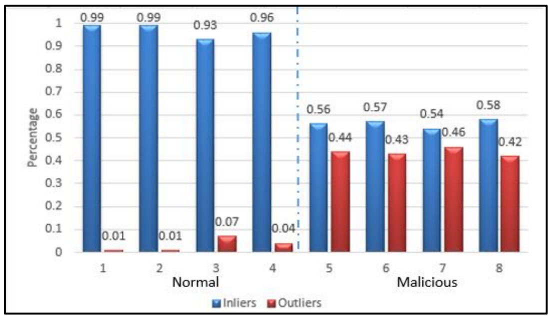


Figure 6.4: Isolation Forest Results

Isolation Forest is the version of the Random Forest train with only one class of data. As shown in Figure 6.4, each test case scores for inliers and outliers. Inliers are considered normal points, and outliers are considered anomalous points. When tested the normal data again, *93% to 99%* of data were classified as inliers, and *1% to 4%* were classified as outliers. Similarly, when tested against malicious data, *54% to 58%* of data were classified as inliers, and the rest, *42% to 46%* of data, were classified as outliers.

6.4.3 Agglomerative Clustering

Agglomerative clustering is a type of hierarchical clustering. We use the Euclidean distance and ward linkage method to generate the dendogram shown in Figures 6.5 and 6.6.

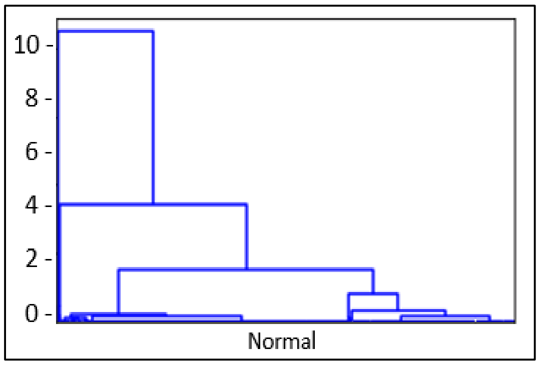


Figure 6.5: Agglomerative Clustering Results (Normal)

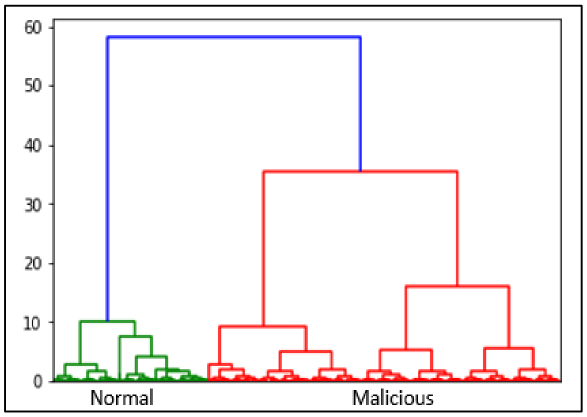


Figure 6.6: Agglomerative Clustering Results (Malicious)

The height value in the dendogram represents the distance between two clusters. As shown in Figure 6.5, when it is tested against the normal data, it shows only one cluster (normal) of data. Figure 6.6 shows the result when tested with malicious data, where we see two distinct clusters of data.

6.4.4 OCSVM

One class SVM is a type of SVM where we feed only one class (normal) of data during the training process. It finds the hyperplane that separates the far-away data points from the groups of data in a cluster. The results of OCSVM are shown in Figures 6.7 and Figure 6.8.

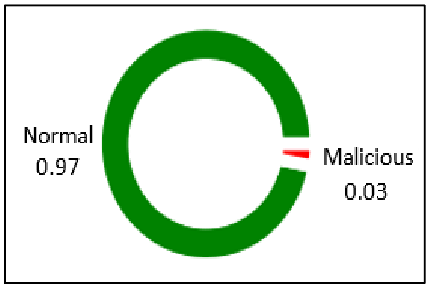


Figure 6.7: OCSVM Results (Normal)

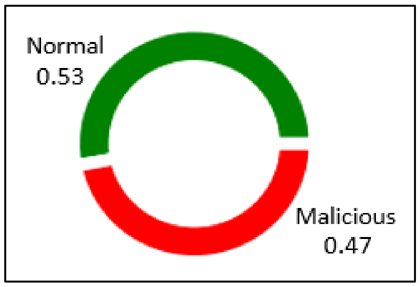


Figure 6.8: OCSVM Results (Malicious)

Figure 6.7 shows the results when trained OCSVM was tested against the normal data, giving a *97%* accuracy rate. Figure 6.8 shows the results when tested against the malicious data, and it was able to distinguish the normal data points and malicious data points.

6.5 Ensemble Approach

Table 6.4: Four Unique TestCase Results

Test Cases	Algorithm	Output Score	Threshold Value	Score Difference	Normalized Score Difference	Result	Ensemble Decision
1	Mahalanobis Classifier	0.93	0.70	+ 0.23	+0.76	Normal	Normal
	Agglomerative Clustering	0.97	0.80	+ 0.17	+0.85	Normal	
	Isolation Forest	0.92	0.80	+ 0.12	+0.60	Normal	
	One Class SVM	0.95	0.75	+ 0.20	+0.80	Normal	
2	Mahalanobis Classifier	0.45	0.70	- 0.25	-0.35	Compromised	Compromised
	Agglomerative Clustering	0.70	0.80	- 0.10	-0.12	Compromised	
	Isolation Forest	0.93	0.80	+ 0.13	+0.65	Normal	
	One Class SVM	0.80	0.75	+ 0.05	+0.20	Normal	
3	Mahalanobis Classifier	0.54	0.70	- 0.16	-0.22	Compromised	Compromised
	Agglomerative Clustering	0.62	0.80	- 0.18	-0.22	Compromised	
	Isolation Forest	0.85	0.80	+ 0.05	+0.25	Normal	
	One Class SVM	0.55	0.75	- 0.20	-0.26	Compromised	
4	Mahalanobis Classifier	0.42	0.70	- 0.28	-0.40	Compromised	Compromised
	Agglomerative Clustering	0.39	0.80	- 0.41	-0.51	Compromised	
	Isolation Forest	0.65	0.80	- 0.15	-0.18	Compromised	
	One Class SVM	0.40	0.75	- 0.35	-0.46	Compromised	

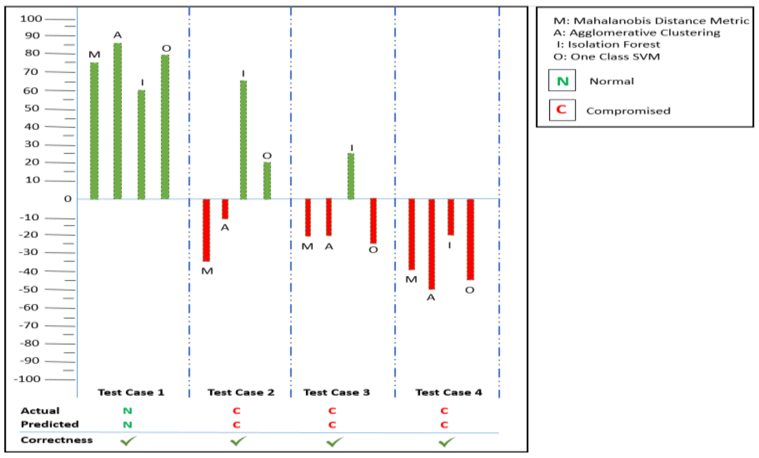


Figure 6.9: Ensemble Decision for each TestCase Scenario

The score difference value is calculated as below:

$$ScoreDifference = OutputScoreValue - Threshold_{Alg} \tag{6.6}$$

Since every algorithm has a different static threshold, we normalize the score difference between *-1.0 to +1.0*. This value is derived by dividing the score difference by the maximum possible score difference.

For $ScoreDifference$ >= 0 (i.e. Normal Cases),

$$NormalizedScoreDifference = \frac{ScoreDifference}{1 - ThresholdValue} \tag{6.7}$$

The characteristics of the test cases are as follows. *Test case 1* is the normal scenario where no malicious vectors were injected. All algorithms predicted this test case as normal. Thus the ensemble output is also normal. In *test case 2*, a single run of one malicious test vector was executed during the application process. In this case, two algorithms (Mahalanobis Distance and Agglomerative Clustering) predicted the outcome as compromised. Thus, the ensemble output is compromised since at least half of the algorithms yielded compromised predictions.

In *test case 3*, two runs of the same malicious test vector as in *test case 2* were executed during the application process. In this case, three algorithms (Mahalanobis Distance, Agglomerative Clustering, and OC-SVM) predicted the outcome as compromised. Thus, the ensemble output is compromised since at least half of the algorithms yielded compromised predictions.

In *test case 4*, two different malicious test vectors were executed during the application process. In this case, all four algorithms predicted the outcome as compromised. Thus, the ensemble output is compromised since at least half of the algorithms yielded compromised predictions.

6.6 Conclusion

One-class classification is primarily valuable for anomaly detection when data points of abnormal class are expensive to extract. As the behavior of an application process belongs to this category, we proposed a framework for anomaly detection in a process running on Xen hypervisor. This host-based approach analyzes various in-memory data structures of a process to classify its behavior as either normal or malicious. This framework utilizes the LibVMI library to extract the data and analyzes them in two stages. We pre-process the data with statistical approaches in the first stage. Then, PCA is performed in the same stage for dimensionality reduction of the data. This, in turn, reduces the training time of the algorithms. In the last part of the first stage, four different one-class classifiers, namely Mahalanobis Distance classifiers, Dendogram-based Agglomerative Clustering, Isolation Forest, and One-Class SVM, are trained on the normal class of data. These algorithms learn the normal behavior of a process during their training period. A unique static threshold is assigned to each algorithm. Each algorithm uses its threshold value to classify the behavior of the process.

The second stage of the framework applies the ensemble approach to the output scores from all the algorithms. A process is considered compromised if at least half of the algorithms determine its behavior as compromised. We presented the results of this ensemble approach for four different test cases and two different test vectors.

6.7 Future Work

In our current work, an anomaly is detected by the trained algorithms based on a static threshold relative to baseline results under the assumption that the workflow of the process remains unchanged. However, this static threshold may not be suitable for a baseline dataset of processes in a different environment. This can be overcome by employing a dynamic thresholding approach where the threshold is determined during the training phase for a given application. We plan to extend the current work by computing the dynamic threshold for anomaly detection irrespective of the underlying application environment.

The accuracy for the same baseline data varies across different algorithms since multiple algorithms provide a wide range of insights. Thus, it is essential to incorporate the results from multiple algorithms with an ensemble approach for anomaly detection. In this study, we presented four different one-class classifiers. We intend to add new algorithms such as local outlier factor, self-organizing Maps, and restricted Boltzmann machines for improved performance.

Each trained algorithm has been allocated the same weight with the assumption that all the models are equally skillful for the ensemble result. Majority voting combines the outcome from all the trained algorithms to generate the final anomaly detection score. To enrich this approach, we will modify the majority voting by applying probabilistic-based weights to each algorithm, wherein the algorithm that has performed the best during the training will be allocated a higher weight value.

CHAPTER 7

ADA-THRES: AN ADAPTIVE THRESHOLDING METHOD TO MITIGATE THE FALSE ALARMS

In a wide variety of domains, the advanced intrusion detection system consists of a learning-based detection method and a signature-based analysis approach. Such a system scans the incoming data, performs the analytics on it by using an anomaly detection algorithm, and finally transfers the report of suspicious activity for further analysis if found. The major problem of such a current system is the high false-positive rate (FPR), specifically in the case of a highly complex system with a large dataset. Such high FPRs, which are non-crucial, can easily overwhelm the user of the system and can further increase the likelihood of ignoring such indications. Therefore, mitigation approaches aim to develop a technique to reduce high FPR without losing any potential harmful threats.

Thus, in this chapter, we develop an adaptive thresholding algorithm that can mitigate the issue of high FPR. The proposed algorithm applies three scoring mechanisms. They are *Anomaly Pruning*, *Sequence Scoring*, and *Adaptive Thresholding*. The model is trained on sequential data. *Anomaly Pruning* gives a score to an individual data point. It either rejects or accepts the data points to be considered for Sequence Scoring. This *Sequence Scoring* will give a score to an individual sequence. Finally, an *Adaptive Thresholding* is applied to the cumulative score of all the sequences to detect the anomalous nature of the analyzed data. Multiple experiments have been conducted using various optimizers to access our proposed approach. Using the proposed approach, we train a deep learning-based LSTM algorithm widely adopted for sequential data. Furthermore, we validate it with three different datasets of various sizes. From the experimental results, we infer that the proposed approach allows the model to train faster and reach the minimum loss at a faster rate.

7.1 Introduction

Intrusion detection and Threat Detection are the cybersecurity systems that constantly take the huge volume of the multivariate dataset and perform anomaly detection and protect the systems. One of the prerequisites of any anomaly-based detection system is to learn the normal routine behavior of the process efficiently. Such a process can be heterogeneous and complex in nature. The trained model can thus better perform the estimation of the expected observations and can thus help in detecting abnormal behavior. Recent deep learning methods have shown substantial capabilities in predicting future observations.

An anomaly detection system has to accomplish numerous challenges: Methodology for scoring the observations and subsequently ranking them, a proper threshold to check the compromised state of the system, outlier removals, reducing the false positive rate, and finally, the explainability of the anomaly detection.

With the high uncertainties such as base rate [99], in detecting an anomaly, it is not easy to set the optimal value of threshold to identify particular observations as normal or anomalous. This remains true not only for single detectors but also for the multivariate and high dimensional streams of datasets. Even if the threshold is misadjusted slightly, it can generate a high number of false alarms or can miss the true anomalies [98, 101].

There is a major difference between anomalies and malicious events. Oftentimes, having high accuracy in anomaly detection does not guarantee the overall high detection of malicious events. The application of anomaly detection to discriminate the normal from abnormal can produce high false alarms since malicious observations are rare in nature.

7.2 Related Work

Anomaly Detection based on the prediction methods is solely focused on the prediction and not on the detection. Nonetheless, adjusting the threshold value to a specific range is vital. There is mainly two way to perform this calculation. They are parametric and non-parametric.

The data and its distribution must be known prior to the parametric-based techniques. Thus, the fixed thresholding approach is a big limitation in this approach. To overcome this issue, Clark et al. [100] introduce an approach that tries various threshold values and uses the one with the lowest false positive rate. Sequential data are basically a stream of data, and thus its data distribution can vary due to the concept drift problem. Therefore it is difficult to recognize anomalies from the normal change in the data distribution. Furthermore, they defined zero hypotheses to indicate that there is no concept drift problem in the current observation window, and thus the threshold is not required. Next, if the delta variation in the mean value of the anomaly score is detected by *Z-test*, then the threshold is updated. *P-value* scoring is employed in [29], which rejects the null hypothesis. A Gaussian distribution approach is utilized in [27]. The authors use the *maximum likelihood estimation* to generate the values of μ and σ using the vector generated by error calculation. Next, they employed a function to fit the error vectors into a normal distribution. A scoring method based on the double window is introduced by Ahmad et al. [28]. To detect the change in the short-term window and long-term window distribution, they maintain a history of the current short window and previous long window. Thus, using this approach, they solve the problem of concept drift in the data. There are some limitations in the parametric approaches. First, the underlying assumed distribution of data may not always be followed by anomalies. Due to this violation in statistical assumption, oftentimes, the residuals seem normal. Next, it is difficult to depend continuously on the hypothesis test since it assumes the absence of alternative hypotheses. Finally, the current world data is highly messy, multivariate, and high-dimensional in nature. These complex features are highly ignored by the parametric approaches and thus produce a high number of false alarms.

The assumption on the data distribution is not required in the non-parametric-based approaches. They usually take less computational power and are more popular than parametric-based methods. Such methods use the distance measure between the model and the test data and subsequently apply the threshold value on the calculated distance to detect whether an observation is an anomaly or not. Wang et al. assign an anomaly score to the test data by calculating its distance from the other groups. These methods are based on the machine learning algorithms, specifically, the supervised classification type, which brings an overhead for a real-time application. Furthermore, the computation of a threshold is always an issue in such a distance-based approach. Evaluation of residuals based on the unsupervised thresholding approach is presented by Hundman et al. [31]. However, their approach gives scoring to the whole sequence for anomaly detection.

7.3 Contribution of the work

We developed an *adaptive thresholding algorithm* that can tune adaptively based on the dataset and reduce the high false alarm rates. The proposed algorithm works in three steps:

Anomaly Pruning, Sequence Scoring, and Adaptive Thresholding. Anomaly Pruning rejects the irrelevant anomalies which increase the false alarms. Sequence Scoring provides the anomaly score for each window of the sequence. Finally, the cumulative score of all the sequence windows is calculated and compared with the Adaptive Thresholding to determine whether a test sequence is anomalous or not. We also developed a brute force algorithm to compare the efficacy of the developed algorithm. Three different datasets are being used to evaluate the proposed algorithm. Each dataset selected for the evaluation is of varying sizes.

7.4 Organization of the chapter

In section 7.5, we define and explain the problem formulation with the challenges of different types of anomaly. The static thresholding approach is described in section 7.6. The proposed adaptive tuning of the threshold with its subcomponents is explained in depth in section 7.7, with the discussion of the experimental results in section 7.8. Finally, we conclude in section 7.9.

7.5 Problem Formulation

User processes go into the kernel mode to interact with the operating system. A particular part of the code is compiled to execute the program. System calls are being made to execute various lines of code. Thus, any process's behavior can be studied by examining its sequence of system calls. These sequences will change if an attacker tries to manipulate the program. LSTM model is trained to learn such sequential data. It takes a particular window length of system calls as an input and predicts the next window of the system calls. In our previous chapters, we have proposed a dynamic approach for optimizing the training time of the algorithm. Thus, let us assume the LSTM model is trained and is now ready for detection. The model is trained with the normal sequences of the system, so it only knows the normal behavior of the processes. Anything that deviates from normal behavior is considered an anomaly. The problem is as follows. Given an unknown test sequence of a system call, check whether it is normal or anomalous. The process is executed in the following manner. Upon receiving the test sequence of the system calls, it is first transformed into a unique integer number. Next, this integer sequence is divided into batches of input and output of fixed length. Each input batch is given to the LSTM model to predict the output sequence. The predicted output sequence is compared with the actual output sequence, and even if one single system call is mispredicted, the whole input sequence is currently considered anomalous. This way, the count of the total anomalous sequence is computed, and a static threshold is applied to generate the final result.

The major limitation is the calculation of the threshold value. The static value needs to be tuned every time a new model is trained with new process data. Furthermore, this static approach often fails with the different varieties of malware. Since each malware behaves differently, the different static threshold has to be set for different malware. This is not feasible since there is a high surge in the complex malware generated every day. Also, such a static threshold creates too many false alarms and may sometimes miss the actual anomaly. Thus to avoid this problem, a dynamic threshold needs to be set.

7.5.1 Challenges

Anomaly Detection is the methodology of detecting the patterns or structure in the data that deviates from the normal pattern or behavior. Such patterns are often called an anomaly.

A straightforward method of detecting an anomaly is to define an area of the normal behavior of the process with the static threshold and deduce any data points or observations outside the range of the normal area as anomalous. This simple approach is not practical due to the following reasons:

1. Describing a normal area that can include all the possible behavior of the normal operation of the process is not easy. Furthermore, the precise boundary line that can discriminate between normal and malicious behavior is inaccurate. Therefore, some of the malicious data points near the decision boundary can actually be normal data points.

2. The adversaries always try to make the malicious data points in such a way that it looks like the normal point of observation. Therefore, the main goal of defining normal behavior is tough.

3. Today's real-world applications are highly complex in nature with multivariate features. Furthermore, they keep evolving with the increasing trends. Thus, the current representation of the normal behavior may not be an accurate and precise representation of the future version of the applications.

4. For numerous domains with a wide variety of applications, the precise definition of the anomaly is difficult to develop. For instance, even a nominal deviation from the normal

operation in the medical field will be considered an anomaly. In contrast, the same fluctuation in the domain of the stock market can be viewed as normal. Therefore, a methodology proposed and created for a particular domain may not be sufficient and easy to apply to another domain set.

5. There is always a critical issue in the availability of the training, validation, and testing dataset in the form of the label as being normal or anomalous for the model development purpose.

6. Many times, some of the observations of the dataset are noisy, which seems to be an actual anomalous point, and henceforth, such noisy points are challenging to identify and discard.

7.5.2 Types of Anomaly

In this section, we discuss the different types of anomalies in the sequential data. Comprehensively, the anomalies can be classified into three distinct categories:

Point Anomalies: A particular data point can be considered an anomaly if it deviates largely compared to the remaining data observations of the dataset. It is one of the simplest forms of detection of an anomaly.

Contextual Anomalies: If a particular data instance is anomalous with respect to a unique context; then, it is called contextual anomalies. Behavioral and Contextual are two types of features considered to detect such anomalies. Behavioral features represent the non-contextual form of the observations of the dataset. The anomalous behavior is computed by selecting the different values of the behavioral features pertaining to the distinct context.

A particular observation might be considered an anomaly in a specific context but may be considered a normal point in some other context. On the other hand, the contextual features denote some form of context or neighbor of detecting an anomaly. This is one of the important characteristics in differentiating the behavioral features from the contextual features.

Collective Anomalies: The set of observations that are similar to each other but different from the rest of the data are considered collective anomalies. The individual observations may not be malicious on their own, but their presence in the collective anomalies makes it an anomaly.

In the next section, we discuss the implementation of the static threshold and its applicability for anomaly detection.

7.6 Static Threshold Approach

Figure 7.1 depicts the overall framework for detecting the test sequence as anomalous or normal. The step-by-step procedure is described in Algorithm 1. It takes as an input the system call sequences of base data as well as the test data. Next, upon the transformation of the sequence to its corresponding numeric format, the sequence of both the dataset are converted into input and output sequences. The input and output sequence of the base data is used for training the model. The loss of the trained model is used to generate the threshold value. Now, we fed the input sequence of the test data to the trained model. The model generates the predicted output sequence, compared with the test output sequence, and the error is calculated. If the error is less than the static threshold, then the test sequence is considered normal else, anomalous.

Algorithm 1 Static Threshold Algorithm

Input: Base and Test System Call Sequences

Output: Probability of the anomalous

```
 1: $Base_{InputBatch}$ = $Base_{InputSequence}$
 2: $Base_{OutputBatch}$ = $Base_{OutputSequence}$
 3: LSTM = Train the model with the $Base_{InputBatch}$ and $Base_{OutputBatch}$
 4: $Static_{Threshold}$ = Loss of the LSTM Model
 5: $Test_{InputBatch}$ = $Test_{InputSequence}$
 6: $Test_{OutputBatch}$ = $Test_{OutputSequence}$
 7: $Anomalous_{Count}$ = 0
 8: for i = 0 to Totalbatches do
 9:     $Predicted_{OutputBatch}$ = OutputBatch Predicted by the LSTM model
10:     Error = $|Predicted_{OutputBatch}|$ - $|Test_{OutputBatch}|$
11:     if Error! = 0 then
12:         $Anomalous_{Count}$ += 1
13:     end if
14: end for
15: if $Anomalous_{Count}$ > $Static_{Threshold}$ then
16:     Test SystemCallSequence is Anomalous
17: else
18:     Test SystemCallSequence is Normal
19: end if
```

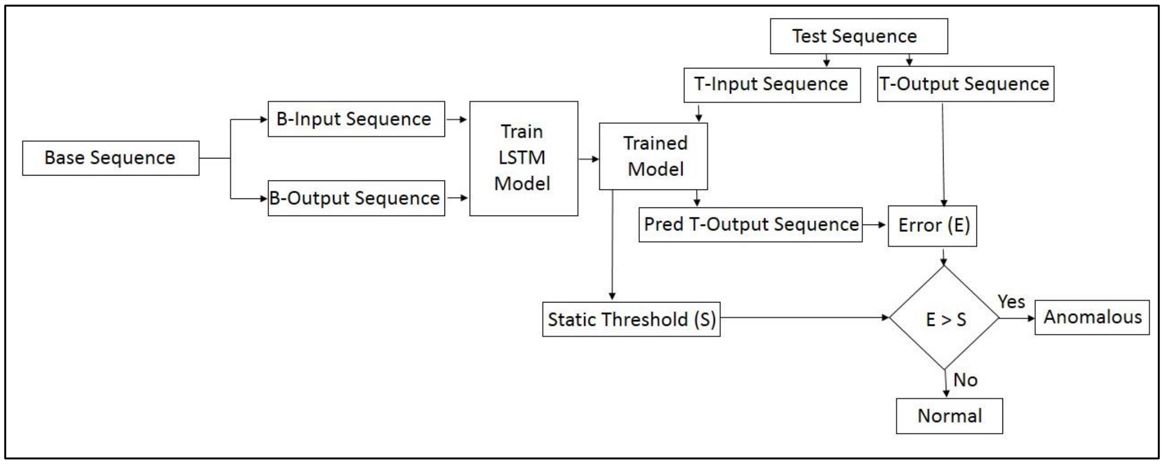


Figure 7.1: Static Threshold Approach

The major limitation of this approach is non-adaptiveness, where a new static threshold has to be set for a new process each time. Additionally, it induces high false alarm rates since it gives a binary score to each sequence and generates the total cumulative score of all the test data sequences.

To overcome this problem, we propose an adaptive thresholding approach that not only can just be applied to any process but reduces the false alarms rate by pruning the irrelevant error points.

7.7 Proposed Adaptive Thresholding Algorithm

In this section, we discuss the implementation of the proposed algorithm that can adaptively tune the threshold value, which helps in reducing the overall false alarm rates.

Algorithm 1 Adaptive Thresholding Algorithm

```
Input: Normal System Call Sequences
Output: E, P, DT
 1: $IO_{Batch}$ = Input and Output Batches of Sequences
 2: for sequence in $IO_{Batch}$ do
 3:     $Predicted_{OutputBatch}$ = Output from LSTM Model
 4:     $Raw_{Difference} = |Predicted_{OutputBatch}| - |IO_{Batch}|$
 5:     $NormalizedRaw_{Difference} = \frac{|\bar{X}-X|}{X}$
 6:     $Error_{Sum} = \sum Errors$
 7:     $Error_{Count} = \sum Count\ of\ NonZero\ Errors$
 8: end for
 9: $Epoch\ Non\ Zero\ Avg\ Pair\ Error = \frac{Epoch\ Total\ pair\ Error\ Values}{Epoch\ Count\ of\ Pair\ Errors}$
10: $Epoch\ Pair\ Error = \frac{Epoch\ Count\ of\ Pair\ Errors}{\sum Sequences}$
11: Continue step from 2 to 10 for multiple epochs until the model converges
```

The proposed adaptive thresholding algorithm is depicted in the Figure 7.2. This algorithm computes the threshold, which is adaptive to any process. It has three main subcomponents:

We calculate the Anomaly Pruning, Sequence Scoring, and Adaptive Threshold Value during the training phase. This section is divided into *Training* and *Testing* Phase. We apply these scores to evaluate the test data during the testing phase.

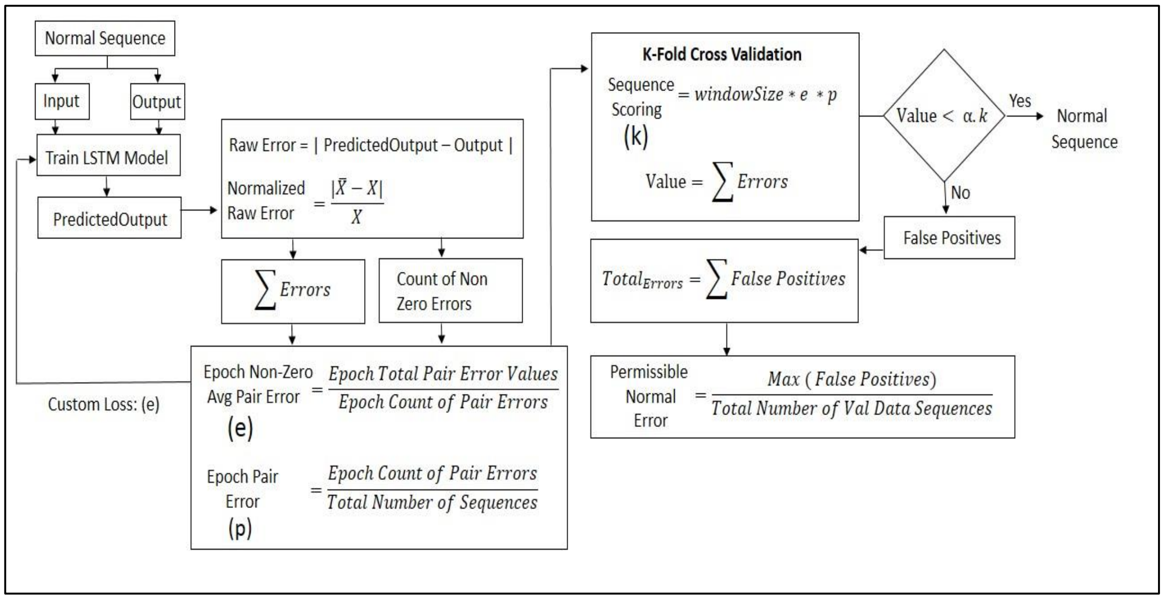
7.7.1 Training Phase

The three components mentioned above are calculated during the training of the LSTM algorithm.

Anomaly Pruning: Score used to discard the irrelevant anomalies.

Sequence Scoring: A score is given to each sequence of the batch.

Adaptive Threshold: Score used at an Epoch Level for the whole dataset.



Normal Sequence
Input
Output
Train LSTM Model
PredictedOutput
Raw Error = | PredictedOutput – Output |
Normalized Raw Error $= \frac{|\bar{X} - X|}{X}$
$\sum Errors$
Count of Non Zero Errors
Epoch Non-Zero Avg Pair Error (e) $= \frac{Epoch\ Total\ Pair\ Error\ Values}{Epoch\ Count\ of\ Pair\ Errors}$
Epoch Pair Error (p) $= \frac{Epoch\ Count\ of\ Pair\ Errors}{Total\ Number\ of\ Sequences}$
Custom Loss: (e)
K-Fold Cross Validation
Sequence Scoring (k) $= windowSize * e * p$
Value $= \sum Errors$
Value $< \alpha.k$
Yes
Normal Sequence
No
False Positives
$Total_{Errors} = \sum False\ Positives$
Permissible Normal Error $= \frac{Max\ (False\ Positives)}{Total\ Number\ of\ Val\ Data\ Sequences}$

Figure 7.2: Adaptive Threshold Training Phase

It takes a sequence of system calls of the normal processes as input. The sequence is transformed into the numeric format. Next, it is divided into Input and Output batches of sequences. The LSTM model is trained with these Input and Output batches. Each time, a single Input sequence is fed to the model to predict the corresponding output sequence. The predicted sequence is compared one on one with the actual output sequence. Furthermore, the normalized score is calculated as shown in equation 7.1 to normalize the error on the particular sequence.

$$NormalizedRaw_{Difference} = \frac{|\bar{X} - X|}{X} \tag{7.1}$$

X is the actual output value, and $\bar{X}$ is the predicted output value.

Using this, we compute two different scores.

1. Sum of the Errors

2. The total count of the non-zero errors

The above two scores are calculated for each batch of the data and finally at each epoch level. Next, we develop the custom loss *e* from the above score, which the LSTM model will use to calculate the gradient.

$$Epoch\ Non - Zero\ Avg\ Pair\ Error\ \text{(e)} = \frac{Epoch\ Total\ pair\ Error\ Values}{Epoch\ Count\ of\ Pair\ Errors} \tag{7.2}$$

$$Epoch\ Pair\ Error\ \text{(p)} = \frac{Epoch\ Count\ of\ Pair\ Errors}{\sum Sequences} \tag{7.3}$$

We use the *K-Fold Cross Validation technique* to get the optimal value of *e* and *p*. For experimental purposes, we set the value of *K as 10*. Each time nine parts of the data are used for training, and the tenth part of the data is used for validation. The score *k* to the sequence is given with the below equation:

$$Sequence\ Scoring\ (k) = windowSize * e * p \tag{7.4}$$

Next, we calculate the total error made by the model. If that error is less than $\alpha * k$, then the sequence is considered normal else, the model considers it anomalous, which in this case is viewed as False Positives.

$$Total_{Errors} = \sum False\ Postives \tag{7.5}$$

We calculate the False Positives of all the sequences of the validation dataset and compute the total error made by the model on the current validation dataset. This step is repeated for each set of data. With k being *10*, we have ten different errors computed. Thus, using this value, we finally compute the adaptive threshold as below:

$$Permissible\ Normal\ Error = \frac{MAX\ (False\ Positives)}{\sum ValSequence} \tag{7.6}$$

This Permissible Normal Error is the adaptive threshold value.

7.7.2 Testing Phase

During the testing phase, the unknown test data, which contains the sequence of system calls, is transformed and divided into the input and output sequence of the batches. Anomaly Pruning Score (e), Sequence Scoring (SS), and Final Anomaly Score are used to obtain the final resultant value, as shown in Figure 7.3. Initially, the trained LSTM model receives the input sequence, predicting the output sequence, which is compared with the test output sequence. Next, the normalized raw error is calculated, and if any individual error in that normalized score is less than the Anomaly Pruning Score, it is considered an anomaly.

Next, we count the total number of errors TE made in a particular sequence and is discarded based on the equation below.

$$TE > WindowSize * e \tag{7.7}$$

We use the equation below to get the total number of Anomalous Sequences.

$$\text{T} = \frac{\sum TS}{\sum TestDataSequences} \tag{7.8}$$

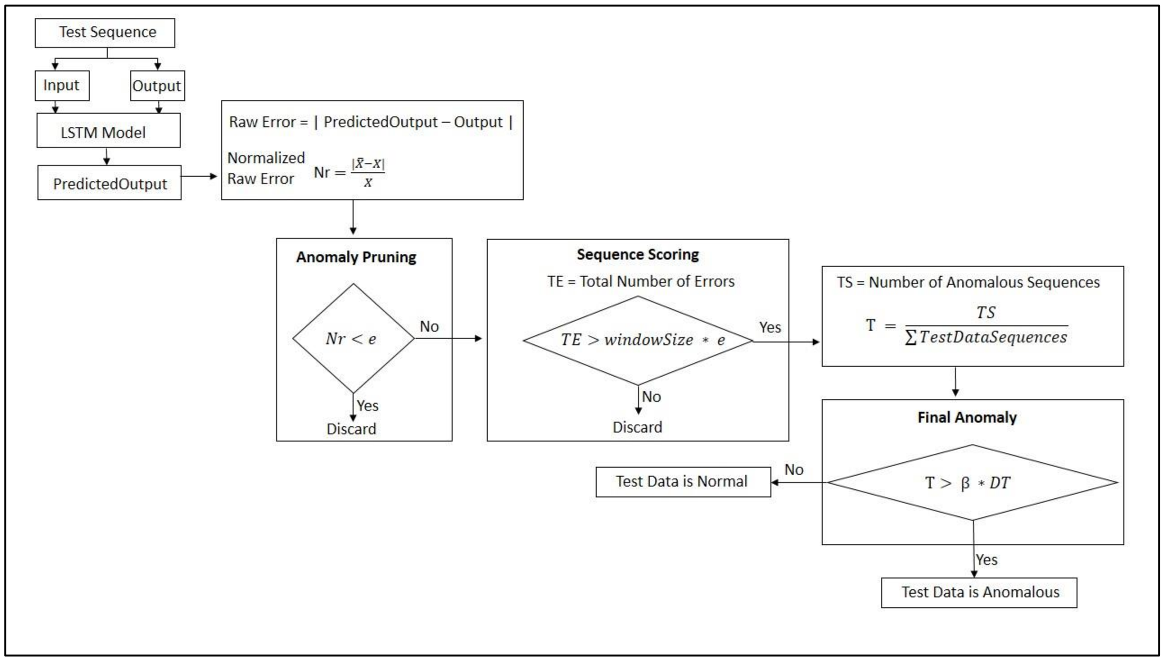


Figure 7.3: Adaptive Threshold Testing Phase

Finally, if $T > \beta * DT$, then the test data is considered anomalous else, it is observed as normal.

Here, α and β are the hyper-parameters of the proposed algorithm.

7.8 Experimental Results

This section discusses the results obtained by applying the proposed algorithm to three sequential datasets: JMPS, ADFA-LD, and UNM.

Table 7.1: FPR for JMPS Dataset

Algorithm	**False Positive Rate**
Static Approach	12%
Adaptive Thresholding Algorithm	**3.1%**

For the JMPS dataset, we experimented with both the static approach and the proposed adaptive thresholding approach. For the static approach, we had to manually tune the threshold to achieve the lowest FPR of *12%* as shown in Table 7.1. However, with the Adaptive thresholding algorithm, the FPR is drastically reduced to *3.1%*. The main improvement is pruning the individual anomaly, scoring the input/output sequence, and comparing it with the permissible error.

Table 7.2: FPR for ADFA-LD Dataset

Algorithm	**False Positive Rate**
Decision-Based Engine [102]	23%
OCSVM [103]	20%
Semantic Approach [104]	4.2%
EWR [105]	2.4%
Adaptive Thresholding Algorithm	**0.7%**

Table 7.2 shows the comparison results of the Adaptive Thresholding algorithm with the other detection methods for ADFA-LD dataset. In [103], the frequencies-based approach is used to classify the system call sequences of the test data. In their approach, they perform various empirical tests to set the window size value. This results in unstable comparative

performance since the sequence length varies with different datasets. In [102], the authors extract the dataset's most uncommon and common subsequence with their *min* and *max* number of system calls. However, sequential information is not utilized in their approach. The proposed adaptive thresholding approach gives the least false positive rate of *0.7%*.

Table 7.3: FPR for UNM Dataset

Algorithm	**False Positive Rate**
Dynamic Methodology [106]	19%
Probabilistic Approach [107]	14%
EWR [105]	10%
VMGaurd[108]	6.2%
Adaptive Thresholding Algorithm	**1.2%**

Table 7.3 shows the comparison results of the Adaptive Thresholding algorithm with the other detection methods for UNM dataset. In [106], the authors train various machine learning-based algorithms and achieve the lowest FPR of *19%*. The major drawback of this approach is that they converted the sequences into vectors of frequencies during the preprocessing steps. Thus the contextual semantics of the temporal part is not taken into consideration. The statistical based approach, namely maximum likelihood estimation, is utilized by Srinivasan et al. [107]. They use the *n-gram* approach to convert the sequence of system calls into meaningful features. The main limitation of this approach is the value of *n*. With the increase in the value of *n*, their proposed model tends to give a higher false-positive rate. VMGaurd method was developed by Mishra et al. [108]. They employed the *Term Frequency-Inverse Document Frequency method* to extract the dataset's features. Next, a machine learning-based random forest classifier is trained on those features. The major limitation of their approach is that they required the labeled dataset for the normal and attack test vectors. Our proposed approach requires only the normal dataset for training

the model. Based on the comparative analysis, our proposed adaptive thresholding approach provides the least FPR of *1.2%*.

7.9 Conclusion

With the increase in complex and multivalued malware attacks, detection has become challenging. Also, the real-world dataset is messy and highly imbalanced, where most of the dataset belongs to the normal class. Thus the classification-based approach is not appropriate in such a scenario. Therefore, an Anomaly Detection System is developed, which trains the model with the only normal data class. The model learns the behavior of the data and decides the decision boundary with some threshold. Anything that deviates from the decision boundary is considered anomalous. The major limitation of the current approach is the manual setting of the threshold. It has the following two issues: high false-positive rate and threshold need to be changed for every new process data. To solve this problem, we propose an adaptive threshold algorithm. Based on Anomaly pruning, sequence scoring, and final adaptive thresholding components, we trained and validated the model that generates a low false-positive rate. We evaluated our proposed approach on JMPS, ADFA-LD, and UNM datasets. ADFA-LD and UNM are open-source benchmark datasets. We can conclude that our proposed approach results in a low false-positive rate for all three datasets based on the experimental results.

CHAPTER 8

EA-NET: A HYBRID AND ENSEMBLE MULTI-LEVEL APPROACH FOR ROBUST ANOMALY DETECTION

In the current world, the applications of anomaly detection range from fraud detection to diagnosis in the medical area. Most of the current methodologies are applicable only when a particular dataset pertains to certain assumptions and a distinct domain. Such assumptions require prior knowledge of the dataset. The training development cycle time to find the best single model is time-consuming and challenging. Unsupervised anomaly detection methods do not use the target label for training. However, they result in high false positive rates.

In this chapter, we address the problem of the ensemble anomaly detection approach that generalizes well across multiple domains. We design a multi-level hybrid approach. At the First Level, we train several weak classifiers (weak one class classifiers). Next, we utilize deep learning-based AutoEncoder to reduce the dimension of the dataset. These are the two sets of hybrid features. Next, different one-class classifiers have their strength and limitations. Thus, we propose an adaptive weightage approach that gives the weight to each classifier. Next, this input is passed to the second level. At this level, we have a deep neural network that learns the patterns of the dataset and generates an adaptive dynamic threshold to discriminate the input feature as an anomaly or benign. The major benefit of this approach is the *reduced training time* and *low false-positive rate*. The training time is reduced due to the reduction of the input feature dimensions at the first level.

8.1 Introduction

Anomaly Detection refers to the methodology of finding the data observations that deviate from the expected normal patterns or behavior of data. Developing an efficient anomaly detection solution is always a challenging task, even with the recent surge in the development of learning-based algorithms. Most of the prior work conveys that the usage of supervised-based machine learning algorithms can only recognize the anomalies available in the dataset used for training the model. Nonetheless, any observation that diverges from the expected behavior has been termed an irregularity. Therefore, such irregularities may not be similar to those already available in the dataset [109]. Secondly, different detection-based techniques rely on diverse and distinct rules in the dataset. Often such algorithms are specific to a particular domain application. Thus, detecting anomalies from across the multiple domains and in a wide variety of scenarios by a single model is challenging [110]. Simply training multiple one-class classifiers iteratively with different hyper-parameter optimization techniques is a time-consuming task. Furthermore, the anomaly detection approach based on traditional methods often requires features that are processed and engineered in a particular manner. This requires a high amount of computational power and memory. Deep learning-based anomaly detection algorithms [111] have computed higher efficiency to address the abovementioned challenges. Nonetheless, their approach requires the data to be in a particular distribution, and also, the developed methodology lacks the generalizability across multiple domains. Thus, in this work, we propose a *hybrid multi-level ensemble anomaly detection* that learns to combine the predictions from multiple one-class classifiers and trains a deep neural network that gives the final probability of the observation as being normal or anomalous.

8.2 Literature Review

Based on the availability of the data, the anomaly detection approach is divided into three main categories: supervised, semi-supervised and unsupervised. The supervised-based approach trains the model on binary/multiclass data. It is not used widely for anomaly detection due to the lack of class imbalance problem and lack of training data [84]. The unsupervised approach detects abnormalities based solely on the normal class of data. The conventional approach includes support vector machines [112] and data descriptors [113]. Such algorithms assume data to be normal. The major limitations of traditional approaches are: that the outcome is highly sensitive to the complex hyper-parameters. The trained model cannot be extended to the multiclass dataset. The clustering approach is utilized in [114, 115]. The limitations of these approaches are high computational time, and the results are biased towards the static threshold value. Deep learning-based AutoEncoder is trained, which generates the reconstruction error. This error is used to compute the anomaly score [116]. Compared to traditional approaches, anomaly detection algorithms based on deep learning have shown high results in extracting the complex feature representations of the data. Scalability is one of the advantages of such an approach. Recently, a hybrid approach is being implemented where authors in [117] use autoencoder to learn the latent space of high dimensional complex dataset. This learned latent space is given as input to the one-class classifiers for anomaly detection. It combines the feature extraction capability of the neural network with the discriminative capabilities of the one-class classifiers. The limitation of this approach is to rely solely on the AutoEncoder for feature extraction. To overcome this problem, we enhance the approach that not only uses the AE for feature extraction but also several weak one-class classifiers. This results in low false-positive rates.

8.3 Contribution of the work

We develop a hybrid and multi-level ensemble anomaly detection framework. At the first level, we reduce the feature dimensionality of the dataset. These features are hybrid since we train multiple one-class classifiers and an AutoEncoder model. Such features have *high information gain and low entropy value*. Different one-class classifiers have different characteristics. Thus, we apply weightage to each of these weak classifiers. Next, we use these features at the second level to train a deep neural network that outputs the anomaly score. Here, we propose an *adaptive threshold approach* to decide the boundary. The proposed framework has a low false-positive rate and trains the model at reduce computational time.

8.4 Organization of the chapter

The rest of the chapter is structured in the following ways. In section 8.5, we explain various one-class classifiers with dimensionality reduction techniques. Information theory is explained in section 8.6. Open Source benchmark datasets used for evaluation are described in section 8.7. The proposed hybrid multi-level ensemble anomaly detection framework and algorithm are briefly explained in section 8.8. Next, we discuss the experimental results in section 8.9 with the conclusion in section 8.10.

8.5 Learning Based Algorithms

8.5.1 One Class Classifiers

One class classification algorithms uses the only normal class of data. Thus, it learns the normal behavior of the application, and anything that deviates will be considered an anomaly. In this section, we will discuss one class classifiers such as *Elliptical Envelop* and *Local Outlier Factor*. We already explained *Mahalanobis Distance, One-Class Support Vector Machine*, and *Isolation Forest* in chapter 6.

Elliptical Envelope Method

This method extends the statistical $\mu \pm \sigma$ approach for high dimensional feature vector. It calculates the covariance matrix of the multi-dimensional gaussian distributed dataset. This is achieved by transforming the dataset into an elliptical format. Thus, those observational points far away from the transformed elliptical format are considered anomalous.

Here, the distance between a data point and its distribution of the model is computed using the following equation:

$$Distance_{\mu,\zeta}\,(x_i)^2 = (x_i - \mu)^T \zeta^{-1} (x_i - \mu) \tag{8.1}$$

Where $\mu \; and \; \zeta$ are the specified attribute and the covariance of the Gaussian data.

To compute the covariance matrix of a high dimensional dataset, the minimum covariance determinant estimator is employed, up to $\frac{(p-k-1)}{2}$, where $p \; and \; k$ are the integer numbers representing the total number of samples and the total number of variables, respectively. MCD algorithm is used to compute the standard estimates.

Local Outlier Factor

This algorithm calculates the deviation of a particular data point's local density to its neighboring data points. An observation is considered an anomaly if its density is lower than that of neighbors.

It computes the density relative to individual data points.

A-Distance: The total distance between a point and it's A neighbor.

Reachability Distance: It is calculated as a maximum of the A-Distance of the neighboring points using the below equation.

$$LRD_A(m) = \frac{N_A(m)}{\sum_{o \epsilon N_A(m)} d_A(m,o)} \tag{8.2}$$

Now, the following steps are used to compute the LOF score.

1. For each data observation, its A-nearest neighbors are computed.

2. Local Reachability Density (LRD) is computed by calculating the local density of a data point.

3. Finally, the LOF score is generated by comparing LRD with the LRD of A-neighbors.

$$LOF(m) = \frac{\sum_{o \epsilon N_A(m)} \frac{LRD_k(o)}{LRD_A(m)}}{|N_A(m)|} \tag{8.3}$$

This calculation makes LOF an efficient anomaly detection algorithm for the high-dimensional dataset with a substantial imbalance of target labels.

8.5.2 Dimensionality Reduction

Current world datasets are very high-dimensional in nature. The total training time of the algorithm increases substantially with the increase in the feature set of the dataset. Thus, this section discusses the two most popular dimensionality reduction techniques.

Principal Component Analysis (PCA)

PCA decreases the total number of features while preserving vital information.

The following steps are used by PCA

Standardization: The continuous variables are transformed into a standardized format so that every feature is on the same scale and thus can contribute evenly during analysis.

Covariance Matrix Calculation: The main goal of this step is to determine how different features of the dataset vary from the mean to check if any relationship exists between them.

Eigen Values and Eigen Vector Calculation: In this step, the covariance matrix from step 2 calculates the eigenvalue and eigenvector to generate the final principal components. These components explain the maximum amount of the variance of the dataset but with less number of features. The first component will have the highest amount of information, and the second component will have the second-highest variance of the information, and so on. The interpretability of these components is less critical since they do not have any semantic meaning associated with them.

AutoEncoder

AutoEncoder is a deep learning-based unsupervised algorithm that learns to compress the extensive feature data into a compressed space. Next, it tries to reconstruct the original data from the compressed space. The error between the original and reconstructed data is termed a reconstruction error. This algorithm has the following four steps:

Encoder: The model learns to reduce the representation of the dataset.

Latent Space: A layer that stores the compressed representation.

Decoder: The model learns to reconstruct the data from the compressed space.

Reconstruction Error: It is the error between the original and reconstructed input. The less the error, the better the model learned the data.

To minimize the reconstruction error, the backpropagation algorithm is employed. Following are the hyper-parameters of this model that needs to be tuned.

Latent Size: Total number of nodes used to compress the data. The smaller the size, the compact the latent space.

The number of Hidden Layers: Autoencoder can be deep and thus have multiple hidden layers.

Number of Nodes: The total number of nodes per hidden layer shrinks from the input layer to the latent space, and it grows back to the original value at the output layer.

8.6 Information Theory

Information is nothing but anything that can enhance our knowledge in understanding the system or process. It is the reverse version of uncertainty. The more confident we are in a particular area, the better our understanding is and vice-versa. This is the fundamental principle of information theory. We are more uncertain when we have less knowledge about it. Therefore, with proper information, we reduce the uncertainty.

The entropy is defined as function E of probabilities$(R_1, R_2, \ldots., R_n)$. To quantify the uncertainty, it has to satisfy three conditions:

- E is continuous. It defines that if any R_i fluctuates, the uncertainty will not change much.
- The uncertainty will increase with the increase in *n* if all the probabilities have equal value.
- Finally, the decomposition of the probabilities as a sum of weighted uncertainties will always result in the same.

The only function that satisfies all three conditions is the Shannon Entropy.

$$\mathrm{E}(R_1, R_2, \ldots., R_n) = -J \sum_{i=1}^{n} R_i \log(R_i) \tag{8.4}$$

Where *J* is a constant number.

The amount of information gained can be calculated based on the entropy as follows. Let us assume that W is the random variable that models our current representation of knowledge before we learn a particular information d. Our updated knowledge will be thus $(K \mid d)$. Thus the information gain is:

$$I(d) = E(W) - E(K \mid d) \tag{8.5}$$

8.7 Datasets

The following three open-source benchmark datasets are used for experiment purposes. Each dataset is unique and has a varying size of feature set.

CIC-IDS2017 Dataset

It is one of the intrusion detection datasets released in 2017. There are a total of 2.8 million records with 79 features. This dataset is generated by Canadian Institute for CyberSecurity. It is generated over a period of five days. This dataset contains information on real-world network traffic, which include the normal traces and the malicious traces in the PCAP format.

UNSW-NB15 Dataset

This dataset is developed using the IXIA PerfectStorm tool. It was created in the Australian Center for Cyber Security(ACCS) lab. It has a total of two million records with 44 features. The dataset is a hybrid that captures the real-world scenario of normal activities. On the other hand, it captures the synthetic attack behavior of the network traffic. There are nine different types of attacks recorded in this dataset.

NSL-KDD Dataset

This dataset is an improved version of the KDD Cup 99 dataset. Each data observation is labeled as a normal or malicious class of network data. There are a total of five classes of data. They are Probing, Remote to User, User to Root, Denial of Service, and Normal. There are 41 features (discrete and continuous) with 125K training data and 22K testing data.

8.8 Proposed Ensemble Anomaly Detection Algorithm

In this section, we explain the proposed Ensemble Anomaly Detection Algorithm. The pictorial view is depicted in Figure 8.1. It comprises two levels.

1. Hybrid Feature Extraction

2. Anomaly Detector

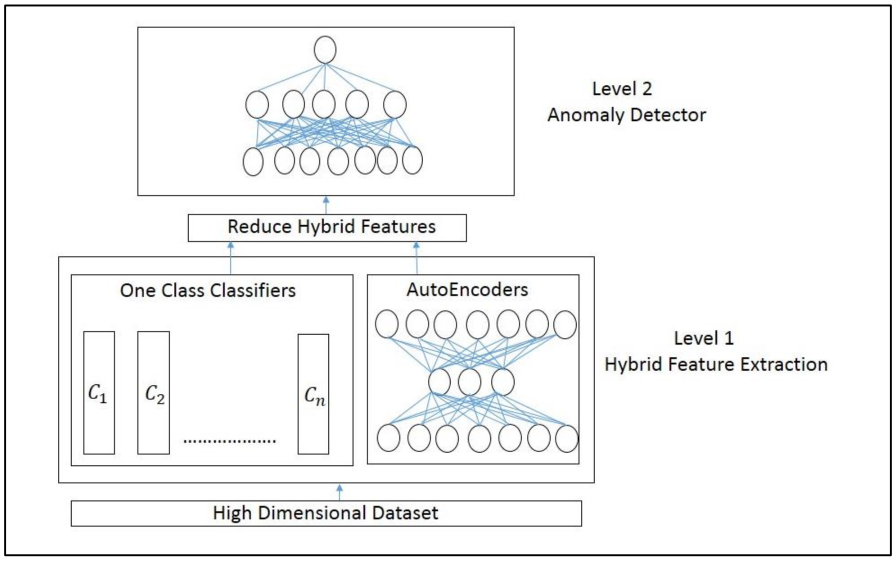


Figure 8.1: Ensemble Anomaly Framework

8.8.1 Hybrid Feature Extraction

In this component, we train multiple one-class classifiers: *OCSVM, Isolation Forest, Mahalanobis Classifier, Local Outlier Factor, and Elliptical Envelope*. Each one class classifier has its unique characteristics. Thus, we apply an adaptive weightage to each of these algorithms. The workflow for calculating the weightage is described in Figure 8.2.

We apply the K-Fold cross-validation technique where the value of K is set to 10. Each time, we calculate the total number of False Positives produced by the algorithm, and the cumulative error is generated.

$$Total_{FP} = \frac{FP_i + FP_{i+1} + \ \dots + FP_k}{k \ * \ \sum Val\ Data} \tag{8.6}$$

Now, based on the above equation, we calculate the weight of each of the classifiers as follows:

$$Weight_{Classifier} = 1 - \ Total_{FP} \tag{8.7}$$

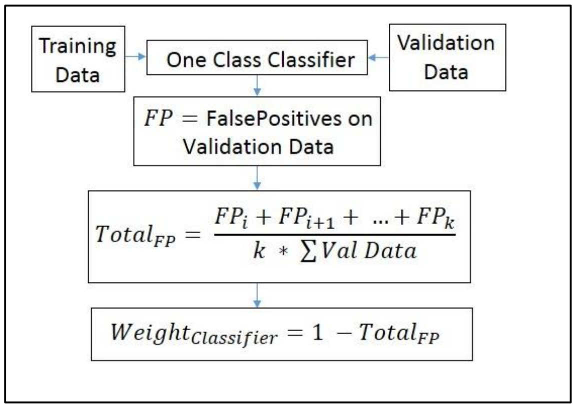
Next, we train deep learning-based AutoEncoder to reduce the dimensionality of the dataset to a smaller latent space. This algorithm takes as input the feature set and will reduce it to a lower dimension.

Algorithm 1 Ensemble Anomaly Algorithm

```
Input: DataSet
Output: Normal or Anomalous Data Points
  1: N = Number of Rows
  2: $Classifier_{Output}$ = Train multiple One Class Classifiers and Generate Prediction
  3: $FP$ = False Positives on the $Validation_{Data}$
  4: $Total_{FP} = \frac{FP_i+FP_{i+1}+ \dots+FP_k}{k * \sum Val\ Data}$
  5: $Weight_{Classifier} = 1 - \ Total_{FP}$
  6: $Weighted_{Features} = Classifier_{Output} * \ Weight_{Classifier}$
```

```
 7: $AE_{Output}$ = Output from trained AutoEncoder
 8: $Combined_{Features}$ = $Weighted_{Features}$ U $AE_{Output}$
 9: DNN = Trained Neural Net on $Combined_{Features}$
10: for i in range 0 to N do
11:     $Output_{DNN}$ = Prediction using DNN for $Data_i$
12:     if $Output_{DNN}$ > $Adaptive_{Threshold}$ then
13:         Data point is anomalous
14:     else
15:         Data point is normal
16:     end if
17: end for
```

Next, it will reconstruct the original feature from the compressed space. The error in reconstruction is the loss. The backpropagation algorithm is applied to update the weight and reduce the loss. Thus, these two distinct sets of features are then fed to Anomaly Detector.



Training Data
One Class Classifier
Validation Data
FP = FalsePositives on Validation Data
$Total_{FP} = \frac{FP_i + FP_{i+1} + \ldots + FP_k}{k * \sum Val\ Data}$
$Weight_{Classifier} = 1 - Total_{FP}$

Figure 8.2: Weight Mechanism

We achieve the following three benefits from these hybrid features:

High information gain, Low Entropy, and Reduced Dimensionality

To validate this, we calculate the information gain through Shannon entropy.

8.8.2 Anomaly Detector

This is the second level of the proposed framework. It will take as input the hybrid features generated from *Level 1*. Next, it will train the deep neural network and output the probability of observation as normal or anomalous. Here, we use K-Fold Cross Validation to generate the value for Dynamic Threshold.

8.9 Experimental Results Analysis

This section discusses the results obtained by applying the proposed algorithm to three intrusion detection datasets: CIC-IDS2017, UNSW-NB15, and NSL-KDD.

Table 8.1 shows the comparison results of the Proposed Ensemble Anomaly Detection algorithm with the other detection methods for the CIC-IDS2017 dataset.

Table 8.1: Metrics for CIC-IDS2017 Dataset

Technique	**False Positive Rate**
Consolidated J-48 [119]	6.64
LIBSVM [120]	5.13
FURIA [121]	3.16
WiSarD [118]	2.86
DT-Rule [122]	1.14
Proposed Approach	**0.56**

The authors [119] applied different resampling strategies to train the classification-based machine learning algorithms. Their approach is based on the class distribution of the training dataset. FURIA [121], authors proposed a novel fuzzy rule-based method for classification purposes. The model learns the fuzzy rules instead of traditional rules, which are often based on conventional unordered sets. LIBSVM [120] applies quadratic minimization to the traditional SVM algorithm. WiSarD [118] transform the data into patterns of the *n-tuple* recognizer and further trains the model by passing tuples as input. DT-Rule [122] framework proposed by Ahmed et al. trains an ensemble of JRip, Forest PA, and REP tree. Most of the traditional approaches are based on binary classification. Our proposed ensemble anomaly approach provides the least FPR of *0.56%* based on the comparative analysis.

Table 8.2 shows the comparison results of the Proposed Ensemble Anomaly Detection algorithm with the other detection methods for the UNSW-NB15 dataset.

Table 8.2: Metrics for UNSW-NB15 Dataset

Technique	**False Positive Rate**
E-Max [123]	23.79
Two-level Classification [124]	15.64
Stack Ensemble [125]	8.90
GBM [126]	8.60
Proposed Approach	**4.37**

The performance result of our proposed approach has shown a considerable improvement compared to the existing works. E-Max [123] uses statistical analysis for ranking the attributes and then uses features correlation techniques. They finally trained five different classification algorithms. Two-level classification [124] is employed by Zong et al. They train the model to detect the majority and minority classes of the dataset. Two-level Ensemble is proposed in [125], where authors developed a feature selection method and ensemble of two-level classification. Gradient Boosting Classifier is trained by Tama et al. [126] with grid search optimization techniques. The major limitation of this approach is the training time due to the high complexity of optimizing the hyper-parameters. Our proposed ensemble anomaly approach provides the least FPR of *4.37%* based on the comparative analysis.

Table 8.3 shows the comparison results of the Proposed Ensemble Anomaly Detection algorithm with the other detection methods for the NSL-KDD dataset.

Table 8.3: Metrics for NSL-KDD Dataset

Technique	**False Positive Rate**
SVM [127]	15.0
GAR [128]	12.2
TDTC [129]	5.56
TwoLevelEnsemble [130]	2.52
Proposed Approach	**1.09**

Pervez et al. [127] trained the Support Vector Machine algorithm, merging different feature selection techniques. They solve the binary class classification problem. Kanakarajan et al. [128] utilize a meta-heuristic approach that enables the trained forest-based algorithm to reach the optimal global value. To increase the ensemble diversity, they applied the annealed randomness procedure. TDTC [129] uses a linear discriminant and component analysis approach to reduce the feature set of the data and further train the k-nearest neighbor algorithm. Two Level Ensemble was proposed in [130], where they train an ensemble of multiple weak classifiers and further perform the statistical significance test. The major drawback of these approaches are that the weak classifiers are trained on binary classification dataset, and thus their approaches fail to detect any new unknown malware data observation. Our proposed approach trains the model on the normal dataset and applies the dynamic threshold. Thus, anything that deviates from normal behavior will be considered an anomaly. Our proposed ensemble anomaly approach provides the least FPR of *1.09%* based on the comparative analysis.

8.10 Conclusion

This study explores anomaly detection for various highly imbalanced classes of the dataset. Binary class and Multiclass are less efficient in detecting the new anomaly since they are trained on the labeled dataset. Currently, various one-class classifiers have been developed, which take as input the normal class of the dataset and learn the normal behavior of the dataset. Anything that deviates from the normal decision boundary is considered an anomaly. Each one class classifier has its characteristics. Thus, training only one algorithm is not efficient for the highly complex real-world dataset with high dimensionality. Therefore, we propose a hybrid two-level anomaly detection framework in this study. At the first level, we train several one-class classifiers and an AutoEncoder algorithm. Next, we apply the weight to each one class classifiers algorithm. These reduced feature sets will be passed to the second level. We also calculate the information gain of the reduced features. The second level trains a deep neural network that outputs the probability value for the normal and anomalous points. We evaluated our proposed approach on open-source benchmark CIC-IDS2017, UNSW-NB15, and NSL-KDD datasets. Our proposed approach results in a low false-positive rate compared to the previous work for all three datasets based on the experimental results.

CHAPTER 9

CONCLUSION

This dissertation focuses on the specific approach to model and analyze the sequential data with the high dimensional feature set. Specifically, we proposed a framework and algorithms for performing anomaly detection.

The first contribution is enhancing the behavior modeling of the system calls for anomaly detection. We analyzed two types of behavior: *Temporal* and *Non-Temporal* behavior. We trained sequential deep learning-based algorithms for temporal behavior, namely Long Short Term Memory (LSTM). For training the model, only a normal class of data was supplied. Next, the non-temporal behavior is analyzed independently by evaluating frequency and commonality behavior. We applied Cosine Similarity to analyze frequency behavior and Jaccard Similarity to analyze the commonality behavior. We propose an extension called Point-Bag of System Calls (P-BoSC), a Natural Language Processing-based technique to detect the anomaly and output the anomalous window.

The second contribution is providing a hybrid algorithm, Dynamic Batch Size and Learning Rate (DynaB-LR) that tunes the batch size and learning rate dynamically and consecutively. It is fast and can be applied to any time series-based dataset. At the same time, it accounts for the learning algorithm obtaining the optimal value of these hyper-parameters without the need for the user to manually input the number, unlike most of the other established algorithms. We achieve three significant benefits: *Reduced training time, Optimized memory usage*, and *Loss reduction* at an early epoch stage.

The third contribution is developing an adaptive thresholding algorithm that can mitigate the issue of high FPR. The proposed algorithm applies three scoring mechanisms. They are *Anomaly Pruning*, *Sequence Scoring*, and *Adaptive Thresholding*. The model is trained on sequential data. *Anomaly Pruning* gives a score to an individual data point. It either rejects or accepts the data points to be considered for Sequence Scoring. This *Sequence Scoring* will give a score to an individual sequence. Finally, an *Adaptive Thresholding* is applied to the cumulative score of all the sequences to detect the anomalous nature of the analyzed data.

The third contribution is designing a multi-level hybrid ensemble anomaly detection approach. At the First Level, we train several weak classifiers (weak one class classifiers). Next, we utilize deep learning-based AutoEncoder to reduce the dimension of the dataset. These are the two sets of hybrid features. Next, different one-class classifiers have their strength and limitations. Thus, we propose an adaptive weightage approach that gives the weight to each classifier. Next, this input is passed to the second level. At this level, we have a deep neural network that learns the patterns of the dataset and generates an adaptive dynamic threshold to discriminate the input feature as an anomaly or benign. The significant benefit of this approach is the *reduced training time* and *high anomaly detection rate*.

Additional improvements can be made, including the explainability and interpretability of the deep learning algorithms and support of online learning in the anomaly detection area.

BIBLIOGRAPHY

[1] AV-TEST Security Report. (accessed: May 05th, 2022) https://www.av-test.org/fileadmin/pdf/security_ report/AV-TEST_Security_Report.pdf.

[2] McGraw, Gary, and Greg Morrisett. "Attacking malicious code: A report to the infosec research council." *IEEE software* 17, no. 5 (2000): 33-41.

[3] Gryaznov, Dmitry. "Scanners of the year 2000: Heuristics." In Proceedings of the 5th International Virus Bulletin, vol. 113. 1999.

[4] Arp, Daniel, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and C. E. R. T. Siemens. "Drebin: Effective and explainable detection of android malware in your pocket." In *Ndss*, vol. 14, pp. 23-26. 2014.

[5] Kolosnjaji, Bojan, Apostolis Zarras, George Webster, and Claudia Eckert. "Deep learning for classification of malware system call sequences." In *Australasian joint conference on artificial intelligence*, pp. 137-149. Springer, Cham, 2016.

[6] Miller, Brad, Alex Kantchelian, Michael Carl Tschantz, Sadia Afroz, Rekha Bachwani, Riyaz Faizullabhoy, Ling Huang et al. "Reviewer integration and performance measurement for malware detection." In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 122-141. Springer, Cham, 2016.

[7] Nissim, Nir, Aviad Cohen, Robert Moskovitch, Assaf Shabtai, Mattan Edry, Oren Bar-Ad, and Yuval Elovici. "Alpd: Active learning framework for enhancing the detection of malicious pdf files." In *2014 IEEE Joint Intelligence and Security Informatics Conference*, pp. 91-98. IEEE, 2014.

[8] Forrest, Stephanie, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. "A sense of self for unix processes." In *Proceedings 1996 IEEE Symposium on Security and Privacy*, pp. 120-128. IEEE, 1996.

[9] Čeponis, Dainius, and Nikolaj Goranin. "Investigation of dual-flow deep learning models LSTM-FCN and GRU-FCN efficiency against single-flow CNN models for the host-based intrusion and malware detection task on univariate times series data." *Applied Sciences* 10, no. 7 (2020): 2373.

[10] Fournier, Quentin, Daniel Aloise, Seyed Vahid Azhari, and François Tetreault. "On Improving Deep Learning Trace Analysis with System Call Arguments." In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pp. 120-130. IEEE, 2021.

[11] Tandon, Gaurav, and Philip K. Chan. "Learning Useful System Call Attributes for Anomaly Detection." In *FLAIRS Conference*, pp. 405-411. 2005.

[12] Warrender, Christina, Stephanie Forrest, and Barak Pearlmutter. "Detecting intrusions using system calls: Alternative data models." In *Proceedings of the 1999 IEEE symposium on security and privacy (Cat. No. 99CB36344)*, pp. 133-145. IEEE, 1999.

[13] Liu, Ying, Han Tong Loh, and Aixin Sun. "Imbalanced text classification: A term weighting approach." *Expert systems with Applications* 36, no. 1 (2009): 690-701.

[14] Khor, Kok-Chin, Choo-Yee Ting, and Somnuk Phon-Amnuaisuk. "A cascaded classifier approach for improving detection rates on rare attack categories in network intrusion detection." *Applied Intelligence* 36, no. 2 (2012): 320-329.

[15] Fawcett, Tom, and Foster Provost. "Adaptive fraud detection." *Data mining and knowledge discovery* 1, no. 3 (1997): 291-316.

[16] Malhotra, Pankaj, Anusha Ramakrishnan, Gaurangi Anand, Lovekesh Vig, Puneet Agarwal, and Gautam Shroff. "LSTM-based encoder-decoder for multi-sensor anomaly detection." *arXiv preprint arXiv:1607.00148* (2016).

[17] Lee, Sunwoo, Qiao Kang, Sandeep Madireddy, Prasanna Balaprakash, Ankit Agrawal, Alok Choudhary, Richard Archibald, and Wei-keng Liao. "Improving scalability of parallel CNN training by adjusting mini-batch size at run-time." In *2019 IEEE International Conference on Big Data (Big Data)*, pp. 830-839. IEEE, 2019.

[18] Balles, Lukas, Javier Romero, and Philipp Hennig. "Coupling adaptive batch sizes with learning rates." *arXiv preprint arXiv:1612.05086* (2016).

[19] Smith, Samuel L., Pieter-Jan Kindermans, Chris Ying, and Quoc V. Le. "Don't decay the learning rate, increase the batch size." *arXiv preprint arXiv:1711.00489* (2017).

[20] Balles, Lukas, and Philipp Hennig. "Dissecting adam: The sign, magnitude and variance of stochastic gradients." In *International Conference on Machine Learning*, pp. 404-413. PMLR, 2018.

[21] Zhang, Guodong, Lala Li, Zachary Nado, James Martens, Sushant Sachdeva, George Dahl, Chris Shallue, and Roger B. Grosse. "Which algorithmic choices matter at which batch sizes? insights from a noisy quadratic model." *Advances in neural information processing systems* 32 (2019).

[22] Devarakonda, Aditya, Maxim Naumov, and Michael Garland. "Adabatch: Adaptive batch sizes for training deep neural networks." *arXiv preprint arXiv:1712.02029* (2017).

[23] Li, Yanghao, Naiyan Wang, Jianping Shi, Xiaodi Hou, and Jiaying Liu. "Adaptive batch normalization for practical domain adaptation." *Pattern Recognition* 80 (2018): 109-117.

[24] You, Yang, Igor Gitman, and Boris Ginsburg. "Scaling sgd batch size to 32k for imagenet training." *arXiv preprint arXiv:1708.03888* 6, no. 12 (2017): 6.

[25] Mikami, Hiroaki, Hisahiro Suganuma, Yoshiki Tanaka, and Yuichi Kageyama. "Massively distributed SGD: ImageNet/ResNet-50 training in a flash." *arXiv preprint arXiv:1811.05233* (2018).

[26] You, Yang, Jonathan Hseu, Chris Ying, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. "Large-batch training for LSTM and beyond." In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1-16. 2019.

[27] Malhotra, Pankaj, Lovekesh Vig, Gautam Shroff, and Puneet Agarwal. "Long short term memory networks for anomaly detection in time series." In *Proceedings*, vol. 89, pp. 89-94. 2015.

[28] Ahmad, Subutai, Alexander Lavin, Scott Purdy, and Zuha Agha. "Unsupervised real-time anomaly detection for streaming data." *Neurocomputing* 262 (2017): 134-147.

[29] Schervish, Mark J. "P values: what they are and what they are not." *The American Statistician* 50, no. 3 (1996): 203-206.

[30] Shipmon, Dominique T., Jason M. Gurevitch, Paolo M. Piselli, and Stephen T. Edwards. "Time series anomaly detection; detection of anomalous drops with limited features and sparse examples in noisy highly periodic data." *arXiv preprint arXiv:1708.03665* (2017).

[31] Hundman, Kyle, Valentino Constantinou, Christopher Laporte, Ian Colwell, and Tom Soderstrom. "Detecting spacecraft anomalies using lstms and nonparametric dynamic thresholding." In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 387-395. 2018.

[32] Audibert, Julien, Pietro Michiardi, Frédéric Guyard, Sébastien Marti, and Maria A. Zuluaga. "USAD: unsupervised anomaly detection on multivariate time series." In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 3395-3404. 2020.

[33] Lazarevic, Aleksandar, and Vipin Kumar. "Feature bagging for outlier detection." In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pp. 157-166. 2005.

[34] Kim, Gyuwan, Hayoon Yi, Jangho Lee, Yunheung Paek, and Sungroh Yoon. "LSTM-based system-call language modeling and robust ensemble method for designing host-based intrusion detection systems." *arXiv preprint arXiv:1611.01726* (2016).

[35] Ligh, Michael Hale, Andrew Case, Jamie Levy, and Aaron Walters. *The art of memory forensics: detecting malware and threats in windows, linux, and Mac memory*. John Wiley & Sons, 2014.

[36] Xen Project. (accessed: May 17th, 2022) https://www.xenproject.org

[37] Hizver, Jennia, and Tzi-cker Chiueh. "Real-time deep virtual machine introspection and its applications." In *Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pp. 3-14. 2014.

[38] Egele, Manuel, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. "A survey on automated dynamic malware-analysis techniques and tools." *ACM computing surveys (CSUR)* 44, no. 2 (2008): 1-42.

[39] Joshi, N., and D. B. Choksi. "Implementation of process forensic for system calls." *International Journal of Advanced Research in Engineering & Technology (IJARET)* 5, no. 6 (2014): 77-82.

[40] Payet, Étienne, and Fausto Spoto. "Static analysis of Android programs." *Information and Software Technology* 54, no. 11 (2012): 1192-1201.

[41] Ye, Yanfang, Dingding Wang, Tao Li, and Dongyi Ye. "IMDS: Intelligent malware detection system." In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 1043-1047. 2007.

[42] Masud, Mohammad M., Latifur Khan, and Bhavani Thuraisingham. "A scalable multi-level feature extraction technique to detect malicious executables." *Information Systems Frontiers* 10, no. 1 (2008): 33-45.

[43] Alarifi, Suaad, and Stephen Wolthusen. "Anomaly detection for ephemeral cloud IaaS virtual machines." In *International Conference on Network and System Security*, pp. 321-335. Springer, Berlin, Heidelberg, 2013.

[44] Wang, Wei, Xiao-Hong Guan, and Xiang-Liang Zhang. "Modeling program behaviors by hidden Markov models for intrusion detection." In *Proceedings of*

2004 International Conference on Machine Learning and Cybernetics (IEEE Cat. No. 04EX826), vol. 5, pp. 2830-2835. IEEE, 2004.

[45] Cho, Sung-Bae, and Hyuk-Jang Park. "Efficient anomaly detection by modeling privilege flows using hidden Markov model." *computers & security* 22, no. 1 (2003): 45-55.

[46] Soni, Jayesh, Nagarajan Prabakar, and Himanshu Upadhyay. "Deep learning approach to detect malicious attacks at system level: poster." In *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*, pp. 314-315. 2019.

[47] Ryan, Jake, Meng-Jang Lin, and Risto Miikkulainen. "Intrusion detection with neural networks. advances in neural information processing systems." (1998).

[48] Soni, Jayesh, and Nagarajan Prabakar. "Effective machine learning approach to detect groups of fake reviewers." In *Proceedings of the 14th international conference on data science (ICDATA'18), Las Vegas, NV*, pp. 3-9. 2018.

[49] Wang, Gang, Jinxing Hao, Jian Ma, and Lihua Huang. "A new approach to intrusion detection using Artificial Neural Networks and fuzzy clustering." *Expert systems with applications* 37, no. 9 (2010): 6225-6232.

[50] Creech, Gideon, and Jiankun Hu. "A semantic approach to host-based intrusion detection systems using contiguousand discontiguous system call patterns." *IEEE Transactions on Computers* 63, no. 4 (2013): 807-819.

[51] Soni, Jayesh, Nagarajan Prabakar, and Himanshu Upadhyay. "Feature extraction through deepwalk on weighted graph." In *Proceedings of the 15th international conference on data science (ICDATA'19), Las Vegas, NV*. 2019.

[52] Staudemeyer, Ralf C., and Christian W. Omlin. "Evaluating performance of long short-term memory recurrent neural networks on intrusion detection data." In *Proceedings of the South African institute for computer scientists and information technologists conference*, pp. 218-224. 2013.

[53] Debar, Herve, Monique Becker, and Didier Siboni. "A neural network component for an intrusion detection system." In *Proceedings 1992 IEEE Computer Society Symposium on Research in Security and Privacy*, pp. 240-240. IEEE Computer Society, 1992.

[54] Mukkamala, Srinivas, Guadalupe Janoski, and Andrew Sung. "Intrusion detection using neural networks and support vector machines." In *Proceedings of the 2002 International Joint Conference on Neural Networks. IJCNN'02 (Cat. No. 02CH37290)*, vol. 2, pp. 1702-1707. IEEE, 2002.

[55] Staudemeyer, Ralf C. "Applying long short-term memory recurrent neural networks to intrusion detection." *South African Computer Journal* 56, no. 1 (2015): 136-154.

[56] Soni, Jayesh, Nagarajan Prabakar, and Jong-Hoon Kim. "Prediction of component failures of telepresence robot with temporal data." In *30th Florida conference on recent advances in robotics*. 2017.

[57] Thejas, G. S., Jayesh Soni, Kshitij Chandna, S. S. Iyengar, N. R. Sunitha, and Nagarajan Prabakar. "Learning-based model to fight against fake like clicks on instagram posts." In *2019 SoutheastCon*, pp. 1-8. IEEE, 2019.

[58] Kang, Dae-Ki, Doug Fuller, and Vasant Honavar. "Learning classifiers for misuse and anomaly detection using a bag of system calls representation." In *Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop*, pp. 118-125. IEEE, 2005.

[59] Yeung, Dit-Yan, and Yuxin Ding. "Host-based intrusion detection using dynamic and static behavioral models." *Pattern recognition* 36, no. 1 (2003): 229-243.

[60] Hoang, Xuan Dau, Jiankun Hu, and Peter Bertok. "A multi-layer model for anomaly intrusion detection using program sequences of system calls." In *Proc. 11th IEEE Int'l. Conf*. 2003.

[61] Lee, Wenke, and Salvatore Stolfo. "Data mining approaches for intrusion detection." (1998).

[62] Murtaza, Syed Shariyar, Wael Khreich, Abdelwahab Hamou-Lhadj, and Mario Couture. "A host-based anomaly detection approach by representing system calls as states of kernel modules." In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 431-440. IEEE, 2013.

[63] Suresh Kumar, P., and S. Ramachandram. "Fuzzy-based integration of security and trust in distributed computing." In *Soft Computing for Problem Solving*, pp. 899-912. Springer, Singapore, 2019.

[64] Peddoju, Suresh K., Himanshu Upadhyay, and Shekhar Bhansali. "Health monitoring with low power IoT devices using anomaly detection algorithm." In *2019 Fourth international conference on fog and mobile edge computing (FMEC)*, pp. 278-282. IEEE, 2019.

[65] Reddy, A. Rishika, and P. Suresh Kumar. "Predictive big data analytics in healthcare." In *2016 Second International Conference on Computational Intelligence & Communication Technology (CICT)*, pp. 623-626. IEEE, 2016.

[66] Bottou, Léon, Frank E. Curtis, and Jorge Nocedal. "Optimization methods for large-scale machine learning." *Siam Review* 60, no. 2 (2018): 223-311.

[67] Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[68] Friedlander, Michael P., and Mark Schmidt. "Hybrid deterministic-stochastic methods for data fitting." *SIAM Journal on Scientific Computing* 34, no. 3 (2012): A1380-A1405.

[69] De, Soham, Abhay Yadav, David Jacobs, and Tom Goldstein. "Automated inference with adaptive batches." In *Artificial Intelligence and Statistics*, pp. 1504-1513. PMLR, 2017.

[70] Defazio, Aaron, Francis Bach, and Simon Lacoste-Julien. "SAGA: A fast incremental gradient method with support for non-strongly convex composite objectives." *Advances in neural information processing systems* 27 (2014).

[71] Daneshmand, Hadi, Aurelien Lucchi, and Thomas Hofmann. "Starting small-learning with adaptive sample sizes." In *International conference on machine learning*, pp. 1463-1471. PMLR, 2016.

[72] Vasilkoski, Zlatko, Heather Ames, Ben Chandler, Anatoli Gorchetchnikov, Jasmin Léveillé, Gennady Livitz, Ennio Mingolla, and Massimiliano Versace. "Review of stability properties of neural plasticity rules for implementation on memristive neuromorphic hardware." In *The 2011 International Joint Conference on Neural Networks*, pp. 2563-2569. IEEE, 2011.

[73] Waltz, M., and K. S. Fu. "A heuristic approach to reinforcement learning control systems." *IEEE Transactions on Automatic Control* 10, no. 4 (1965): 390-398.

[74] Yu, Xiao-Hu, and Guo-An Chen. "Efficient backpropagation learning using optimal learning rate and momentum." *Neural Networks* 10, no. 3 (1997): 517-527.

[75] Darken, Christian, and John Moody. "Towards faster stochastic gradient search." *Advances in neural information processing systems* 4 (1991).

[76] Duchi, John, and Yoram Singer. "Efficient online and batch learning using forward backward splitting." *The Journal of Machine Learning Research* 10 (2009): 2899-2934.

[77] Zeiler, Matthew D., Graham W. Taylor, and Rob Fergus. "Adaptive deconvolutional networks for mid and high level feature learning." In *2011 international conference on computer vision*, pp. 2018-2025. IEEE, 2011.

[78] Zou, Fangyu, Li Shen, Zequn Jie, Weizhong Zhang, and Wei Liu. "A sufficient condition for convergences of adam and rmsprop." In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 11127-11135. 2019.

[79] Zhang, Zijun. "Improved adam optimizer for deep neural networks." In *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*, pp. 1-2. IEEE, 2018.

[80] Zeiler, Matthew D. "Adadelta: an adaptive learning rate method." *arXiv preprint arXiv:1212.5701* (2012).

[81] Ruder, Sebastian. "An overview of gradient descent optimization algorithms." *arXiv preprint arXiv:1609.04747* (2016).

[82] Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need." *Advances in neural information processing systems* 30 (2017).

[83] Needell, Deanna, Rachel Ward, and Nati Srebro. "Stochastic gradient descent, weighted sampling, and the randomized Kaczmarz algorithm." *Advances in neural information processing systems* 27 (2014).

[84] Chandola, Varun. "Anomaly detection: A survey varun chandola, arindam banerjee, and vipin kumar." (2007).

[85] Tavallaee, Mahbod, Natalia Stakhanova, and Ali Akbar Ghorbani. "Toward credible evaluation of anomaly-based intrusion-detection methods." *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 40, no. 5 (2010): 516-524.

[86] Deshpande, Prachi, Subhash Chander Sharma, Sateesh K. Peddoju, and S. Junaid. "HIDS: A host based intrusion detection system for cloud computing environment." *International Journal of System Assurance Engineering and Management* 9, no. 3 (2018): 567-576.

[87] Rawat, Sanjay, Ved P. Gulati, Arun K. Pujari, and V. Rao Vemuri. "Intrusion detection using text processing techniques with a binary-weighted cosine metric." *Journal of Information Assurance and Security* 1, no. 1 (2006): 43-50.

[88] Aghaei, Ehsan, and Ehab Al-Shaer. "Threatzoom: neural network for automated vulnerability mitigation." In *Proceedings of the 6th Annual Symposium on Hot Topics in the Science of Security*, pp. 1-3. 2019.

[89] Mahrach, Safaa, Iman El Mir, Abdelkrim Haqiq, and Dijiang Huang. "SDN-based SYN Flooding Defense in Cloud." *Journal of Information Assurance & Security* 13, no. 1 (2018).

[90] Aghaei, Ehsan, and Gursel Serpen. "Ensemble classifier for misuse detection using N-gram feature vectors through operating system call traces." *International Journal of Hybrid Intelligent Systems* 14, no. 3 (2017): 141-154.

[91] Sirigineedi, Surya Srikar, Jayesh Soni, and Himanshu Upadhyay. "Learning-based models to detect runtime phishing activities using urls." In *Proceedings of the 2020 the 4th International Conference on Compute and Data Analysis*, pp. 102-106. 2020.

[92] Serpen, Gursel, and Ehsan Aghaei. "Host-based misuse intrusion detection using PCA feature extraction and kNN classification algorithms." *Intelligent Data Analysis* 22, no. 5 (2018): 1101-1114.

[93] Peddoju, S. K., Upadhyay, H., Soni, J., & Prabakar, N. (2020). Natural language processing based anomalous system call sequences detection with virtual memory introspection. *International Journal of Advanced Computer Science and Applications (IJACSA)*, *11*(5).

[94] Liu, Fei Tony, Kai Ming Ting, and Zhi-Hua Zhou. "Isolation forest." In *2008 eighth ieee international conference on data mining*, pp. 413-422. IEEE, 2008.

[95] Maglaras, Leandros A., Jianmin Jiang, and Tiago Cruz. "Integrated OCSVM mechanism for intrusion detection in SCADA systems." *Electronics Letters* 50, no. 25 (2014): 1935-1936.

[96] Day, William HE, and Herbert Edelsbrunner. "Efficient algorithms for agglomerative hierarchical clustering methods." *Journal of classification* 1, no. 1 (1984): 7-24.

[97] Xiang, Shiming, Feiping Nie, and Changshui Zhang. "Learning a Mahalanobis distance metric for data clustering and classification." *Pattern recognition* 41, no. 12 (2008): 3600-3612.

[98] Bridges, Robert A., Jessie D. Jamieson, and Joel W. Reed. "Setting the threshold for high throughput detectors: A mathematical approach for ensembles of dynamic, heterogeneous, probabilistic anomaly detectors." In *2017 IEEE International Conference on Big Data (Big Data)*, pp. 1071-1078. IEEE, 2017.

[99] Sommer, Robin, and Vern Paxson. "Outside the closed world: On using machine learning for network intrusion detection." In *2010 IEEE symposium on security and privacy*, pp. 305-316. IEEE, 2010.

[100] Clark, James, Zhen Liu, and Nathalie Japkowicz. "Adaptive threshold for outlier detection on data streams." In *2018 IEEE 5th International Conference on Data Science and Advanced Analytics (DSAA)*, pp. 41-49. IEEE, 2018.

[101] Wang, Ke, and Salvatore J. Stolfo. "Anomalous payload-based network intrusion detection." In *International workshop on recent advances in intrusion detection*, pp. 203-222. Springer, Berlin, Heidelberg, 2004.

[102] Haider, Waqas, Jiankun Hu, and Miao Xie. "Towards reliable data feature retrieval and decision engine in host-based anomaly detection systems." In *2015 IEEE 10th Conference on Industrial Electronics and Applications (ICIEA)*, pp. 513-517. IEEE, 2015.

[103] Xie, Miao, Jiankun Hu, and Jill Slay. "Evaluating host-based anomaly detection systems: Application of the one-class SVM algorithm to ADFA-LD." In *2014 11th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*, pp. 978-982. IEEE, 2014.

[104] Crochemore, Maxime, and Wojciech Rytter. *Jewels of stringology: text algorithms*. World Scientific, 2002.

[105] Liao, Xiaoyao, Changzhi Wang, and Wen Chen. "Anomaly Detection of System Call Sequence Based on Dynamic Features and Relaxed-SVM." *Security and Communication Networks* 2022 (2022).

[106] Melvin, A. Alfred Raja, G. Jaspher W. Kathrine, S. Sudhakar Ilango, S. Vimal, Seungmin Rho, Neal N. Xiong, and Yunyoung Nam. "Dynamic malware attack dataset leveraging virtual machine monitor audit data for the detection of intrusions

in cloud." *Transactions on Emerging Telecommunications Technologies* (2021): e4287.

[107] Srinivasan, Siddharth, Akshay Kumar, Manik Mahajan, Dinkar Sitaram, and Sanchika Gupta. "Probabilistic real-time intrusion detection system for docker containers." In *International Symposium on Security in Computing and Communication*, pp. 336-347. Springer, Singapore, 2018.

[108] Mishra, Preeti, Vijay Varadharajan, Emmanuel S. Pilli, and Uday Tupakula. "Vmguard: A vmi-based security architecture for intrusion detection in cloud environment." *IEEE Transactions on Cloud Computing* 8, no. 3 (2018): 957-971.

[109] Ruff, Lukas, Robert A. Vandermeulen, Nico Görnitz, Alexander Binder, Emmanuel Müller, Klaus-Robert Müller, and Marius Kloft. "Deep semi-supervised anomaly detection." *arXiv preprint arXiv:1906.02694* (2019).

[110] Aggarwal, Charu C. "Outlier ensembles: position paper." *ACM SIGKDD Explorations Newsletter* 14, no. 2 (2013): 49-58.

[111] Pang, Guansong, Chunhua Shen, and Anton van den Hengel. "Deep anomaly detection with deviation networks." In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 353-362. 2019.

[112] Schölkopf, Bernhard, John C. Platt, John Shawe-Taylor, Alex J. Smola, and Robert C. Williamson. "Estimating the support of a high-dimensional distribution." *Neural computation* 13, no. 7 (2001): 1443-1471.

[113] Tax, David MJ, and Robert PW Duin. "Support vector domain description." *Pattern recognition letters* 20, no. 11-13 (1999): 1191-1199.

[114] Eskin, Eleazar, Andrew Arnold, Michael Prerau, Leonid Portnoy, and Sal Stolfo. "A geometric framework for unsupervised anomaly detection." In *Applications of data mining in computer security*, pp. 77-101. Springer, Boston, MA, 2002.

[115] McInnes, Leland, John Healy, and Steve Astels. "hdbscan: Hierarchical density based clustering." *J. Open Source Softw.* 2, no. 11 (2017): 205.

[116] Zhou, Chong, and Randy C. Paffenroth. "Anomaly detection with robust deep autoencoders." In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pp. 665-674. 2017.

[117] Erfani, Sarah M., Sutharshan Rajasegarar, Shanika Karunasekera, and Christopher Leckie. "High-dimensional and large-scale anomaly detection using a linear one-class SVM with deep learning." *Pattern Recognition* 58 (2016): 121-134.

[118] De Gregorio, Massimo, and Maurizio Giordano. "An experimental evaluation of weightless neural networks for multi-class classification." *Applied Soft Computing* 72 (2018): 338-354.

[119] Ibarguren, Igor, Jesús M. Pérez, Javier Muguerza, Ibai Gurrutxaga, and Olatz Arbelaitz. "Coverage-based resampling: Building robust consolidated decision trees." *Knowledge-Based Systems* 79 (2015): 51-67.

[120] Chang, Chih-Chung, and Chih-Jen Lin. "LIBSVM: a library for support vector machines." *ACM transactions on intelligent systems and technology (TIST)* 2, no. 3 (2011): 1-27.

[121] Hühn, Jens, and Eyke Hüllermeier. "FURIA: an algorithm for unordered fuzzy rule induction." *Data Mining and Knowledge Discovery* 19, no. 3 (2009): 293-319.

[122] Ahmim, Ahmed, Leandros Maglaras, Mohamed Amine Ferrag, Makhlouf Derdour, and Helge Janicke. "A novel hierarchical intrusion detection system based on decision tree and rules-based models." In *2019 15th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pp. 228-233. IEEE, 2019.

[123] Moustafa, Nour, and Jill Slay. "The evaluation of Network Anomaly Detection Systems: Statistical analysis of the UNSW-NB15 data set and the comparison with the KDD99 data set." *Information Security Journal: A Global Perspective* 25, no. 1-3 (2016): 18-31.

[124] Zong, Wei, Yang-Wai Chow, and Willy Susilo. "A two-stage classifier approach for network intrusion detection." In *International Conference on Information Security Practice and Experience*, pp. 329-340. Springer, Cham, 2018.

[125] Tama, Bayu Adhi, Marco Comuzzi, and Kyung-Hyune Rhee. "TSE-IDS: A two-stage classifier ensemble for intelligent anomaly-based intrusion detection system." *IEEE Access* 7 (2019): 94497-94507.

[126] Tama, Bayu Adhi, and Kyung-Hyune Rhee. "An in-depth experimental study of anomaly detection using gradient boosted machine." *Neural Computing and Applications* 31, no. 4 (2019): 955-965.

[127] Pervez, Muhammad Shakil, and Dewan Md Farid. "Feature selection and intrusion classification in NSL-KDD cup 99 dataset employing SVMs." In *The 8th International Conference on Software, Knowledge, Information Management and Applications (SKIMA 2014)*, pp. 1-6. IEEE, 2014.

[128] Kanakarajan, Navaneeth Kumar, and Kandasamy Muniasamy. "Improving the accuracy of intrusion detection using gar-forest with feature selection." In *Proceedings of the 4th International Conference on Frontiers in Intelligent Computing: Theory and Applications (FICTA) 2015*, pp. 539-547. Springer, New Delhi, 2016.

[129] Pajouh, Hamed Haddad, Reza Javidan, Raouf Khayami, Ali Dehghantanha, and Kim-Kwang Raymond Choo. "A two-layer dimension reduction and two-tier classification model for anomaly-based intrusion detection in IoT backbone networks." *IEEE Transactions on Emerging Topics in Computing* 7, no. 2 (2016): 314-323.

[130] Tama, Bayu Adhi, Lewis Nkenyereye, SM Riazul Islam, and Kyung-Sup Kwak. "An enhanced anomaly detection in web traffic using a stack of classifier ensemble." *IEEE Access* 8 (2020): 24120-24134.

VITA

JAYESH SONI

2018-Present	Ph.D., Computer Science Knight School of Computing and Information Sciences Florida International University, Miami, Florida
2016-2018	Research Scholar Knight School of Computing and Information Sciences Florida International University, Miami, Florida
2015-2017	Master of Technology, Computer Science Dept. of Computer Science and Engineering Manipal University Jaipur (MUJ) Jaipur, Rajasthan, India
2014-2015	Software Developer Refresh IT Software Solutions Ahmedabad, Gujarat, India
2010 -2014	Bachelor of Engineering, Computer Science Dept. of Computer Science and Engineering Gujarat Technological University (GTU) Ahmedabad, Gujarat, India

PUBLICATIONS AND PRESENTATIONS

Soni, Jayesh, and Nagarajan Prabakar. "KeyNet: Enhancing Cybersecurity with Deep Learning-Based LSTM on Keystroke Dynamics for Authentication." In International Conference on Intelligent Human Computer Interaction, pp. 761-771. Springer, Cham, 2021.

Soni, Jayesh, Suresh K. Peddoju, Nagarajan Prabakar, and Himanshu Upadhyay. "Comparative Analysis of LSTM, One-Class SVM, and PCA to Monitor Real-Time Malware Threats Using System Call Sequences and Virtual Machine Introspection." In International Conference on Communication, Computing and Electronics Systems, pp. 113-127. Springer, Singapore, 2021.

Soni, Jayesh, Nagarajan Prabakar, and Himanshu Upadhyay. "Visualizing High-Dimensional Data Using t-Distributed Stochastic Neighbor Embedding Algorithm." In Principles of Data Science, pp. 189-206. Springer, Cham, 2020.

Soni, Jayesh, Nagarajan Prabakar, and Himanshu Upadhyay. "Behavioral Analysis of System Call Sequences Using LSTM Seq-Seq, Cosine Similarity and Jaccard Similarity for Real-Time Anomaly Detection." In 2019 International Conference on Computational Science and Computational Intelligence (CSCI), pp. 214-219. IEEE, 2019.

Soni, Jayesh, Nagarajan Prabakar, and Himanshu Upadhyay. “Feature Extraction through Deepwalk on Weighted Graph”, Proceedings of the 15th Springer International Conference on Data Science (ICDATA), Las Vegas, Nevada, pp. 164-170, July 29 - Aug 1, 2019.

Soni, Jayesh, Nagarajan Prabakar, and Himanshu Upadhyay. “Comparative Analysis of LSTM Sequence-Sequence and Auto Encoder for real-time anomaly detection using system call sequences”, International Journal of Innovative Research in Computer and Communication Engineering (IJIRCCE), Vol. 7, Issue 12, pp. 4225-4230, January 2019.

Soni, Jayesh, Nagarajan Prabakar, and Himanshu Upadhyay."Deep learning approach to detect malicious attacks at system level: poster." In Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks, pp. 314-315. 2019.

Soni, Jayesh, and Nagarajan Prabakar. "Effective machine learning approach to detect groups of fake reviewers." In Proceedings of the 14th international conference on data science (ICDATA’18), Las Vegas, NV, pp. 3-9. 2018.

Soni, Jayesh, Nagarajan Prabakar, and Himanshu Upadhyay. “Towards detecting fake spammers groups in social media: An unsupervised Deep Learning approach”, accepted for publication in Deep Learning for Social Media Data Analytics (SMDA 2021) in Springer Book Series “Studies in Big Data”.

Soni, Jayesh, Nagarajan Prabakar, and Himanshu Upadhyay. “Machine Learning-based Cyber Threat Anomaly Detection in Virtualized Application Processes” submitted for Review in The 24th International Conference on Artificial Intelligence (ICAI'22) in Springer Book Series “Transactions on Computational Science & Computational Intelligence”.

Soni, Jayesh, Nagarajan Prabakar, and Himanshu Upadhyay. “Quantum Computing enabled Machine Learning for an enhanced model training approach”, submitted for Review in Springer QC2022: Quantum Computing: A Shift from Bits to Qubits in “Studies in Computational Intelligence" book series.

Soni, Jayesh, Nagarajan Prabakar, and Himanshu Upadhyay. “A Transfer Learning Approach for Hurricane Damage Assessment using satellite imagery”, submitted for Review in SciTech Publishing EDMDL2022: “Earth Observation Data Analytics Using Machine and Deep Learning”.