APPROVAL SHEET

Title of Thesis: CAPTURING AND ANALYSING KERNEL EVENTS FOR ANOMALY DETECTION IN WINDOWS O.S.

Name of Candidate: Swapnil Mahendra Bhosale
Master of Science
Computer Science, 2021

Thesis and Abstract Approved: Anupam Joshi

Dr. Anupam Joshi
Director, UMBC Center for Cybersecurity Professor and Chair, Computer Science and Electrical Engineering

Date Approved: 7/28/2021

ABSTRACT

Title of dissertation: CAPTURING AND ANALYZING KERNEL EVENTS FOR ANOMALY DETECTION IN WINDOWS O.S.

Swapnil M. Bhosale, Master of Science, 2021

Dissertation directed by: Dr. Anupam Joshi
Department of Computer Science

This thesis applies recent advances in NLP to anomaly detection in Windows OS. More specifically, we experiment using fastText as an embedding combined with an LSTM for state prediction. We explore whether we can model the normal process behavior on Windows and recognize deviations caused by malware. The actions performed by malware typically involve modifying the file system, modifying the Windows registry to change the system configuration & network actions. We developed a Windows kernel driver to capture file, registry, network events. We use fastText to train the embedding model to represent events as vectors. FastText learns not only the syntactic information but also semantic information hidden in the observed kernel events. The IP address, file path, process path, registry key etc. have syntactic structure and semantic relationships. Next, we train a sequence-based anomaly detection model using LSTM to learn the typical behavior of the Windows OS and the processes running in the system. Lastly, we propose a technique to identify measured windows event sequences as normal, or anomalies representing an

attack. We evaluate the performance of this anomaly detection system to detect attacks on a system from their kernel level behavior. We collect datasets for normal (attack-free) and process takeover (attack) using the kernel driver system we develop, and use these to test our detection. We show that our approach has high accuracy, precision, and recall. We also propose to release our kernel driver to capture events as open source, to facilitate further research in this area.

CAPTURING AND ANALYSING KERNEL EVENTS FOR ANOMALY DETECTION IN WINDOWS O.S.

by

Swapnil Bhosale

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, Baltimore County in partial fulfillment
of the requirements for the degree of
Master of Science
2021

Advisory Committee:
Dr. Anupam Joshi, Chair/Advisor
Dr. Jeff Avery
Dr. James Oates
Dr. Tim Finin

Acknowledgments

I owe my gratitude to all the people who helped me in the journey and made this thesis possible. First, I would like to give my deepest thanks to my advisor, Dr. Anupam Joshi. I want to thank him for all the time and effort he put in to guide me and for always making sure that I am on the right track. I highly value all the advice and technical insights he has provided to help me reach this point. I want to extend my appreciation to the rest of my committee - Dr. Jeff Avery, Dr. James Oates, and Dr. Tim Finin. I want to extend my sincere thanks to Dr. Jeff Avery. His insights and attention have helped me grow and shape the quality of my writing and thought process, which has been especially crucial in these past few months dedicated to the completion of this thesis. I would especially like to thank Amina Mahmood and Swati K. for helping me collecting the datasets for the different experiments. I would also like to thank all the EBIQUITY members; it has been my privilege working alongside you. Finally, I would like to thank my family and friends for their endless support.!

Table of Contents

List of Figures

List of Abbreviations

LSTM	Long Short Term Memory
O.S	Operating System
t-SNE	t-distributed Stochastic Neighbor Embedding
VM	Virtual Machine
NLP	Natural Language Processing
SVM	Support Vector Machines
MSE	Mean Squared Error
MAE	Mean Absolute Error
WFP	Windows Filtering Platform
SDK	Software Development Kit
WDK	Windows Development Kit
DLL	Dynamic Link Library
REST	Representational State Transfer
API	Application Program Interface
CLI	Command Line Interface
RCE	Remote Code Execution
IP	Internet Protocol
CSV	Comma Separated Value
TCP	Transmission Control Protocol
UTF	Unicode Transformation Format
BSOD	Blue Screen Of Death

Chapter 1: Introduction

Cyberattacks, and the related malware used to affect them, are seeing significant growth [1] and damage caused by these attacks is causing siginificant harm, economic and otherwise [2, 3]. Detecting attacks, especially zero days, is a serious challenge. Windows being the most widely used operating system, is also one of the most often attacked [4]. Creating a system that can provide the data needed to detect cyber attacks/malware is thus essential. The actions performed by malware typically involve modifying the file system, modifying the Windows registry to change the system configuration & Network actions [5]. Malware is also becoming intelligent and can evade and/or trick userspace-based, signature driven detection approaches. Even such malware however leaves footprints in the kernel space. Also, antivirus and network intrusion detection systems have proven inefficient against detecting the advanced threats [6]. Moreover, commercially available virus scanners and Windows have proprietary codebases and a tiny active community with limited online resources on Windows kernel driver development. Thus, it is challenging but essential to have a system that can provide data of process events in the kernel space.

The goal of anomaly detection systems is to monitor the computer system

to detect abnormal behavior, which conventional signature-based methods could not detect. Unsuprevised Anomaly detection systems overcome the limitation of the signature-based detection to identify the unknown attack. The idea behind this technique is that the profile of a malicious activity differs from typical user behaviour [7]. Anomaly detection techniques have the ability of solid generalizability and to detect the new attacks [8]. However, anomaly detection techniques have a significant drawback of sizeable false alarm rates. We reduce the false alarms and improve the system's accuracy by using the anomaly counts per minute as a measure to identify the anomaly window.

Since we can model the malicious activities [5] using file, registry, network access, and due to limited availability of tools and information about Windows kernel development, in this work we developed a kernel resident system to tap into the file, registry, and network events in Windows. We show that we capture all events relating to files, registry, and networks, including processes spawned in the O.S. Next, this thesis applies recent advancements in the NLP domain to Windows kernel events anomaly detection. It tests whether advanced NLP embedding approaches can be used to model kernel events that do not contain typical natural language but are human-readable text. Then we verify if such representation is suitable for building an unsupervised Windows anomaly detection system that uses rich information provided in the kernel events.

The key contribution of our work:

1. Develop a Windows kernel driver capable of tapping into the file, registry, and

network events in kernel space

2. Applies the fastText embedding to file, registry, and network events captured in kernel space of Windows to Anomaly Detection

Chapter 2: Related work

In this section we discuss literature associated with Anomaly Detection in Windows and corresponding approaches.

2.1 Windows Anomaly Detection

Several approaches have been proposed for anomaly detection in Windows, but they are based on univariate data such as a file, registry, or network [4, 9]. Stolfo presented an anomaly detector of Windows that uses a probabilistic model and support vector machines to use only registry events in October 2005 paper [4]. The researchers developed a registry sensor to monitor user space registry events. This sensor deploys a wrapper over systems like RegMon from sysinternals [10]. The probabilistic model is based upon a probability density estimation, while the second one using SVM that iteratively finds the maximal margin hyperplane, which best separates the training data from the origin. It May be viewed as a regular two-class Support Vector Machine (SVM) where all the training data lies in the first class, and the origin is taken as the only member of the second class. Thus, the hyperplane (or linear decision boundary) corresponds to the classification rule [4].

Rabadi and Teo [11] in their paper suggests detecting the malware by gener-

alizing the system API calls happening in the Operating System. They use cuckoo sandbox [12] to generate the data for the execution of the API calls along with their parameters. M. Alazab uses a similar approach in the paper [13] to profile malware using System API calls. In the paper, [14] Lin used the builtin tools available with containerization software to extract the System calls made by the containers. After feature engineering, they train the typical behavior model of each different application. To detect the anomalies, they also developed and introduced an Autoencoder to model normal application behavior. Based on the same hypothesis, if the reconstruction error is higher, it indicates that the model did not encounter that data before, so the result is one of an anomaly. Berlin and slaughter suggested in their paper [15] the idea of the agentless anomaly detection system. Using audit logs and cuckoo reports of malicious code execution are the primary data source for the machine learning model. To classify benign and malware samples, they train an automated linear logistic regression classifier. According to Jindal in their paper [16], a new idea of automatically learning the process behavior from the dynamic analysis report of the binary is introduced. With their approach, they train a word embedding model and the LSTM based neural network. Our approach in this thesis is different from these approaches in few ways: we capture the file, registry, and network events in kernel space using the kernel driver developed by us, we train a model on typical Windows user behavior, thus generalizing the Operating System behavior, we present an anomaly detection technique based on threshold count per minute in the event windows sequence.

2.2 FastText Sentence Embedding

In their paper, Mikolov [17] presented a pioneering approach in the domain of Natural Language Processing (NLP) that was a breakthrough both in terms of quality and computing complexity. It is a neural net with two layers that accepts the text as input to the model and generates the embedding vectors. However, Word2Vec has one limitation, which is essential for our work, it can only represent the words present in the training corpus. Thus, one needs to handle the case when the input text to the model was not part of the training corpus. As a result of the following research, fastText was developed and presented [18, 19]. FastText has a configurable parameter called n-gram. It allows it to train the model not only on the input words in the corpus but also the character n-grams of the words.

Training on n-gram enables to generate embedding for the unseen words and incorporate some of the syntactic structure of words in the embedding. Borawdekar, in their work [20] describes the idea of adding semantic information to the text by prepending the column names to the text. We employ a similar technique in our approach to add semantic information to the event data. Jindal, in their paper [16] trains a Word2Vec model for word embedding. FastText is also applied in the log anomaly detection systems [21, 22] to represent the logs as vectors. The logs also consist of syntactic structure such as keyvalue fields in the log statement, thus allowing fastText to learn the hidden information present in the log statement structure.

An n-gram is a contiguous sequence of n items from the given sequence, in this

context, the characters. N-gram is a commonly used term in many NLP methods. FastText works on the principle that there exists hidden information in the structure of the word. The unknown words can be created from the n-grams or the shorter words. For example, if there is no embedding for the word 'woman', the embedding for the 'man' is present in the training corpus. Moreover, the 2-gram 'wo' is also learned from the other words present in the training corpus. Then the aggregated embedding of the *wo* and *man* given the good approximation for the *woman*.

In NLP, there is also a common task to create embedding for the sentences or the long text parts. It is a challenging and tricky problem depending on the used language model to aggregate embedding for the different words. Using FastText is very straightforward to generate embeddings for the sentences. It does this by taking the L2 norm of each word vector in the sentence and then divide each vector by its norm. Finally, all word vectors are averaged to create a single vector to represent the sentence, and this is believed to capture the semantic information present in the words of the sentences. We apply a similar approach in our work. Since each event, i.e., file, registry, and network, have different features, we can model each event as a sentence, and using fastText, we can generate an embedding of fixed size for each event. In addition to its capability of generating vectors for unseen words, fastText is also plausible in this context as a mechanism for embedding.

Chapter 3: Event Capturing

3.1 Introduction

One popularly known tool-set, Windows Sysinternals, allows users to view advanced system information and utilities. These tools are impactful in viewing user-level processes and system-level when they run through admin users in Windows. With the tremendous increase in cyber attacks, tools such as Sysinternals have become crucial in identifying threats within the system and the network. A common problem in using this particular tool-set is that each tool has different uses. There is a challenge in finding a tool that would be useful in scenarios of malware, rootkits, and logging inside a Windows system that would be able to display all of the system's data. Windows filter manager is a system that allows third-party "Miniflters" drivers and already comes installed with the operating system. It is essential to enable third-party Miniflter drivers to be loaded and run on system level. As a result, these kernel drivers have the highest privileges and are harder to mitigate against. We use "Minifilter" - Windows kernel driver architecture to implement our Windows Kernel Driver to tap into the file, registry & network events in the Operating System. The Minifilter driver will be introduced that is capable of being used in such various scenarios. This solves the problem of combining different

tooling capabilities into one and safe from the attacks where the attacker can observe the user-space applications and user-space application events, data, logs, etc tampered with. The driver architecture and how to tap into these events will be described in the later sections.

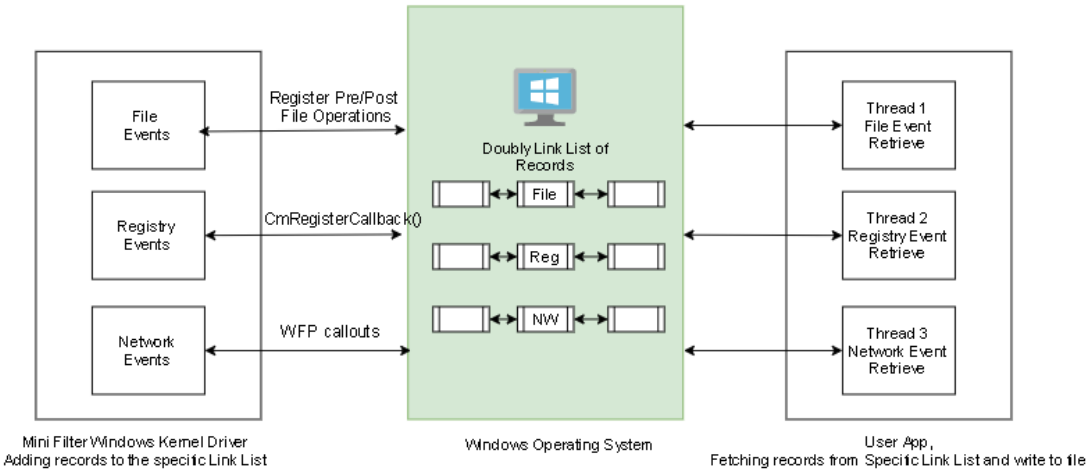


Figure 3.1: Minifilter kernel driver architecture

3.2 Implementation

Windows is a proprietary Operating System. There is a small active community that focuses on kernel driver development. Moreover, most virus scanner companies do not disclose their driver code information out of tactical business advantage reasons. Thus, it is challenging to implement a Windows kernel driver on your own. There is an abundance of information available out there on the Internet. But, most information is vast, misleading and you can very soon find yourself confused. The information out there is challenging to develop a kernel driver for our usecase. Debugging kernel issues is also a challenging task and requires knowl-

edge of special tools like Wingdb and analysis of the memory dump. One memory leak or error can lead to the famous Blue Screen of Death (BSOD) error on Windows. Hence, it is essential to handle memory management carefully in the driver implementation.

In Windows, there are two types of drivers: legacy file system filter drivers and minifilter drivers. A legacy Windows file system filter driver can directly modify file system behavior and is called during each file system input/output (I/O) operation. However, Microsoft currently favors the minifilter drivers. A minifilter driver indirectly connects to the file system by registering all the needed callback filtering procedures in the filter manager. The filter manager calls the filtering functions registered by a minifilter for a specific I/O operation. In short, even though minifilter do not have direct access to the file system, minifilters can still capture or change the system behavior. However, they are much easier to develop and use than the legacy alternatives. We build out file, registry, and network tapping system on top of the minifilter driver architecture.

A minifilter driver‘s DriverEntry routine is executed when the Operating System loads the minifilter driver. Thus, all the initialization of variables and data structure is done within it‘s DriverEntry routine. The minifilter driver calls FltRegisterFilter to register callback routines with the filter manager and FltStartFiltering to notify the filter manager that the minifilter driver is ready to start attaching to volumes and filtering I/O requests. Minifilter driver instances are defined in the INF file used to install the minifilter driver. A minifilter driver's INF file must define a default instance, and it can define additional instances. These definitions apply

across all volumes. Each instance definition includes the instance name, altitude, and flags indicating whether the instance can be attached automatically, manually, or both. The default instance is used to order minifilter drivers so that the filter manager calls the minifilter driver's mount an instance setup callback routines in the correct order.

The individual details on how to tap into file, registry and network events will be described in sections 3.2.1, 3.2.2 & 3.2.3. As shown in figure 3.1, we maintain a doubly linked list data structure in the kernel memory space to store the captured events. These events will be deleted from the linked list when the driver is unloaded from the system or our userspace application has consumed the data. Each sensor file, registry, and network writes the new record structure at the one end of the doubly linked list, and the user application consumes the records from the other end of the doubly linked list. Also, we need to take care of the synchronization issues using semaphores and locks because these events happen at microsecond granularity; thus, modification of the linked list data structure is the critical section part of the driver code. The userspace application uses **FilterConnectCommunicationPort** API to connect to the driver's communication port. We employ a multithreading technique in the user application, and each thread communicates with the specific event sensor. Thus, the user application generates three individual files for file, registry, and network events in Comma Separated Value (CSV) format.

3.2.1 Tapping file events

File system filtering is the mechanism by which drivers can intercept calls destined to the file system. The main function of a file system Minifilter is processing the I/O operations by implementing 'pre' and/or 'post' callbacks for the operations of interest. We hook into 40 different types of 'pre' and 'post' file system operations. First, we need to register the operations callback. Our Minifilter driver must indicate which operations it's interested in. This is done at registration time with an array of **FLT OPERATION REGISTRATION** structures. Now we pass this structure to **FLT REGISTRATION** structure to register our driver. Figure 3.2 explains about prototype of the function used for tapping into file events.

```
//register callbacks for required operations
CONST FLT_OPERATION_REGISTRATION Callbacks[] = {
    {IRP_MJ_CREATE, 0, PreCreateCallback, nullptr},
    {IRP_MJ_DELETE, 0, PreDeleteCallback, nullptr},
    {IRP_MJ_SET_INFORMATION, 0, PreDeleteSetInformation, nullptr},
    {IRP_MJ_OPERATION_END}};

//pass the filter above t0 filter register method
NTSTATUS FLTAPI FltRegisterFilter(
    PDRIVER_OBJECT Driver,
    const FLT_REGISTRATION *Registration,
    PFLT_FILTER *RetFilter);

//start filtering the file events
status = FltStartFiltering(gFilterHandle);
```

Figure 3.2: Function prototypes for file event tapping

Once the Driver setup is finished, all we need to do is to start filtering in the driver's DriverEntry. A Miniflter driver must register itself as a Minifilter with the Filter manager, specifying various settings, such as what operations it wishes

to intercept and altitude settings. Having done the necessary initialization above, we can call FltRegisterFilter to register. If successful, the driver can do further initialization as needed and finally call FltStartFiltering to actually start filtering operations.

3.2.2 Tapping registry events

For Registry entries, the Configuration Manager (the part in the Executive that deals with the registry) can be used to register for notifications when registry keys are accessed. The basics of building a registry filter driver is that the driver needs to call **CmRegisterCallbackEx** routine during the **DriverEntry** routine, and then call the **CmUnRegisterCallback** during the driver unload. Registry filteres are considered as minifilter drivers are are loaded based on Load Order Groups and Altitude just as file system minifilter are. Figure 3.3 is the prototype of the API signature:

```
NTSTATUS CmRegisterCallbackEx(
    PEX_CALLBACK_FUNCTION Function,
    PCUNICODE_STRING Altitude,
    PVOID Driver,
    PVOID Context,
    PLARGE_INTEGER Cookie,
    PVOID Reserved
);
```

Figure 3.3: Registry tapping function signature

Its function is to register a callback Routine. The Result of a successful registration is passed to the Cookie parameter. Figure 3.4 describes prototype for the

first parameter of the API, the callback routine:

```
NTSTATUS CmRegisterCallbackEx(
    PEX_CALLBACK_FUNCTION Function,
    PCUNICODE_STRING Altitude,
    PVOID Driver,
    PVOID Context,
    PLARGE_INTEGER Cookie,
    PVOID Reserved
);
```

Figure 3.4: Registry callback function prototype

CallbackContext is the Context argument passed to CmRegisterCallbackEx. The first argument is, in fact, an enumeration, REG NOTIFY CLASS, describing the operation for which the callback is invoked, if its 'pre' or 'post' notification. The second argument is a pointer to a structure that contains information that is specific to the type of registry operation. The last argument is again the operation specific. The CmUnRegisterCallback routine as shown is figure 3.5 is called to unregister a callback.

```
NTSTATUS CmUnRegisterCallback(
    LARGE_INTEGER Cookie
);
```

Figure 3.5: Registry unregister callback function prototype

3.2.3 Tapping network events

The Base Filtering Engine is the component of the Windows Filtering Platform that Microsoft has already implemented for you. The entire WFP API is

centered around interacting with this core component. Writing a driver with the Windows Filtering Platform usually involves registering a series of objects to the Base Filtering Engine, each of which allows you to interact with network traffic in different ways. Filters, Callouts, Layers, and Sublayers are the primary objects that you interact with as the author of a WFP driver. All of these objects 'live inside' the Base Filtering Engine. We register with the following layers of Windows Filtering Platform:

1. FWPM LAYER ALE FLOW ESTABLISHED V4
2. FWPM LAYER STREAM V4

The first layer allows us to get callbacks for any new network connection established. This layer also provides metadata such as process id and process path. The second layer allows us to inspect actual packets flowing through the established network connection. The process id and process path metadata are not available in the second layer. Thus to resolve this issue, as shown in figure 3.6 prototype of the API, we use a concept of WFP called associating the flow context. It allows us to attach a user-defined data structure to the established connection packets, which in turn allows us to figure out the process id and name to which these packets belong. The user-defined structure can contain any metadata. We store the process-id and process-name. Because the process-id can be reassigned to another process by the O.S. Thus, retrieving the process name from process-id while dumping the event to file can give us a wrong process name if queried in the future. Moreover, it might be possible when we are capturing the process name from the process id, and there is

no process running in the O.S with that process id in the future. Thus, the decision was taken to store process-id along with process-name in the flow context structure. Thus, we can just copy this structure values when we extract the metadata from the packets.

```
NTSTATUS FwpsFlowAssociateContext(
    flowHandle,
    FWPS_LAYER_STREAM_V4,
    calloutId,
    (UINT64)context
);
```

Figure 3.6: Associate user defined structure to packets

FlowDeleteFn gets called when a connection is closed for the TCP connection which has flowcontext associated with it. With this function we free the memory allocated user defined data structure attached to the flow. Figure 3.7 describes the prototype of the function, *flowContextId* is the address of the allocated user defined flow context structure. Due to the limitation of the metadata unavailable at TCP stream layer, we only process those network event who's connection was established after driver was loaded and the flow context is allocated by our driver in kernel memory.

```
NTSTATUS FlowDeleteFn(
    UINT16 layerId,
    UINT32 calloutId,
    UINT64 flowContext
);
```

Figure 3.7: Deleting flow context and free the user allocated memory

Current implementation of the network filtering only support TCP protocol along with IPv4 protocol. However, the design can be extended easily to tap into IPv6 protocol as well. All we need to do is just tap into the FWPM LAYER ALE FLOW ESTABLISHED V6 layer.

Chapter 4: Implementation of Anomaly Detection System

This chapter presents the complete methodology of the anomaly detection system used in the thesis. The dataset used for training the model and the preprocessing of the data are presented. The architecture and the hyperparameter settings are described for the model and their findings, and finally, the anomaly detection solution used is explained. The first task is to create an event representation suitable for further processing by machine learning. This representation uses NLP embedding to keep the information contained in the Windows events. We present the sentence embedding using the fastText in the section 4.2.

The unsupervised anomaly detection model based on LSTM is presented in section 4.3. Model is trained to predict the next window sequence of events, and also model is trained to reconstruct same event window using Autoencoder [23] model, based on the event windows generated by normal system operations. Distance between predicted and real window sequences of events is measured and compared with the threshold to detect the anomalies. Overview of the whole system is in fig 4.1. First, we perform embedding of the input window sequence and then feed it to the model for the prediction. The output is compared with the original input sequence, and the prediction error is calculated. The anomaly detector identifies

current windows as an anomaly if the error is above the threshold.

The next window prediction model did not perform well. Thus, the research is focused on using the Autoencoder LSTM model to reconstruct the input sequence. The idea of using the Autoencoder is if the reconstruction error is more than the specified threshold, then there is an anomaly because the model has not seen that data in the past or process is accessing some resources which it is not supposed to access. Thus, it is an anomaly.

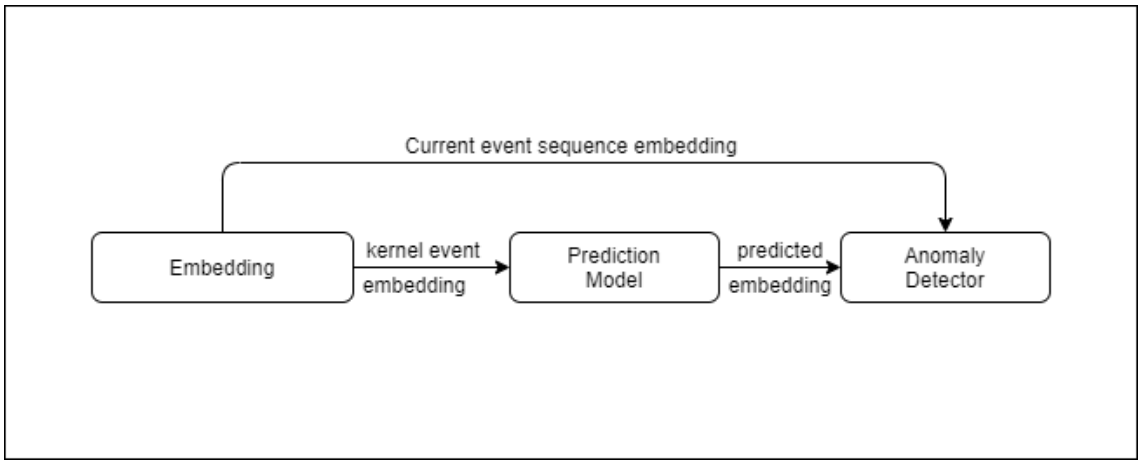


Figure 4.1: Data flow in the proposed architecture

4.1 Data

The data used in this thesis is generated by running the kernel driver developed by us on the Windows 10 host machine. Beyond The data, we discuss how we train a sentence embedding model using data collected from kernel driver events. Two unique aspects characterize our approach: (1) Using unstructured text representation of the structural data as an input to the training process (irrespective of

the column type, all record fields are converted to the text tokens.) and (2) Using the unsupervised sentence embedding technique to generate vectors from different events modeled as a sentence.

4.1.1 Data Generation

We gathered the data by running the kernel driver on the Windows host machine to evaluate the anomaly detection system. Beyond the normal execution of the standard programs such as Windows Word, Internet Explorer, WinZip, Microsoft Paint, Google Chrome, the training also included performing the housekeeping tasks such as emptying the Recycle Bin, using the Control Panel, updating the firewall settings, create, delete, move files folder, etc. [4]. All experiment data were acquired during the typical usage of the real user.

The kernel driver we developed can run on Windows 7, 8, or 10. We used Windows 10 for the experiments because that is the latest V.M. image available on the Windows website. Also, this V.M. comes with all the required libraries and SDK already installed. These V.M. are not licensed and expired after few months, and after that, you can run this V.M. continuously for only 1 hour. After 1 hour is auto shutdowns.

The training data for our experiment was collected on Windows 10 for 3 hours of normal usage. We informally define normal usage [4], which we believe to be a typical use of the Windows platform in the home setting. For example, we assume all users would log in, browse some internet websites, reads some emails, use word

processing software such as Windows word, etc., then log off. This type of session is assumed to be relatively typical for many computer users. Normal programs are those which are bundled with the operating system or in use by most Windows users. Creating a realistic testing environment is an arduous task and testing the system under a variety of environments is a direction for future work.

The simulated home use of Windows generated a clean (attack-free) dataset that has approximately 8 million records in total. File and registry each had approximately 3.5 million records, and the network had ∼1 million records. The reason behind the large discrepancy with network events is described in section 3.2.3. Due to the limitation of the metadata available at the TCP stream layer, we only process those network events whose connection was established after the driver was loaded in kernel memory. This does not cause any issues because we can still capture all the network events while the system is under attack. The new network connections will be established when the system is under attack; thus, we can capture the network events associated with the attack. The normal program run also comprised of browsing using Google Chrome and Elastic Search process. We execute a few REST APIs exposed by ElasticSearch to create and access few database indexes as a part of training data. We later take over these processes and show that we successfully identify the anomalies. These normal programs are intended to simulate an ordinary Windows session. The normal processes also include the processes spawned by O.S., such as sc.exe, logonui.exe, dmclient.exe, etc.

The attacks were carried out using Metasploit framework [24], and the attack run includes the process takeover using the publicly available exploits for the Elas-

tic Search version 1.1.1, vulnerability *CVE-2014-3120* and Google Chrome version 80.0.3987.87, vulnerability *CVE-2020-6418*. The attacks were active for a specific duration of time, and meterpreter TCP reverse shell was spawned to perform Remote Code Execution (RCE) on the target machine. We take over the ElasticSearch process twenty-six times. Out of twenty-six attacks were run, each has a different duration varying from 1 minute to 5 minutes. In these attacks, the activities performed using reverse TCP shell includes spawning a new process like calculator.exe, grabbing a screenshot of the system, streaming the screen share of the remote system on the attacker system, deleting files, uploading privilege escalation script files, migrating meterpreter session to another process, retrieving I.P. tables info, etc. These activities emulate the typical attack session. The attacker might run a new program, upload script files for privilege escalation, share the screen and migrate the session to the process with more system privileges. All attacks were run in different sessions and only happened within a specific period during the session. Thus this gives us the ground truth when the anomaly was present in the system.

Carrying out attack slowly (performing single or small steps after some time interval) and then applying the anomaly detection is an interesting step in determining the accuracy of the model. Thus we emulate slow attack for a 1-hr duration experiment and collect the dataset. We take screenshot after certain time and stream the screen share for $\sim$1 to $\sim$2 mins duration. We collect 8,675,588 records for the experiment. The slow attack was carried out at 20:57, 21:03, 21:15 & 21:39. Later, we explode a malware in controlled environment with our driver running to collect the dataset. Later we evaluate the anomaly detection system against the collected

dataset. The malware was collected from the publicly available website [25]. The dynamic analysis report [26] was collected from the VirusTotal [27]. We verify the dynamic behaviour events with the dataset that we collect and indeed the driver captures all the file, registry and network events reported in the report [26]. In section 5.3 the results are presented.

4.1.2 Data Preprocessing

The kernel driver dumps file, network, and registry events in the CSV format in the individual text files. After loading the events in the dataframe, they are trimmed by removing the columns with the additional system information which are not of interest, such as process id and timestamp, and removing the events where attributes are missing or corrupt. A few events are missing or corrupt, mostly caused by limited kernel memory constraints for storing large text string file path, registry path, and process path. Empty registry values are replaced with the string "none": Some registry operations like "RegNtPreEnumerateValueKey" do not have a registry-value field. Moreover, '\' has a special meaning; thus, we replace all '\' with '\\' to mitigate any Unicode (UTF) encoding issues. It is a general practice in programming to represent path separators with a double backslash.

For the preprocessing of the events, we prepend the column attribute string to the actual value [20] for some of the features of each event type. This is expected to add semantic information to the event data and provide more weightage to the specific attributes of the event for their contribution towards the meaning of the

neighboring words. Individual data frames for file, network, and registry events are then merged and sort by the timestamp. Figure 4.2 describes how individual files, registries, and network events are preprocessed. Each event will act as a sentence in the corpus for sentence embedding using the fastText model. Integer values from the dataset are converted to the corresponding string value. Moreover, all strings are converted to the lower case for consistency. In figure 4.2 the registry type, ports, process name columns are prepended with the column name and converted to the lower case. Finally, all words all joined by a space character representing a sentence.

The hidden information in the sentence embedding approach infers latent semantic information in token associations and co-occurrences and encodes it in vectors. The dataset is sorted with the timestamp, thus, simulating the input data as time-series data for the model to train on. This will capture the semantic relation of the process with file, registry, and network events. Since the dataset is sorted by timestamp, we believe this will capture the temporal attributes of the process such as process 'P', first create file 'X', then perform registry operations 'Y' and then perform network connection to IP address 'Z'.

4.2 FastText embedding

Embedding encodes information contained in the text to the numerical values, usually vectors. In this thesis, we use fastText sentence embedding, and in combination, we apply some textification techniques described 4.1.2. We group the training corpus event by process names and then sort each group by timestamp. It is needed

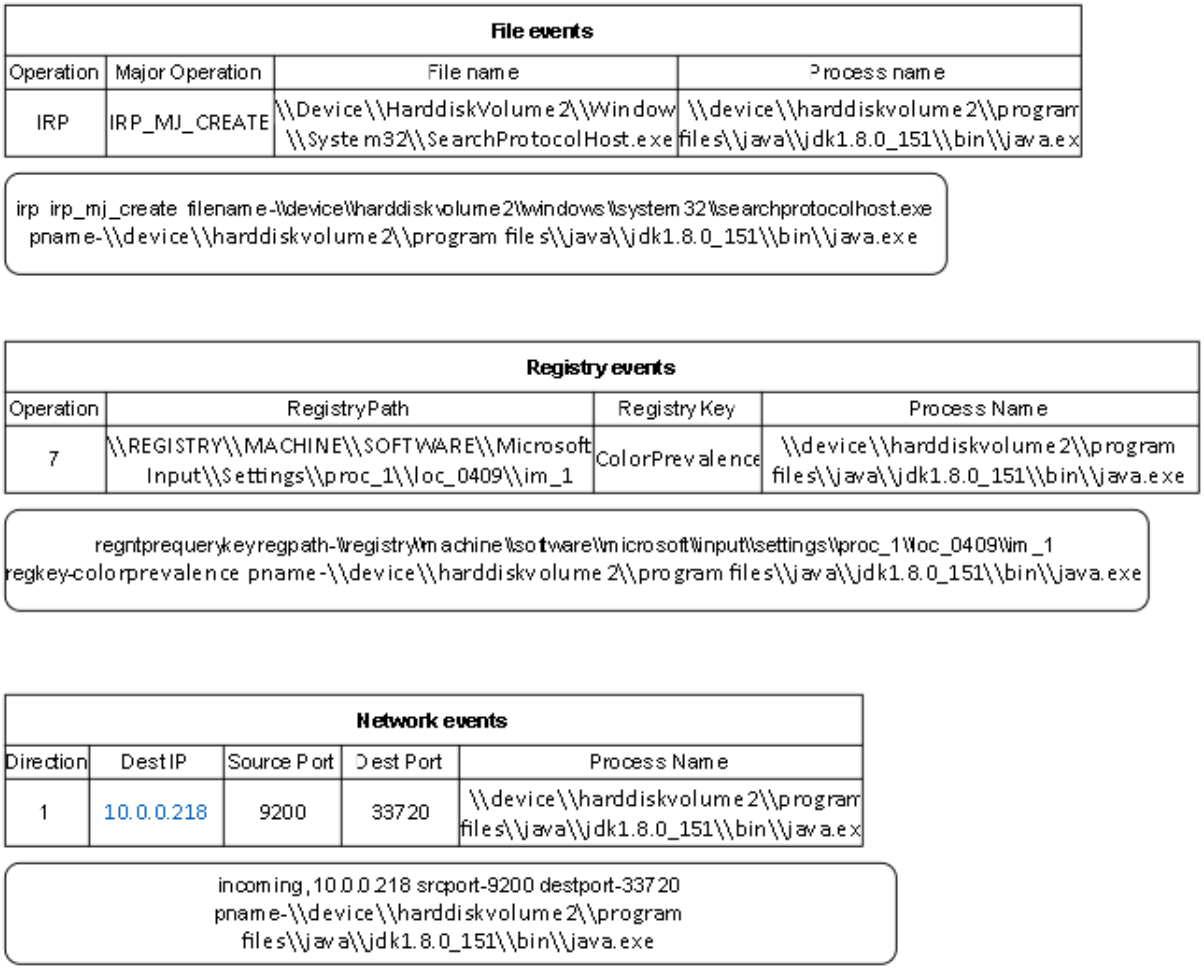


Figure 4.2: Text preprocessing view of the events

because some contextual information might not be included in the single event data. The grouping of events by process name provides essential information to learn for the embedding. Pre-trained word embeddings as Word2Vec from [17] are common in the NLP domain. However, since we are dealing with the data related explicitly to Windows O.S, we need to train our custom embedding model.

In this thesis, as described in section 4.1.2, we combine the NLP sentence embedding technique for each event type and add additional semantic information

to the events. This paper [19] presents an interesting idea for solving the 'unknown' words and tries to exploit the hidden semantic information and syntactic structure. The **IP address**, **file path**, **process path**, **registry key**, clearly have syntactic structure and semantics.

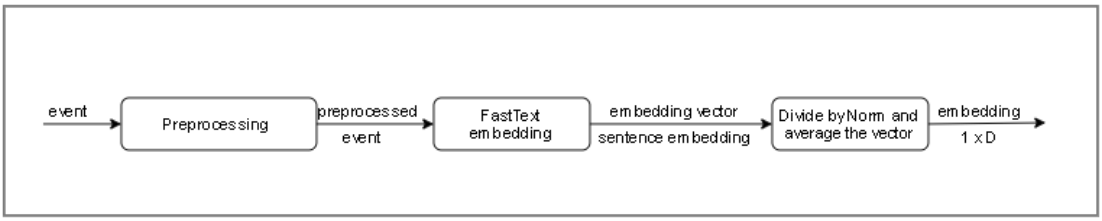


Figure 4.3: Data flow in embedding

The final embedding is for each event is described in the figure 4.3. Raw events first go through the preprocessing step, where the relevant information is extracted from the event. We strip some columns from the events because some of the information contained in the event is irrelevant and noisy such as time and process id. The combined events are, of course, sorted by timestamp, thus preserving the contextual information. Preprocessed data then goes through the fastText model. This part is straightforward. The custom trained model returns a vector of size "d" for each word in the event. For each word vector, we find its L2 norm, divide the vector by norm, and then average the vectors to produce a single vector to represent a single event. Thus irrespective of the input event, the model always outputs a fixed dimension vector.

4.2.1 FastText model configuration and training

We use an unsupervised approach based on FastText implementation to build a sentence embedding model from the Windows events data. Our training approach operates on the unstructured text corpus organized as a collection of English-like sentences. There is no need to label the training data as we use unsupervised training. The fastText code defines a neighborhood window to compute the contribution to the nearby words. The output of the fastText model for each sentence is the N x D dimension.

where, "N" is Number of words in the sentence & "D" is Dimension size for each word

Table 4.1: fastText hyper-parameters settings

vector_size	window_size	min_word	n-gram	iteration	down_sampling
100	20	1	20	50	1e-3

As shown in the table 4.1, the model was trained for 50 iterations. We choose the fastText 'SkipGram' model for training. We also experiment with the various settings, but the above table provides the hyperparameters that worked best in our experiments. There was a total of 7,133,471 actual events and 28,566,338 raw words in the training corpus. The evaluation of the trained model will be discussed in chapter 5.

4.3 Unsupervised Anomaly Detection model

The goal of this thesis is to create an unsupervised anomaly detection model. There are various approaches in sequence analysis, but the most commonly used is to train a model on the normal data to predict the next item of sequence or used an autoencoder to reconstruct the output from the input. Finally, the prediction is compared with the real values to determine if the current values are anomalous. This high-level approach is used in [14, 28–30].

We tried both approaches, but the autoencoder model performed well. Thus we choose an autoencoder neural network for unsupervised model training and anomaly detection to meet our goal of achieving anomaly detection in Windows O.S. The autoencoder learns the latent representation of the input data and learns to reconstruct them. The LSTM encoder learns a fixedlength vector representation of the input timeseries, and the LSTM decoder uses this representation to reconstruct the timeseries using the current hidden state the value predicted at the previous timestep [30]. Intuitively given the input sequence, can autoencoder reconstruct the input? If not, then we can say that the given file, registry, and network event combination do not seem ordinary, thus Anomaly. The autoencoder network consists of encoding and decoding regions shown in model configuration figure 4.6. Input data is compressed in the encoding region and reconstructed in the decoding region. The model replicates its input data after compressing the input data through intermediate neural network layers. During the training phase, a model is built to minimize the difference between its input and output. When the autoen-

coder model is sufficiently trained, the model can produce output data with a small reconstruction error compared with the normal input data. We can then use the reconstruction error to implement the anomaly detection. Specifically, when the autoencoder model produces an output with a high reconstruction error, we can infer an abnormal input data is detected.

Our neural net consists of six layers of neurons with 'tanh' activation in addition to the input and output layers. Each iteration trains the autoencoder model until the model converges. The convergence criteria of the model were set if the learning from the previous iteration is not greater than 1e-3 on the validation data, then stop the learning and restore the weights from the previous epoch. As shown in figure 4.5 After twenty-five epochs, the training was converged. During the training, our objective is to minimize the mean squared error (MSE)4.1. We perform backpropagation with Keras library and use ADAM optimizer.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - x_i)^2 \tag{4.1}$$

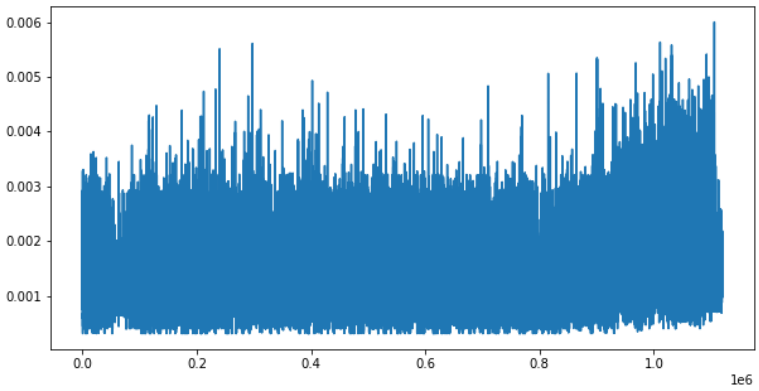
Experiments with different layers and sizes of hidden dimensions will be described in chapter 5. However, the dimension of input and out of the LSTM layer is kept similar to the embedding size, i.e., 100, in our experiments.

The input to the resulting model is divided into batches. The resulting model is trained to reconstruct the sequence of given 'window_size.' The model tried to minimize the MSE error from the input and predicted output vectors. Here it is important to note that the dimension of the vector can be large. We also experimented

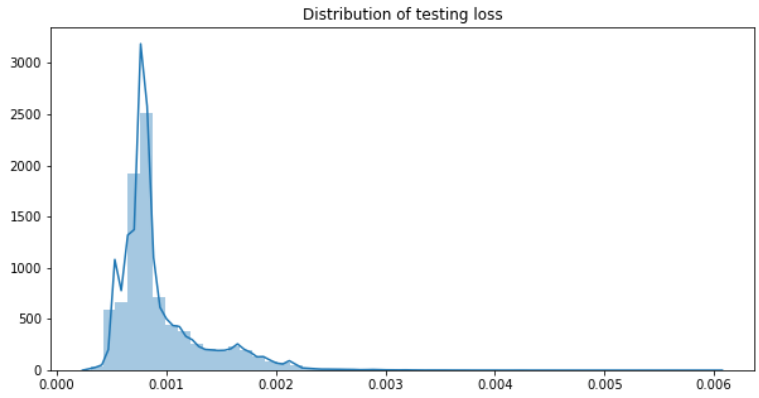
with MAE loss function, however. MSE has proven to work best after few experiments with the metrics mentioned earlier. Even in our case with relatively high dimensions ranging from 100 to 300. The training and validation loss is captured in figure 4.5. The model converged after 25 iterations.

To detect anomalies prediction model is supplied with **(batch_size X window_size X embedding_size)** input vector, and the distance between its output Y^ and Y is measured using the same metric as in the training phase. The current window is label as anomalous if the distance is greater than the threshold. Optimal setting of correct threshold is a challenging problem, and it is beyond the scope of this thesis. Some sophisticated solutions, like dynamic shareholding from exists, but no out-of-box implementation is available. So simple threshold, computed as confidence interval from errors on training data, is used in this thesis. Threshold-based on the standard deviation is computed on the testing data. After training, we evaluate the test data and based on reconstruction error computed from testing data using the following formula.

$$t = mean(errors) + 2 * std(errors) \tag{4.2}$$



(a)



(b)

Figure 4.4: (a) test data reconstruction error (b) distribution of test data reconstruction error

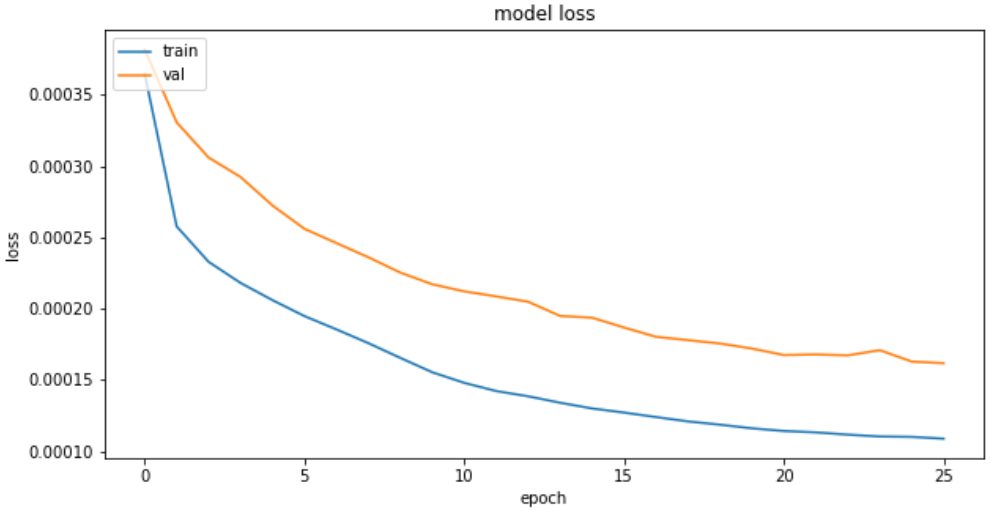


Figure 4.5: LSTM training and validation loss plot

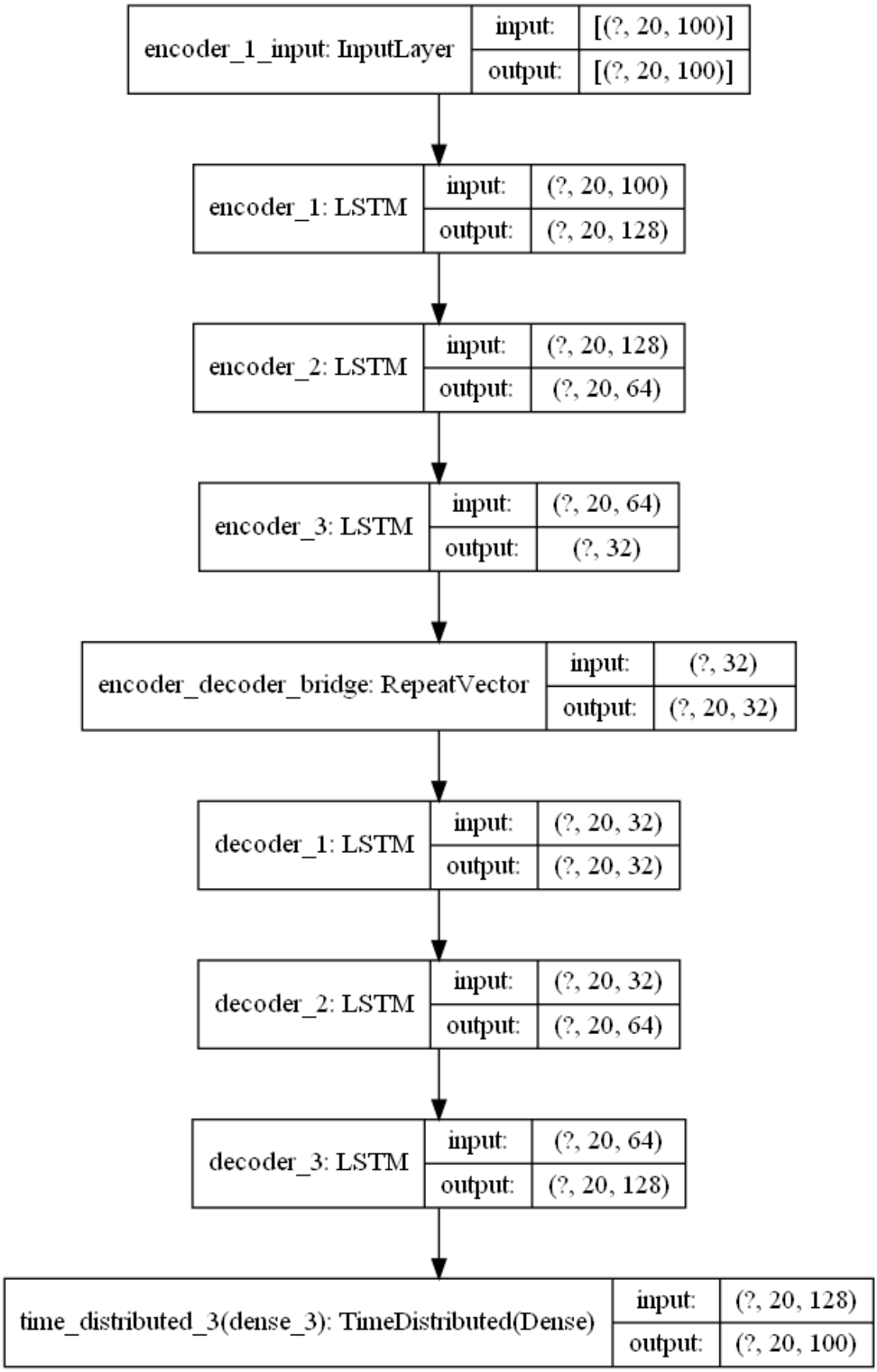


Figure 4.6: Autoencoder LSTM model layer sizes

Chapter 5: Results

This chapter presents the results achieved using the LSTM based sequence anomaly detection model in the Windows Operating System. Multiple experiments conducted to verify the hypothesis and evaluate the proposed solution. This chapter firstly describes the datasets in section 5.1. Then we discuss the analysis of the fastText model for event embedding. Then we evaluate anomaly detection model in section 5.3. Finally, we discuss the summary of findings from the experiments.

5.1 Dataset

As discussed in section 5.1 to evaluate the anomaly detection model, we take over the java ElasticSearch process and Google Chrome browser process using publically available exploits for the Metasploit [24] framework. The developed kernel driver runs on the VM on which the attack is being carried out. The driver in the kernel space will monitor all the Operating System's file, network, and registry activities. Since we know the ground truth when Metasploit took over the machine and when the meterpreter reverse shell connection was closed, we expect the model to show a significant amount of anomalies during that time. Each take-over experiment varied from one minute to five minutes. Moreover, each take-over experiment

dataset was captured for ten to the 1-hour duration time. It is expected to see more anomalies only when the machine was taken over, and anomalous activities were carried out.

Similarly, we collect the twenty-five normal datasets varying in duration from 10 minutes to 2 hours. We defined normal behaviour in section 5.1. We evaluate the model using 55 different datasets collected on different VM's and different users. Out of which, 25 are normal experiments, and 30 are taking over experiments. We propose a threshold calculation technique in the following section, and the F1 score for various threshold settings is described in table 5.1.

Finally, we perform a slow attack as explained in section and explode a malware in controlled environment. The dynamic analysis report has file, registry and network events. We match them against the collected the dataset and indeed the driver captures those all events with associated metadata.

5.2 Embedding Analysis

We train a custom fastText sentence embedding model on the training dataset. Additionally, several parameter configurations were tried. The best parameters that worked for us are described in this section 4.2.1. The parameters discussed in that section directly affect the results of the FastText embedding. The n-gram default range is 3-6. However, the 5-20 range worked best for us. Multiple embedding dimensions (50, 100, 150, 300) were tried out. The more the embedding dimension size is, the more it takes to train the embedding model, the more memory it takes,

and the more it takes for the LSTM model to train on. Thus, the embedding dimension size should be chosen carefully.

The embedding model was trained on the training dataset. Similar words form the cluster, allowing the LSTM model to learn about the semantic meaning of the events. We use the dimensionality reduction method to convert high-dimensional data into lower dimensional data for visualization purposes. Since the experiments, we carried out using high dimensions which can not be visualized directly. We can use the dimensionality reduction to convert the data into two dimensions. This will allow us to show the distributions of the data on the scatter plot. To plot the training data, we choose to use t-Distributed Stochastic Neighbor Embedding (t-SNE) [31]. It is a technique for visualization of the similarity of the data. t-SNE preserves the local structure of the data and some global structures such as clusters while reducing the dimensionality.

Figure 5.1 and 5.2 are visualizations of the fastText embedding of the training dataset and few selected process events from the training dataset. It looks like the fastText model was able to cluster the events by process names. The training dataset has 6,373,181 events, and selected processes for the visualization contribute 322351 of the total events. FastText exposes API *most_similar* which allows you to get the closest words with the given input text word. If we search for process java, we see all the events related to java are in the output. Moreover if we fetch top 10 closest words to 'filename-device\\harddiskvolume2\\windows \\system32 \\cryptbase.dll' then in the response we see all the DLL's which start with name **crypt**. Thus we can say that FastText has learned the syntactical and semantics of the input dataset. In

figure 5.1 all the words in the training corpus were plotted to visualize the training. We can see in the figure that clusters are formed for similar words. The cluster in the center of the figure are the events for the registrykey having prefix 'regkey-registry \\machine \\software \\microsoft \\windows'. In figure 5.1 we select a few processes, and using the tSNE technique, we reduce the higher dimensions to twodimension for visualization. Each point in the figure is the vector representation of the event. We can see, the cluster is formed for the 'java.exe' and 'ctfmon.exe' process. However, the events related to the Google Chrome process have formed multiple small clusters and spread across the 2D plane. Similarly, few small clusters are formed for the 'taskmgr.exe' process.

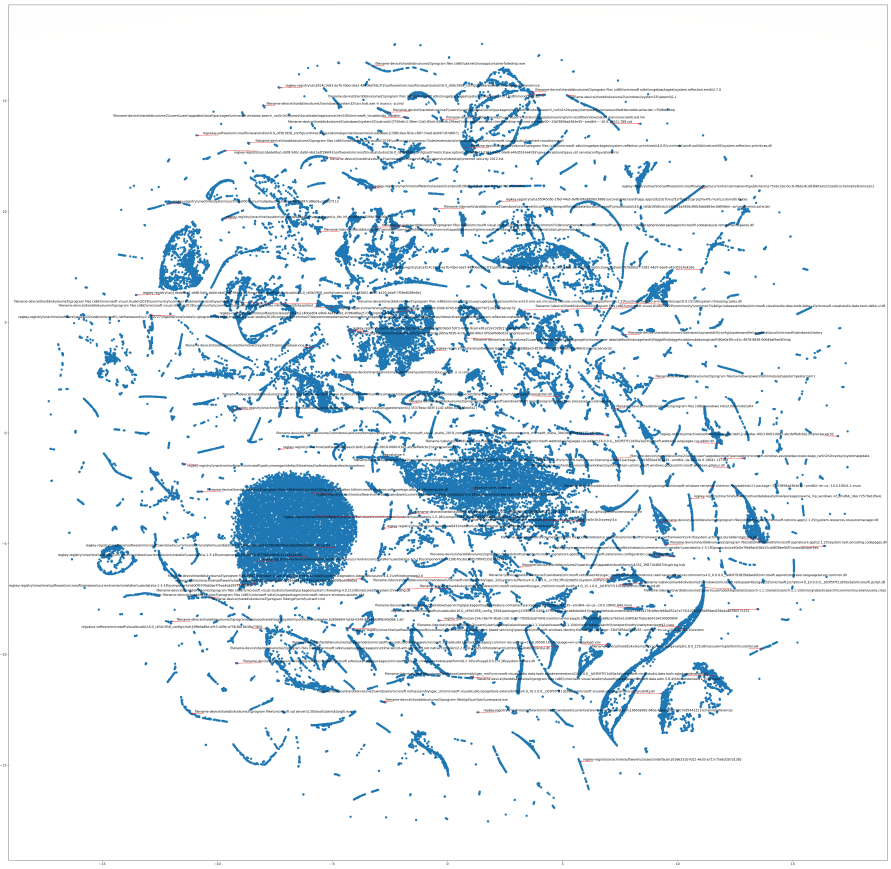


Figure 5.1: t-SNE visualization of words in training dataset

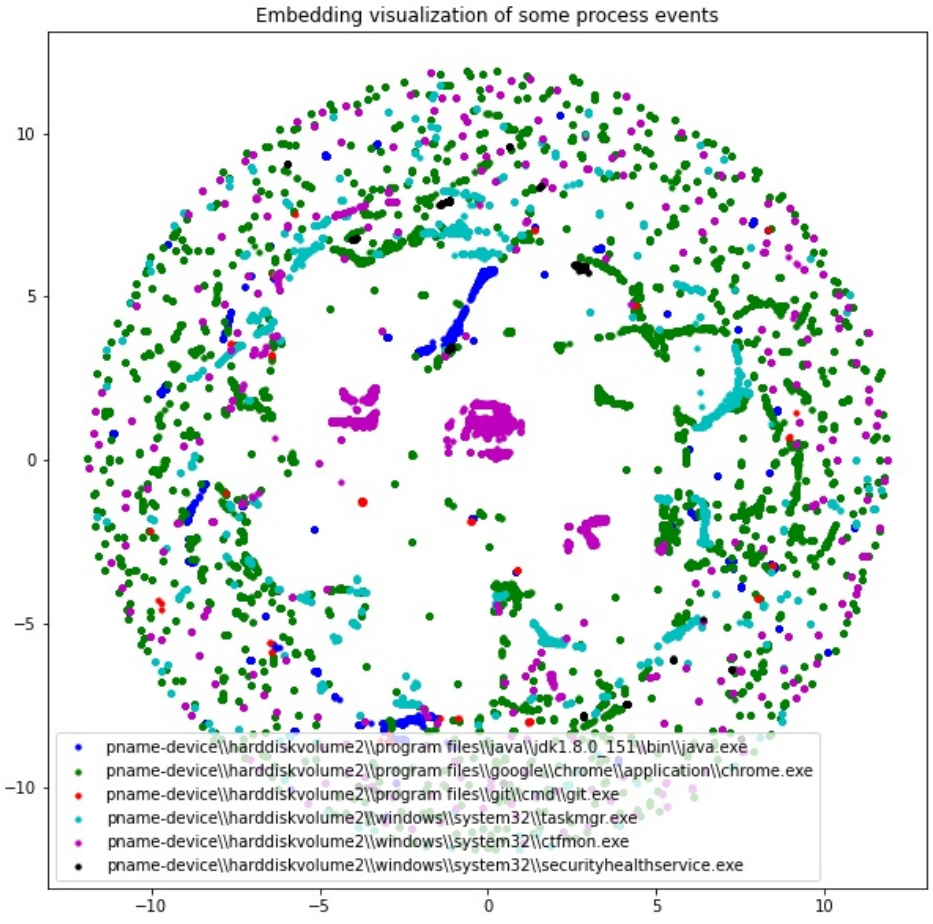


Figure 5.2: t-SNE visualization of events of the few processes

5.3 Anomaly Detection

Firstly, several unsupervised models with different layers and the size of hidden layers were evaluated to determine the model's reasonable parameters. We also perform experiments with the loss function. We experimented using Mean Absolute Error (MAE) and Mean Squared Error (MSE). However, changing the loss function did not affect the model accuracy. We also perform the experiments with various window sizes varying from 5 to 50. However, window size 20 performed best and was used in the evaluation. Large layer size in the Autoencoder model took significantly longer to train while not providing any additional improvements. Parameter values that worked well with all sizes of embedding are 128 as an initial input layer, three layers for compression, each having half number of neurons from the previous layer in the sequence. For the decoding part, we use three layers, and each layer has twice the number of neurons from the previous layer. The last output layer is the dense layer, and the size of the last layer is 100, which is the same as the embedding size. These configuration settings of the model were used in all following experiments.

We developed a benchmarking script to calculate the per minute anomaly threshold count. We will describe the approach in the next section. We evaluate various hyper-parameter settings of the LSTM model like autoencoder number of layers and layer size, loss function, activation functions used in the layers, etc. The results of only the best settings are presented here. As discussed in section 5.1, we collect the datasets for the attack-free and ElasticSearch take-over experiments and perform the F1 matrix calculation using them.

For the given window of the events, if the reconstruction error exceeds the threshold calculated during training, the window is labeled anomalous. We check for all the windows in the given input event sequence and label the corresponding windows as an anomaly or normal. Then, we aggregate and sum the anomalies for each minute in the given input sequence. Now, we evaluate our model on the anomaly counts per minute. At the same time, we observed more spikes in the reconstruction error when the attack was in progress. Thus, aggregating anomaly counts per minute made sense, and the decision was taken to evaluate the model on the anomaly counts per minute metric. In figures 5.4 , 5.5 and 5.6 we can clearly see the more number of anomaly count when the ElasticSearch process was taken over.

Results in Table 5.1 show that the model converges after threshold count 45. In the given table, there were 26 datasets for the ElasticSearch take-over and 25 normal datasets. We also observed that the larger the n-gram size is, the better char-grams that fit the hypothesis that n-grams can help exploit the syntax like file, registry paths, IP address. Dimension of the embedding seemed to have no significant effect on the accuracy than the n-gram configuration.

We now fix the threshold as 30 for per minute anomaly count and evaluate the chrome take-over experiment datasets. In the figure 5.11 we report the anomaly counts reported per minute for those experiments. We take over chrome four times. The models report anomalies all four times in the given period when the chrome process was taken over. Moreover, the dataset was captured with the help of few researchers, thus providing varied user behavior data. It is essential because it helps

validate the hypothesis that the system generalizes the Windows system behavior instead of the particular test user. If the test session data consist of only a single user, then system evaluation might be wrong and biased toward the single user. The anomaly detection model seemed to have generalized the Windows OS behavior and reported zero or no significant anomaly count per minute on the attack-free dataset. Thus, we can conclude that model learned the typical behavior of the processes and learn to detect the anomalies.

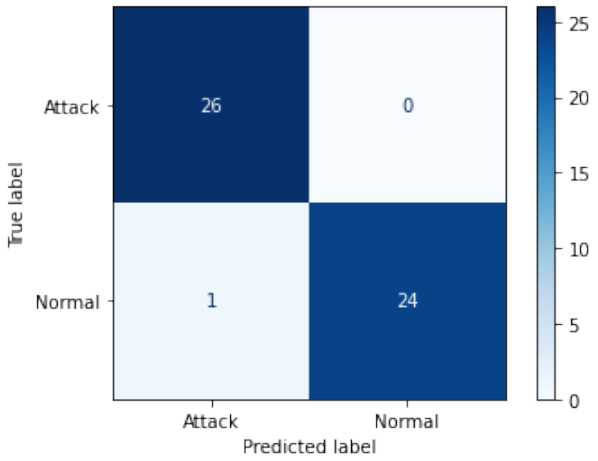


Figure 5.3: Confusion matrix for threshold 30

Table 5.1: F1 matrix calculation

Threshold count per minute	Attack-free metric			Attack metric		
	Precision	Recall	F1	Precision	Recall	F1
5	1.0	.4	.57	.54	1.0	.7
10	1.0	.58	.73	.69	1.0	.82
15	1.0	.84	.91	.87	1.0	.93
20	1.0	.92	.96	.93	1.0	.96
25	1.0	.96	.98	.96	1.0	.98
30	1.0	.96	.98	.96	1.0	.98
35	1.0	.96	.98	.96	1.0	.98
40	1.0	.96	.98	.96	1.0	.98
45	1.0	1.0	1.0	1.0	1.0	1.0

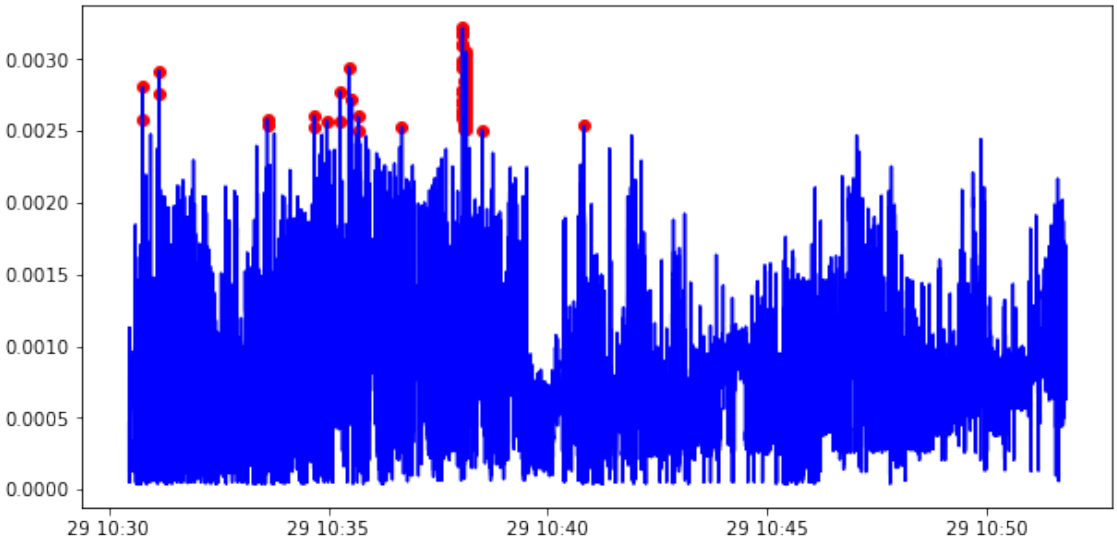


Figure 5.4: ElasticSearch takeover form 10:38 to 10:41

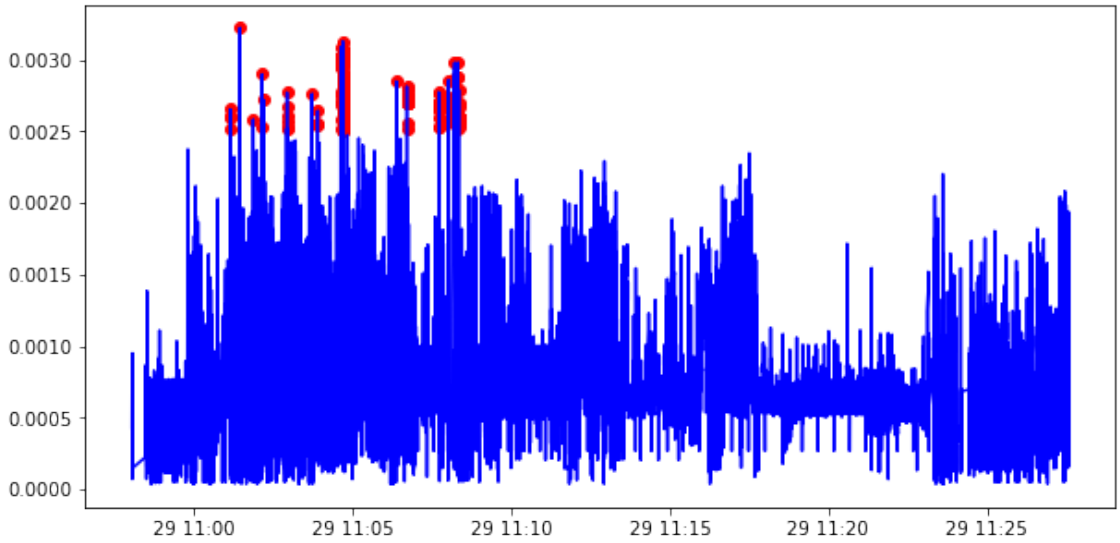


Figure 5.5: ElasticSearch takeover form 11:04 to 11:15

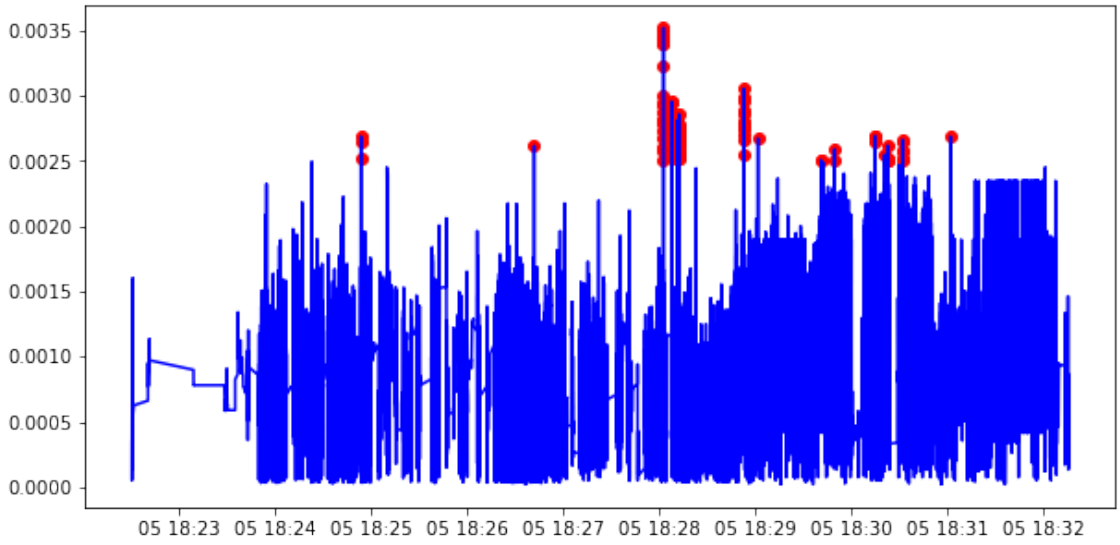


Figure 5.6: ElasticSearch takeover form 18:28 to 18:31

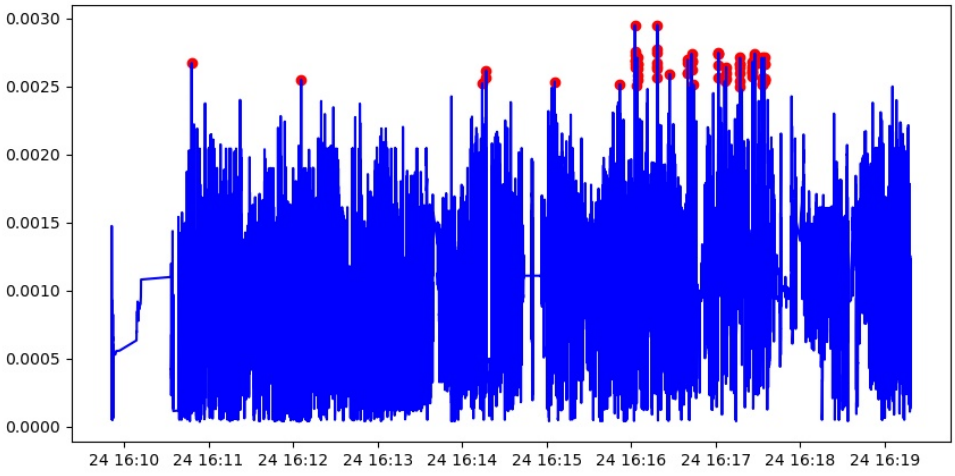


Figure 5.7: Chrome takeover form 16:16 to 16:18

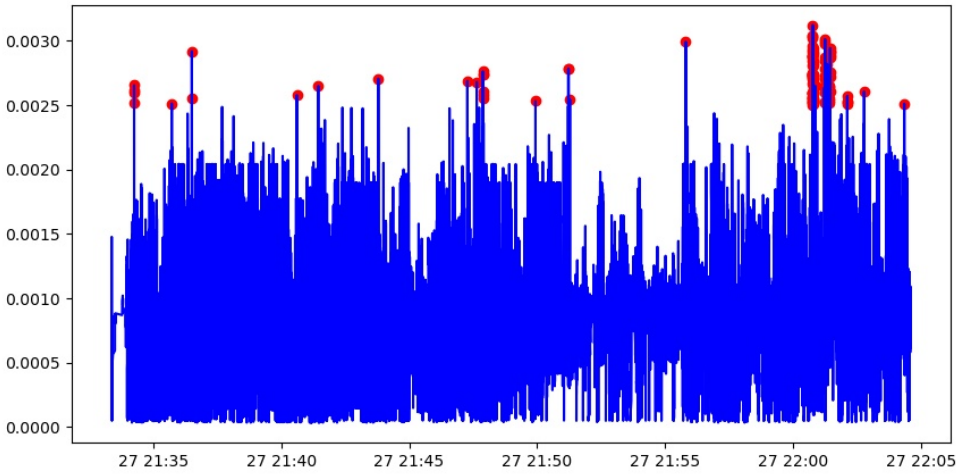


Figure 5.8: Chrome takeover form 22:00 to 22:02

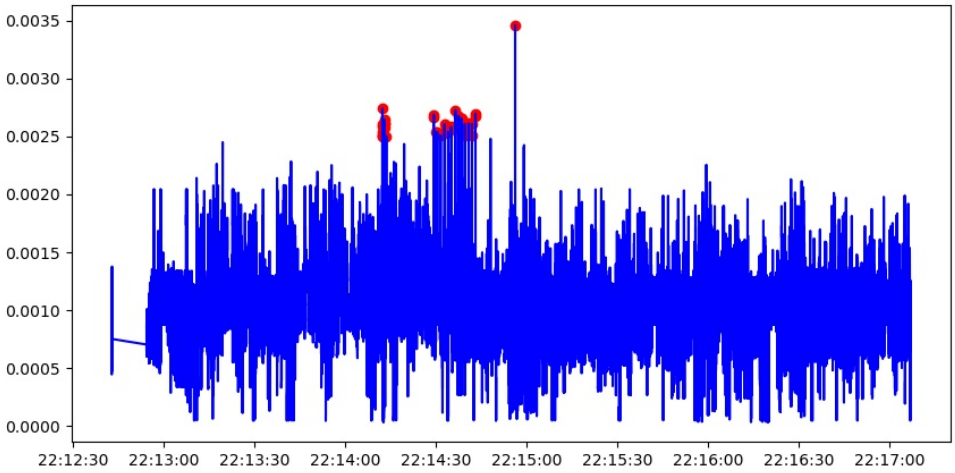


Figure 5.9: Chrome takeover form 22:14 to 22:14

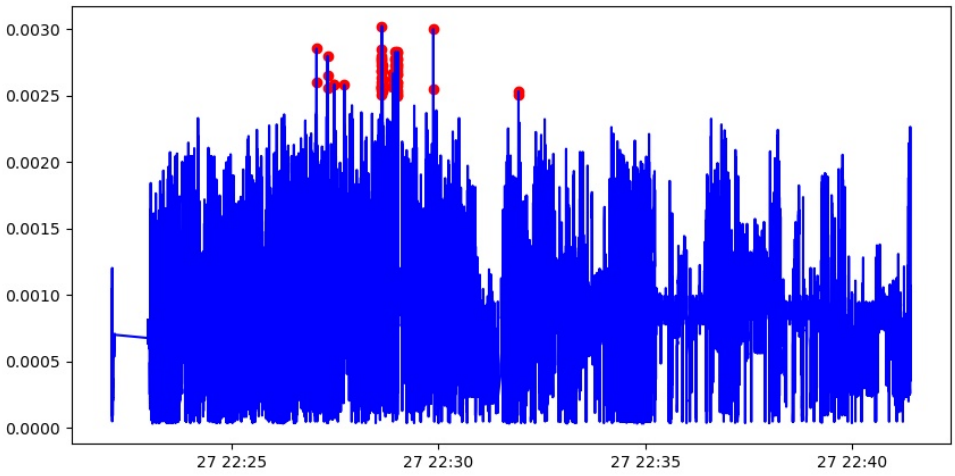


Figure 5.10: Chrome takeover form 22:28 to 22:31

Chrome take over Duration			
16:16 - 16:18	**22:00 - 22:02**	**22:14 - 22:14**	**22:28 - 22:31**
2021-04-24 16:09:00 0 2021-04-24 16:10:00 1 2021-04-24 16:11:00 0 2021-04-24 16:12:00 1 2021-04-24 16:13:00 0 2021-04-24 16:14:00 3 2021-04-24 16:15:00 2 **2021-04-24 16:16:00 33** **2021-04-24 16:17:00 41** **2021-04-24 16:18:00 0** 2021-04-24 16:19:00 0	2021-04-27 21:33:00 0 2021-04-27 21:34:00 4 2021-04-27 21:35:00 1 2021-04-27 21:36:00 2 2021-04-27 21:37:00 0 2021-04-27 21:38:00 0 2021-04-27 21:39:00 0 2021-04-27 21:40:00 1 2021-04-27 21:41:00 1 2021-04-27 21:42:00 0 2021-04-27 21:43:00 1 2021-04-27 21:44:00 0 2021-04-27 21:45:00 0 2021-04-27 21:46:00 0 2021-04-27 21:47:00 8 2021-04-27 21:48:00 0 2021-04-27 21:49:00 1 2021-04-27 21:50:00 0 2021-04-27 21:51:00 3 2021-04-27 21:52:00 0 2021-04-27 21:53:00 0 2021-04-27 21:54:00 0 2021-04-27 21:55:00 1 2021-04-27 21:56:00 0 2021-04-27 21:57:00 0 2021-04-27 21:58:00 0 2021-04-27 21:59:00 0 **2021-04-27 22:00:00 57** **2021-04-27 22:01:00 36** **2021-04-27 22:02:00 5** 2021-04-27 22:03:00 0 2021-04-27 22:04:00 1	2021-04-27 22:12:00 0 2021-04-27 22:13:00 0 **2021-04-27 22:14:00 40** 2021-04-27 22:15:00 0 2021-04-27 22:16:00 0 2021-04-27 22:17:00 0 2021-04-27 22:18:00 3 2021-04-27 22:19:00 1 2021-04-27 22:20:00 0 2021-04-27 22:21:00 0	2021-04-27 22:22:00 0 2021-04-27 22:23:00 0 2021-04-27 22:24:00 0 2021-04-27 22:25:00 0 2021-04-27 22:26:00 0 2021-04-27 22:27:00 8 **2021-04-27 22:28:00 47** **2021-04-27 22:29:00 11** **2021-04-27 22:30:00 0** **2021-04-27 22:31:00 54** 2021-04-27 22:32:00 0 2021-04-27 22:33:00 0 2021-04-27 22:34:00 0 2021-04-27 22:35:00 0 2021-04-27 22:36:00 0 2021-04-27 22:37:00 0 2021-04-27 22:38:00 0 2021-04-27 22:39:00 0 2021-04-27 22:40:00 0 2021-04-27 22:41:00 0

Figure 5.11: Chrome browser takeover result

Finally, we evaluate the simulated slow attack and malware execution experiments. The anomaly count is reported in figure 5.13 and time vs reconstruction error plot is presented in figure 5.12 for the slow attack. We clearly see a significant anomaly count per minute when system was under a slow attack and exceeds the proposed threshold count thus it is an anomaly. Model was able detect the malware execution. The notable anomaly counts were reported when the malware began it‘s execution. The anomaly count for the malware execution exepriment is reported in figure 5.15 and time vs reconstruction error plot is presented in figure 5.14

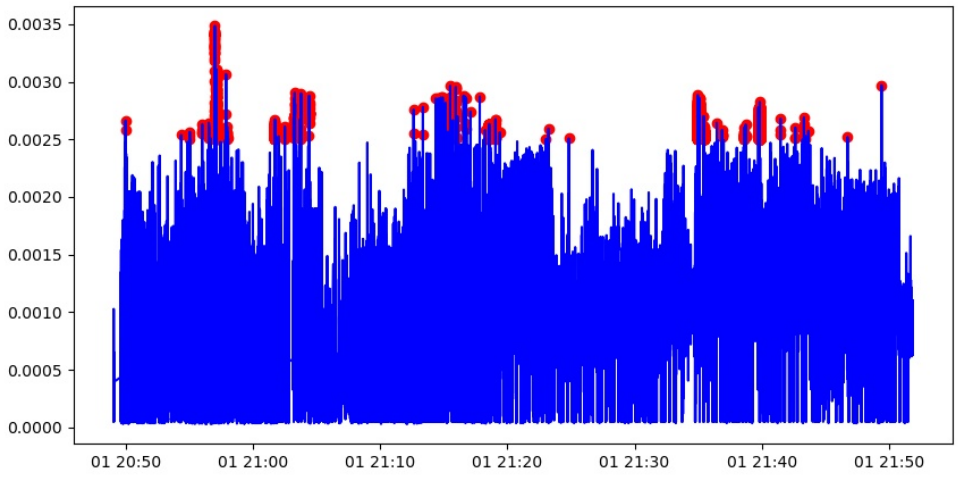


Figure 5.12: Slow attack: ElasticSearch takeover (20:57, 21:03, 21:15 & 21:39)

Date	Anomaly count per minute	Date2	Anomaly count per minute2	Date3	Anomaly count per minute3
2021-06-01 20:49:00	2	2021-06-01 21:10:00	0	2021-06-01 21:31:00	0
2021-06-01 20:50:00	0	2021-06-01 21:11:00	0	2021-06-01 21:32:00	0
2021-06-01 20:51:00	0	2021-06-01 21:12:00	2	2021-06-01 21:33:00	0
2021-06-01 20:52:00	0	2021-06-01 21:13:00	2	2021-06-01 21:34:00	622
2021-06-01 20:53:00	0	2021-06-01 21:14:00	6	2021-06-01 21:35:00	910
2021-06-01 20:54:00	1	2021-06-01 21:15:00	37	2021-06-01 21:36:00	6
2021-06-01 20:55:00	3	2021-06-01 21:16:00	6	2021-06-01 21:37:00	0
2021-06-01 20:56:00	76	2021-06-01 21:17:00	2	2021-06-01 21:38:00	29
2021-06-01 20:57:00	1225	2021-06-01 21:18:00	200	2021-06-01 21:39:00	5151
2021-06-01 20:58:00	1	2021-06-01 21:19:00	12	2021-06-01 21:40:00	0
2021-06-01 20:59:00	0	2021-06-01 21:20:00	0	2021-06-01 21:41:00	3
2021-06-01 21:00:00	0	2021-06-01 21:21:00	0	2021-06-01 21:42:00	3
2021-06-01 21:01:00	106	2021-06-01 21:22:00	0	2021-06-01 21:43:00	6
2021-06-01 21:02:00	9	2021-06-01 21:23:00	2	2021-06-01 21:44:00	0
2021-06-01 21:03:00	50	2021-06-01 21:24:00	1	2021-06-01 21:45:00	0
2021-06-01 21:04:00	18	2021-06-01 21:25:00	0	2021-06-01 21:46:00	1
2021-06-01 21:05:00	0	2021-06-01 21:26:00	0	2021-06-01 21:47:00	0
2021-06-01 21:06:00	0	2021-06-01 21:27:00	0	2021-06-01 21:48:00	0
2021-06-01 21:07:00	0	2021-06-01 21:28:00	0	2021-06-01 21:49:00	1
2021-06-01 21:08:00	0	2021-06-01 21:29:00	0	2021-06-01 21:50:00	0
2021-06-01 21:09:00	0	2021-06-01 21:30:00	0	2021-06-01 21:51:00	0

Figure 5.13: Slow attack: anomaly count per minute)

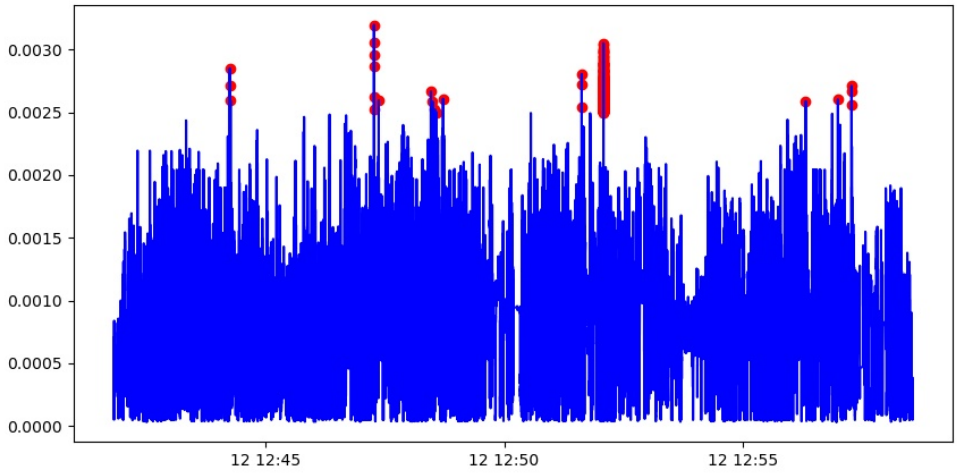


Figure 5.14: Malware execution at 12:52

Date	Anomaly count per minute
2021-06-12 12:41:00	0
2021-06-12 12:42:00	0
2021-06-12 12:43:00	0
2021-06-12 12:44:00	3
2021-06-12 12:45:00	0
2021-06-12 12:46:00	0
2021-06-12 12:47:00	7
2021-06-12 12:48:00	5
2021-06-12 12:49:00	0
2021-06-12 12:50:00	0
2021-06-12 12:51:00	3
2021-06-12 12:52:00	373
2021-06-12 12:53:00	0
2021-06-12 12:54:00	0
2021-06-12 12:55:00	0
2021-06-12 12:56:00	2
2021-06-12 12:57:00	3
2021-06-12 12:58:00	0

Figure 5.15: Malware execution: anomaly count per minute

5.4 Summary

We experimented with multiple fastText configuration settings and verified that fastText embedding provides a meaningful representation of the events. Events related to the process form a cluster in the embedding space; as we saw by using the t-SNE dimensionality reduction technique, we could visualize the embeddings. We also verified that increasing the dimension size of the embedding has less effect on the accuracy and anomaly detection than the n-gram setting of the fastText model.

Finally, the unsupervised anomaly detection model performed well. The model learned a typical sequence of events and reconstructed the events vector embeddings with less reconstruction error. We choose for anomaly detection, i.e., looking at the anomaly counts per minute windows and calculating the acceptable threshold worked in the context for Anomaly detection. We further examine the sequence windows, which reported the reconstruction error more than the threshold: the anomalous windows had events such as 1) accessing the 'meterpreter' class file 2) reverse TCP connection network event on port 4444 3) accessing windows registries where the information about system network configuration is kept. Thus, the model captures the anomalies and captures the actual anomalous events in the anomaly window. After setting the acceptable threshold on the anomaly window to 30, the model could also detect the attack on the google chrome process four out of four times. In figure 5.11 we report the anomaly counts for the Chrome takeover experiment. The model is also able to detect the slow attack and malware execution.

Chapter 6: Conclusion and Future Work

In this thesis, we studied the problem of Anomaly Detection in Windows OS. Many approaches use only a single source of data, such as a file, registry, or network, but they fail to capture the spatial locality between the events. For example, process 'P' usually first open system DLL file 'X' then check the windows registry value 'Y' for configuration and finally performs network activities with specific IP addresses and specific ports 'Z'. These events are enriched with semantic and syntactic information. The file path, registry path, process binary path, windows system DLL file names, IP addresses, etc., clearly have syntactic information. Thus, we developed a Windows kernel driver to tap into the file, registry, and network events.

Results show that we can apply the NLP technique like fastText sentence embedding to represent these kernel events and capture the hidden information present. We proposed the fastText sentence embedding approach and some text preprocessing techniques. LSTM based sequence Autoencoder model was trained with input as an embedding representation of these events. The proposed model is implemented and evaluated in this thesis.

The kernel driver helped us collecting the required dataset for training and experiments. We proposed the approach to calculating the threshold value for the

sequence window to be marked as an anomaly. The collected attack and attack-free datasets were evaluated on the Autoencoder model. The results were used for the F1 matrix and threshold calculation. The result shows that threshold value 30 yields optimal results with an F1 score of 98% for both attack-free and attack metric. The precision is 1 for attack-free metric, and recall is 1 for attack metric. Thus, resulting in addition to the acceptable system for Anomaly detection along with current other techniques.

One interesting future experiment might be testing the system with other Chromium-based web browsers and seeing if web browsing can generalize the browsers based on the web engine behavior. Further, validate how anomaly detection system behaves when any new application is installed and ran. This is expected to show up some Anomalies (due to unseen data in the past) in the system but is expected to be below the threshold count per minute. This can be a limitation of our system that we might have to retrain the model with the new data for the new application.

Moving the system goal to detect the actual malware families would be an exciting step to gain more insights. The trivial idea of exploding few malware samples of family 'X' and then verifying if the model can also detect other samples from the same malware family would be interesting. Also, promoting the current system to the real-time anomaly detection system would be a simple yet exciting next step. For example, how long does the system takes to detect an attack? Alternatively, how long does it takes to detect the malware when it was exploded?. Answers to these questions can provide more insights into the model. Since the model was able to generalize the multiple user behavior on the Windows platform, we believe the

system we proposed has the potential to overcome the problem of virus scanner and other systems such as inefficiency against the unknown attack and/or inefficiency against the specially crafted executable and process behaviour [32]

Appendix A: HoTSoS poster presentation

We participated in the 8th Annual Hot Topics in the Science Security (HoTSoS) symposium. We created and presented the poster on the online platform provided by them. In figure A.1, we present the poster developed for this work.

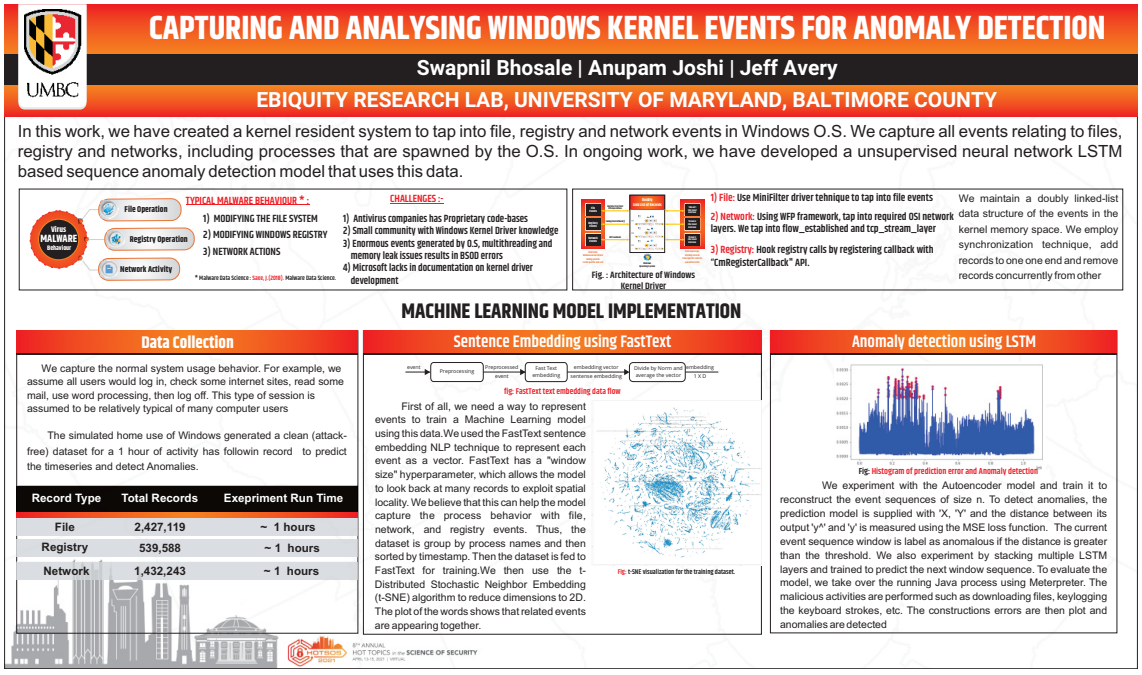


Figure A.1: Poster presented for HoTSoS conference

Bibliography

[1] J. Johnson. Global new malware volume 2020. https://www.statista.com/statistics/680953/global-malware-volume/#:~:text=As%20of%20March%202020%2C%20the,surpass%20700%20million%20within%202020, January 2021.

[2] Xiaoyan Sun, Jun Dai, Peng Liu, Anoop Singhal, and John Yen. Using bayesian networks for probabilistic identification of zero-day attack paths. *IEEE Transactions on Information Forensics and Security*, 13(10):2506–2521, 2018.

[3] Mamoun Alazab and MingJian Tang. *Deep learning applications for cyber security*. Springer, 2019.

[4] Salvatore Stolfo, Frank Apap, Eleazar Eskin, Katherine Heller, Shlomo Hershkop, Andrew Honig, and Krysta Svore. A comparative evaluation of two algorithms for windows registry anomaly detection. *Journal of Computer Security*, 13:659–693, 10 2005.

[5] Joshua Saxe and Hillary Sanders. *Malware Data Science: Attack Detection and Attribution*. No Starch Press, USA, 2018.

[6] Konstantin Berlin, David Slater, and Joshua Saxe. Malicious behavior detection using windows audit logs. In *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security*, AISec '15, page 35–44, New York, NY, USA, 2015. Association for Computing Machinery.

[7] Ammar Alazab, Michael Hobbs, Jemal Abawajy, and Moutaz Alazab. Using feature selection for intrusion detection system. In *2012 international symposium on communications and information technologies (ISCIT)*, pages 296–301. IEEE, 2012.

[8] Ansam Khraisat, Iqbal Gondal, Peter Vamplew, Joarder Kamruzzaman, and Ammar Alazab. Hybrid intrusion detection system based on the stacking ensemble of c5 decision tree classifier and one class support vector machine. *Electronics*, 9(1):173, 2020.

[9] Frank Apap, Andrew Honig, Shlomo Hershkop, Eleazar Eskin, and Sal Stolfo. Detecting malicious software by monitoring anomalous windows registry accesses. In *International Workshop on Recent Advances in Intrusion Detection*, pages 36–53. Springer, 2002.

[10] Regmon for windows - windows sysinternals — microsoft docs. https://docs.microsoft.com/en-us/sysinternals/downloads/regmon. (Accessed on 05/07/2021).

[11] Dima Rabadi and Sin G. Teo. Advanced windows methods on malware detection and classification. In *Annual Computer Security Applications Conference*, ACSAC '20, page 54–68, New York, NY, USA, 2020. Association for Computing Machinery.

[12] Cuckoo sandbox - automated malware analysis. https://cuckoosandbox.org/. (Accessed on 05/07/2021).

[13] Mamoun Alazab. Profiling and classifying the behavior of malicious codes. *Journal of Systems and Software*, 100:91–102, 2015.

[14] Yuhang Lin, Olufogorehan Tunde-Onadele, and Xiaohui Gu. Cdl: Classified distributed learning for detecting security attacks in containerized applications. In *Annual Computer Security Applications Conference*, ACSAC '20, page 179–188, New York, NY, USA, 2020. Association for Computing Machinery.

[15] Konstantin Berlin, David Slater, and Joshua Saxe. Malicious behavior detection using windows audit logs. In *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security*, AISec '15, page 35–44, New York, NY, USA, 2015. Association for Computing Machinery.

[16] Chani Jindal, Christopher Salls, Hojjat Aghakhani, Keith Long, Christopher Kruegel, and Giovanni Vigna. Neurlux: Dynamic malware analysis without feature engineering. In *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC '19, page 444–455, New York, NY, USA, 2019. Association for Computing Machinery.

[17] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[18] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*, 2016.

[19] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2017.

[20] Rajesh Bordawekar, Bortik Bandyopadhyay, and Oded Shmueli. Cognitive database: A step towards endowing relational databases with artificial intelligence capabilities. *arXiv preprint arXiv:1712.07199*, 2017.

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

[22] Lin Yang, Junjie Chen, Zan Wang, Weijing Wang, Jiajun Jiang, Xuyuan Dong, and Wenbin Zhang. Semi-supervised log-based anomaly detection via probabilistic label estimation. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1448–1460. IEEE, 2021.

[23] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[24] Penetration testing software, pen testing security. https://www.metasploit.com/.

[25] Vx vault. http://vxvault.net/. (Accessed on 05/07/2021).

[26] Virustotal malware report. https://www.virustotal.com/gui/file/504229eb947e4ae07f6cb408da3dde9ae2ae2996b4df208ab9d06558f39d2cf5/behavior. (Accessed on 05/07/2021).

[27] Virustotal. http://virustotal.com/. (Accessed on 05/07/2021).

[28] Akash Singh. Anomaly detection for temporal data using long short-term memory (lstm), 2017.

[29] Tolga Ergen and Suleyman Serdar Kozat. Unsupervised anomaly detection with lstm neural networks. *IEEE transactions on neural networks and learning systems*, 31(8):3127–3141, 2019.

[30] Pankaj Malhotra, 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.

[31] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[32] Ishai Rosenberg, Asaf Shabtai, Yuval Elovici, and Lior Rokach. Query-efficient black-box attack against sequence-based malware classifiers. In *Annual Computer Security Applications Conference*, ACSAC '20, page 611–626, New York, NY, USA, 2020. Association for Computing Machinery.