MALICIOUS CODE DETECTION: RUN TRACE ANALYSIS BY LSTM

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS OF
THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

MELİH ŞIRLANCI

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE
OF
MASTER OF SCIENCE
IN
THE DEPARTMENT OF CYBER SECURITY

JUNE 2021

Approval of the thesis:

MALICIOUS CODE DETECTION: RUN TRACE ANALYSIS BY LSTM

submitted by **MELİH ŞIRLANCI** in partial fulfillment of the requirements for the degree of **Master of Science in Cyber Security Department, Middle East Technical University** by,

Prof. Dr. Deniz Zeyrek Bozşahin
Dean, **Graduate School of Informatics** _____

Assist. Prof. Dr. Cihangir Tezcan
Head of Department, **Cyber Security** _____

Assoc. Prof. Dr. Cengiz Acartürk
Supervisor, **Cognitive Science Dept., METU** _____

Dr. Pınar Gürkan Balıkçıoğlu
Co-supervisor, **Cyber Security Dept., METU** _____

Examining Committee Members:

Assoc. Prof. Dr. Aysu Betin Can
Information Systems Dept., METU _____

Assoc. Prof. Dr. Cengiz Acartürk
Cognitive Science Dept., METU _____

Assoc. Prof. Dr. Sevil Şen
Computer Engineering Dept., Hacettepe University _____

Assist. Prof. Dr. Cihangir Tezcan
Cyber Security Dept., METU _____

Assist. Prof. Dr. Aybar Can Acar
Health Informatics Dept., METU _____

Date: 25.06.2021

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Melih Şırlancı

Signature :

ABSTRACT

MALICIOUS CODE DETECTION: RUN TRACE ANALYSIS BY LSTM

Şırlancı, Melih
M.S., Department of Cyber Security
Supervisor: Assoc. Prof. Dr. Cengiz Acartürk
Co-Supervisor: Dr. Pınar Gürkan Balıkçıoğlu

JUNE 2021, 67 pages

Malicious software threats and their detection have been gaining importance as a subdomain of information security due to the expansion of ICT applications in daily settings. A major challenge in designing and developing anti-malware systems is the coverage of the detection, particularly the development of dynamic analysis methods that can detect polymorphic and metamorphic malware efficiently. In the present study, we propose a methodological framework for detecting malicious code by analyzing run trace outputs by Long Short-Term Memory (LSTM). We developed models of run traces of malicious and benign Portable Executable (PE) files. We created our first dataset from run trace outputs obtained from dynamic analysis of PE files. The obtained dataset was in the instruction format as a sequence and was called Instruction as a Sequence Model (ISM). By splitting the first dataset into basic blocks, we obtained the second one called Basic Block as a Sequence Model (BSM). The experiments showed that the ISM achieved an accuracy of 87.51% and a false positive rate of 18.34%, while BSM achieved an accuracy of 99.26% and a false positive rate of 2.62%.

Keywords: Dynamic Analysis, LSTM, Malware Detection, Natural Language Processing, Run Trace

ÖZ

ZARARLI KOD TESPİTİ: LSTM İLE RUN TRACE ANALİZİ

Şırlancı, Melih
Yüksek Lisans, Siber Güvenlik Bölümü
Tez Yöneticisi: Doç. Dr. Cengiz Acartürk
Ortak Tez Yöneticisi: Dr. Pınar Gürkan Balıkçıoğlu

Haziran 2021, 67 sayfa

ICT uygulamalarının günlük hayatta giderek yaygınlaşması sebebiyle zararlı yazılım tehditleri ve tespiti, bilgi güvenliğinin alt alanı olarak önem kazanmaya devam ediyor. Zararlı yazılımlardan koruma sistemlerinin tasarlanmasında ve geliştirilmesinde en büyük zorluk tespit mekanizmasının kapsamı, daha spesifik olarak polimorfik ve metamorfik zararlı yazılımları etkin şekilde tespit edebilecek dinamik analiz methodlarının oluşturulmasıdır. Bu çalışmada zararlı kod parçalarını tespit edebilmek için LSTM ile run trace analizi yapan metadolojik bir framework öneriyoruz. Zararlı ve zararsız çalıştırılabilir dosyaların run trace çıktıları üzerine modeller geliştirdik. Çalıştırılabilir dosyaların dinamik analizinden elde edilen run trace çıktılarından ilk veri setimizi oluşturduk. Elde edilen bu veri setinde diziler asssembly talimatları formatındaydı ve "Instruction as a Sequence Model (ISM)" olarak adlandırıldı. İlk veri setini temel bloklara bölerek "Basic Block as a Sequence Model (BSM)" olarak adlandırdığımız ikinci veri setini elde ettik. Yapılan denemeler ISM modelinin %87,51 doğruluk oranına ve %18,34 yanlış-pozitif oranına ulaştığını gösterirken BSM modelinin %99,26 doğruluk oranına ve %2,62 yanlış-pozitif oranına ulaştığını gösterdi.

Anahtar Kelimeler: Dinamik Analiz, LSTM, Zararlı Yazılım Tespiti, Doğal Dil İşleme, Run Trace

To My Family

ACKNOWLEDGMENTS

First of all, I would like to thank my supervisor Assoc. Prof. Dr. Cengiz Acartürk for his guidance and support at every stage of my thesis and my co-supervisor Dr. Pınar Gürkan Balıkçıoğlu for her contributions and great support to complete this study.

Besides my supervisors, I would like to thank the members of our malware analysis research group, Deniz Demirci, Nazenin Şahin, and Özge Acar Küçük for their support in the scope of this study.

Lastly, I would like to thank my mother Meral, my father Mehmet, and my sister Melike for their support through every moment of my life.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ALGORITHMS

ALGORITHMS

LIST OF ABBREVIATIONS

ANN	Artificial Neural Network
API	Application Programming Interface
AV	Antivirus
BSM	Basic Block as a Sequence Model
CNN	Convolutional Neural Network
DL	Deep Learning
ISM	Instruction as a Sequence Model
LSTM	Long Short-Term Memory
ML	Machine Learning
NLP	Natural Language Processing
PE	Portable Executable
RNN	Recurrent Neural Network

CHAPTER 1

INTRODUCTION

As today's information systems are constantly evolving, they are constantly attacked by people with malicious intent or different motivations. Since the development of the systems makes the attack surface bigger, the number of attacks is increasing each day. One of the main attack methods is malicious software, i.e. malware, which includes specific types such as viruses, worms, and trojans. Malware can be used to attack operating systems and applications, and to cause damage at both personal and corporate levels. Usually, by exploiting a vulnerability in computer systems through malicious software, the availability of real-time systems is targeted and valuable data is rendered unusable. The spread of this type of malware is becoming faster due to the increased connectivity of new devices such as computers, smartphones, smart televisions, and devices in the home area network, i.e. IoT devices. In addition, the increase in the use of mobile devices encourages malware authors to focus on mobile operating systems and applications, which will eventually lead to an expansion of malware detection and mitigation methodologies into novel domains of application.

Over the past decade, the number of new malware obtained daily has been increasing. According to the IT Security Institute, AV-TEST statistics, 350,000 new malware and unwanted applications are examined and classified every day (as of August 2020) [1]. In addition, the online malware analysis service Virustotal reports statistical data on files submitted for analysis [2]. According to these statistics, the average daily number of files sent for analysis was 2 million in the seven days between July 28 and August 4, 2020. The average number of unique files submitted was 1.6 million, of which 800,000 were detected daily by one or more AV (Antivirus) engines. Also, another important point drawn from the statistics for the 7-day period is that 4.4 mil-

lion of the files sent during the week were x86 Windows operating system executable files. The daily amount of submitted suspicious files is continuously increasing. This situation brings the need for a richer set of methodologies for malware analysis. In the present study, we aim at enriching malware analysis methodology by proposing a framework for an automated run trace output analysis, which is a recent challenge in cyber security defense systems.

Traditional methods can no longer perform well on polymorphic and metamorphic types of malware recently. Polymorphic malware uses encryption to be hidden from AV products. Instances from the polymorphic type of malware are kept encrypted on disk so AV products can not detect them with static signature scanners. On the other hand, metamorphic malware has the ability to change its look further on each execution. So, instances from the metamorphic type of malware are even better at hiding from AV products since they have different looks even when they are executed. Because such malware is difficult to detect by signatures generated manually, the focus of the research has shifted to the use of Machine Learning (ML) in automated malware detection systems.

Basically, malware analysis can be classified into two main categories, namely static analysis and dynamic analysis. Static analysis aims at gathering information about a suspected file without executing it to decide whether it is malicious or not. During static analysis, analysts often use a disassembler tool and investigate the assembly code, imported functions, and strings. On the other hand, in dynamic analysis, the suspected file is executed and information about likely malicious operations is collected. During dynamic analysis, the flow of the program is traced and the function calls, as well as the parameter values in registers, are examined by the malware analysts. As in malware analysis, malware detection research by using machine learning and deep learning focuses on similar data collected from files such as assembly code, opcodes, API (Application Programming Interface) calls, control flow graphs, and metadata from file headers, e.g. [3–5].

Machine learning and deep learning techniques are used to detect malware in various fronts, such as conducting binary classification of software as benign or malicious, as well as classifying malware into known types such as virus, worm, and trojan or

known malware families. In our study, we focused on deep learning methods and then used a specialized type of Recurrent Neural Network (RNN) called Long Short-Term Memory (LSTM) proposed by Hochreiter and Schmidhuber [6]. We approach malware detection from the perspective of Natural Language Processing (NLP) by developing and testing models that process run traces of malicious and benign software.

We propose an approach, which we also published in an academic paper [7], to malware detection that focuses on run trace components in a dynamic analysis framework. Recently, there exists a limited number of studies using dynamic analysis with assembly instructions. To the best of our knowledge, there is no study that uses the run trace output for malware detection. In the present study, we aim at exploring the detection performance of dynamically collected data. We report an investigation of run trace data collected at runtime of PE (Portable Executable) files. In particular, we used a semi-automated process to collect run trace output from PE files. First, we created the run trace dataset as an instruction per line, viz. instruction as a sequence. Then, we converted it into a different form as a basic block per line, viz. basic block as a sequence, thus, we obtained a second dataset. After creating our datasets, we chose LSTM as the machine learning technique. As reported in the literature [8] and [9], LSTM shows better performances than its predecessor, RNN (also among customized versions of RNN). We called our proposed methods, "ISM (Instruction as a Sequence Model)" and "BSM (Basic Block as a Sequence Model)". We aim to compare ISM and BSM based on the evaluation results. The methodological details are presented in the following sections.

1.1 Motivation and Problem Definition

We decided to focus on the malware detection problem since it is an increasingly serious threat to information systems. Language modeling and text classification approaches can be useful to solve this problem so we adapted them into malware detection context. We used Long Short-Term Memory (LSTM) to do binary classification on dynamically collected assembly instructions.

Deep learning methods provide more resistance against changes in data since feature extraction is automatically handled by neural networks instead of manual feature extraction in machine learning. In addition, modeling and classification of natural languages by employing various types of RNN were shown to achieve high accuracies in previous works [8] [9]. Among standard RNN and its specialized versions, we preferred to use LSTM since it provides more robust architecture during the training phase by better solving vanishing and exploding gradient problems of standard RNN architecture. Because of the structural and semantic similarities between a natural language and assembly language, to be able to detect malicious software we decided to create language models of assembly instructions from malicious and benign executable files as two different natural languages. Also, we worked on two different forms of the same assembly instruction data to find out the effect of structural and semantic differences of assembly code on the detection capability and whether handling data differently achieves better results.

1.2 Research Questions

The research questions of this study are presented as follows. Our study investigates that is it possible to model dynamic assembly output of malicious and benign PE files to classify those code pieces as natural languages. In addition, the present study investigates the two units in assembly language, which are instruction and basic block, to identify which one is better to use when performing language modeling task on assembly code.

The thesis is organized as follows. In Chapter 2, first, we present the background to give an idea about the concepts related to our study. Then, we present the relevant studies in the topics including malware detection and language modeling. In Chapter 3, we describe our approach, datasets, the proposed models (viz. ISM and BSM), LSTM train and test pipeline, and parameters as well as the setup of the environment used for training and testing. In Chapter 4, we report the results, a comparison of the models, and a discussion of the results. Finally, in Chapter 5, we present a conclusion, the limitations of the study, and the future work.

CHAPTER 2

BACKGROUND AND RELEVANT WORK

2.1 Malware Analysis

2.1.1 What is Malware?

As a word, malware comes from the combination of "malicious" and "software" words. In general, any piece of code, which is capable of doing something malicious on information systems, is called malware. Those malicious operations include a variety of subjects e.g. removing files from a personal computer, getting access to a system, using hacked systems for other attacks. Even if there are not strict categories to label malware, according to their purpose and functionality they can mainly be divided into such categories; virus, worm, trojan, adware, spyware, rootkit, and ransomware [10].

To meet the needs in malware detection, first of all we need to have an idea about the concept of malware. Malware is considered as a file and is often treated as an integral part when developing detection methods. Like every other software, malware consists of codes, which commands the operating system to perform a function. However, as a difference from software, malware has malicious code parts or consists of completely malicious codes, which are employed to achieve a malicious purpose. Therefore, instead of thinking of malicious files as a whole consisting of one large piece, we can focus on smaller pieces of malicious files to develop more efficient detection mechanisms.

Consequently, we can approach the malware detection problem by thinking of malware consisting of bits of code that contain malicious code fragments. This thinking

provides us different perspective while developing new detection methods and even may provide such an opportunity to detect and block those malicious code pieces in real-time use. In addition, such an approach can be applied to each malware type such as virus, worm, and ransomware since each one of them consists of commands that are executed on the CPU through an operating system.

2.1.2 Malware Analysis Methods

Static Analysis

Static analysis is one of the main malware analysis phases, which includes a series of information-gathering operations. In malware analysis, the static analysis techniques help analysts to gain insight into the malicious software, to understand which parts require more focus, and to decide where to start. The important point in this type of analysis is that analysts are limited to use the tools which collect data from malicious software without executing it. The information gathered in this phase includes file type, strings, Portable Executable (PE) header information, function imports and exports, and packer information if the file is packed as well as some other details by antivirus scanners [11].

In addition to information gathering from such tools, analysts statically examine the codes of the malicious software, that is obtained by disassemblers. However, understanding what the malware is doing for what purposes can make static analysis difficult due to the unavailable source code and obfuscation techniques. Malware authors use obfuscation techniques such as encryption and self-code modification to make it harder for analysts to analyze malicious software. Also, since the source code is not available, analysts can only access low-level language disassembled code which is much more difficult to understand compared to high-level language source code.

Dynamic Analysis

Dynamic analysis, which is generally done after static analysis, is the phase of malware analysis in which analysts have more understanding of the suspected file. In dynamic analysis, analysts have the ability to monitor the behaviors of malicious files by executing them. Thus, a dynamic analysis should be performed by isolating the

run file in a controlled environment such as sandboxes or virtual machines in order to avoid possible infections. At this stage, analysts are able to get various information about the malicious file including changes in the file system, the processes running on the operating system, and the network traffic [12]. With this information, analysts become able to understand the characteristics of the file and get some idea of what the file is intended to do. In addition to that, analysts examine the code of malicious files by executing them on a debugger to get more insight into them.

However, dynamic analysis also has some difficulties. The major drawback of this phase is that some malware can behave in a different way when they are executed in a controlled environment like virtual machines. Besides, dynamic analysis requires a serious amount of time and some resources but there is no guarantee to achieve a successful result.

2.1.3 Polymorphic and Metamorphic Malware

Polymorphic and metamorphic malware types emerged to satisfy the need of keeping malicious software hidden on target machines. This functionality made them popular among malware authors and led to be used increasingly. These types of malware are especially used against traditional signature based detection methods since they have the ability to avoid the detection mechanism employed by such detection methods.

Polymorphic malware employs encryption methods to avoid detection by keeping itself hidden from AV products. Usually, it has decryption modules built-in, so it is stored on disk in encrypted form. This software only decrypts and executes malicious parts of it at runtime using decryption modules. When such malware is on the disk, it has a benign appearance as perceived by the host computer and therefore can bypass static AV scanners without changing its appearance [13].

Metamorphic malware uses a series of techniques on the low-level programming language to change its code's look further on each execution. It uses obfuscation techniques to change the appearance and obtain functionally equivalent versions of itself. Since a metamorphic engine produces a malware file that does the same job in each execution but looks different from the previous one, it becomes virtually impossible

to detect by signature-based methods [13].

2.1.4 Malware Detection Methods

Signature-Based Methods

As a word, in malware detection, signature means footprint or pattern extracted from malicious software to protect information systems from this threat. Those signatures that are unique features for each file are extracted by malware analysts and consist of byte sequences from the malicious software analyzed. Signature-based methods are used widely since the early times of malware threats on systems. Being a fast and accurate way of detection made signature-based methods popular and highly preferable [14].

These signature-based malware detection methods work well against known malware variants but have some problems against unknown variants [14]. This was not a problem in the early times of malware detection since the amount of newly created malware is limited. However, there are some factors, which cause a serious increase in the number of new malware variants, such as more people attracted to creating malicious software and more important than that polymorphic and metamorphic malware introduced to the scene. Even if the amount of newly created malware by authors is limited, the number of variants of the same malware created by polymorphic and metamorphic engines are huge. So, nowadays satisfying the amount of manpower and time to create signatures for the huge amount of unique malware variants is not possible and practical. So, even if the signature-based detection mechanisms still are used by antivirus scanners, the increasing amount of research focuses on new more efficient detection methods, particularly based on malware behaviors and machine learning techniques.

Behavior-Based Methods

Behavior-based malware detection methods focus on the behaviors of potentially malicious files. Instead of working on one malicious sample to create a unique signature, in behavior-based methods, malware analysts try to generate behavioral features belonging to malicious software, which can be used to detect malware with similar

behavioral activities. The behaviors used to detect malware include various kinds of activities such as registering for autorun, attempting to disable security controls, attempting to discover the environment executed in, trying to access files on the system, attempting to download and install other software [15].

This type of detection method shows better performance against polymorphic and metamorphic malware since each new variant will have similar behaviors even if they have different looks [15]. This is the most important advantage of behavior-based methods since they have the ability to detect the variants which are not possible to detect by signature-based methods. However, the drawback of behavior-based methods is that they require a high amount of scanning time since they use the dynamic analysis approach. Also, they suffer from a relatively high amount of false positive rate [16].

Machine Learning and Deep Learning Based Methods

The increase in the number of new malware variants detected every day makes it hard to find enough manpower to analyze all those new variants. This growing threat caused by malicious software against information systems requires to development of automated detection methods with any or less human intervention. For this purpose, since the last decade, the focus of academic studies in malware detection has shifted from traditional methods to machine learning classification methods and, in the last few years, from the machine learning classification methods to deep learning neural networks.

Before the machine learning classifiers, researchers were trying to extract signatures in form of mostly string or graph to identify a specific variant or a malware family. After machine learning was introduced to the scene, instead of creating signatures, the data collected from malicious software were used to do feature extraction and those features were given to the machine learning classification and clustering algorithms to classify malicious and benign software. The machine learning classification algorithms used to detect malware in academic studies mostly include Logistic Regression, Naive Bayes Classifier, Support Vector Machine, Decision Trees, Boosted Trees, and Random Forest [17]. In the malware detection studies with the focus on machine learning clustering, the k-means clustering algorithm is the most used ma-

chine learning clustering algorithm. In these studies, various types of data from malicious files are used during the feature extraction phase, including opcode sequences, API calls, system calls, and Control Flow Graphs (CFGs) [17].

The malware detection methods based on machine learning classification and clustering algorithms are not completely automated because of requiring human control over feature extraction. At this point, deep learning techniques, in other words, neural networks, which are thought of as a subcategory of machine learning became an alternative since they can learn from data without feature extraction. In addition to reducing human intervention and making the process more automated, deep learning techniques show better results in some cases while showing similar performances with machine learning classification techniques in some other cases. In malware detection studies using deep learning techniques, Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN) are the most preferred neural network types. CNNs are used for image recognition so, in those studies employing CNNs, the data from malware are converted into images. RNNs are mostly preferred for text classification tasks since they show better performance on sequential data. So, the data collected from malware are put into a sequential format to use on RNNs. As it is in machine learning studies, the data collected from malicious files including opcode sequences, API calls, system calls, and Control Flow Graphs (CFGs) are used to train a neural network and to detect malware.

2.1.5 Obfuscation Techniques

Obfuscation techniques are used by malware authors to keep their malware hidden from antivirus scanners and to make the malware analysis process harder if the file is detected. The purpose of avoiding analysis and reverse engineering is to keep the internals of malware secret as much as possible to avoid signature generation that can be used in detection mechanisms. There are various obfuscation techniques used for this purpose and the most used ones will be explained in the following.

Packing: In some cases, malware is packed completely by a packer and it is unpacked during runtime. This type of obfuscation avoids easy access to malware's codes through disassembling. However, it is not an advanced method since it is not

hard to specify the packing algorithm and do an unpacking operation [18].

Encryption/Decryption: Encryption is another technique used by malware authors to make their malware's code unreadable during static analysis. For this purpose, a variety of encryption algorithms can be used. The encrypted malware is decrypted at runtime via the packer's decryption module, and the decrypted code is placed in memory. This operation can be done by malware analysts to decrypt the code and examine the suspected file [19].

Exclusive Or (XOR) Operation: The binary operation, exclusive or, is used as an obfuscation technique, which is one of the basic and frequently preferred techniques. By applying XOR operation on binaries, the look of some parts of the code or the entire code of the program can be easily changed [20]. Since the cost of the XOR operation is low for the CPU, this method is widely used by malware authors. In this way, some important parts of code such as URL strings are kept hidden from malware analysts during static analysis. Also, in the cases that the XOR operation is partially applied to the code, it becomes harder to find the specific part of the code to access its actual look.

Dead Code Insertion: In this technique, malware authors put some functionally unnecessary code into their malicious software. In this way, since those parts are random unrelated codes, they cause malware analysts to confuse while trying to understand the functionality of the malware during static and dynamic analysis [19].

Instruction Changing: This obfuscation technique aims to replace some instructions with their functionally equivalent ones [18] [19]. There are no functional differences in malicious software when this technique is applied, but just the look of its instructions seem different. So, instruction changing makes it harder for analysts to understand the codes of malware during analysis.

Program Flow Changing: Program flow is another important information source during malware analysis. It helps malware analysts to understand the intent of the suspected file. In this technique, a series of branch instructions are inserted into malware without disrupting its functionality [18]. Some of these branch instructions are never taken while some others are branching to the dead code pieces. So, malware

authors employ this technique to make it seriously harder for analysts to understand the intent of their malware.

2.2 Neural Networks

In the following subsections, first we explain classical neural networks called Artificial Neural Networks (ANNs) which are rooted in the 50s [21]. Then, we introduce two new type of neural network architectures, RNN and LSTM, mostly used for language modeling tasks, which are based on the idea of deep learning that carries the classical ANNs one step further by building a layered structure with the ANNs.

2.2.1 Artificial Neural Networks (ANNs)

Artificial Neural Networks (ANNs), generally called just neural networks, are a subcategory of machine learning, which was developed inspired by biological neural networks [21]. The main unit in neural networks are nodes that are designed by the inspiration of neurons in the human brain. However, a node's functionality is way simpler than a neuron's functionality in several different manners.

Each node in a neural network keeps a numerical value named weight, which is used to make a prediction on a given input. The weights are updated depending on whether a prediction is right or not. To update weights in a neural network, a loss value is calculated according to the output of the neural network and the effect of the loss value spreads nodes through backpropagation by updating the weights of nodes [22]. In this way, the operation referred to as learning is performed.

There are various neural network architectures that are specialized to do different tasks. For example, Convolutional Neural Networks (CNN) show better performance on image data in the course of pattern and image recognition tasks. On the other hand, Recurrent Neural Networks (RNN) perform better on sequential data to achieve language modeling and speech recognition tasks. Thus, it can be said that the performance and success of those networks vary according to the task on which they are performing.

2.2.2 Recurrent Neural Networks (RNNs)

Recurrent Neural Network (RNN) is a special type of neural network. The important part that made RNN distinct and successful compared to its predecessors is its sequential memory [23]. In the design of an RNN node shown in Figure 2.1, in addition to the input of the current step, a node gets the hidden state of the previous step as an input to calculate the current step's output. So, this connection between sequential steps, including the previous step and the current step, provides memory to the nodes in RNN, even if it is short-term memory.

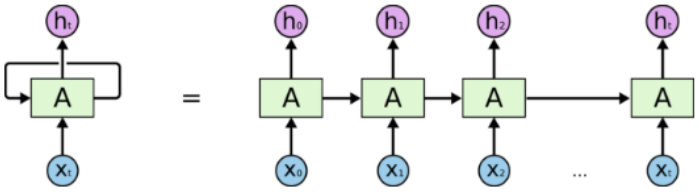

Figure 2.1: The design of a basic RNN node [24]

As the RNN goes through more steps, the effect of information from previous steps will decrease and even be lost because of the vanishing gradient problem, which causes RNN to have short-term memory. A gradient is a calculated value and carries information used during the update of node weights. When the gradient becomes smaller, the parameter updates in nodes become lesser through the first layers of the neural network, which causes the nodes to learn nothing. This short-term memory, because of vanishing gradients, decreases the performance of RNNs and makes them less effective on long sequence data. So, to solve such problems in RNN, new neural network architectures such as Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) are developed by modifying and improving the basic RNN architecture [25].

2.2.3 Long Short-Term Memory (LSTM)

Long Shor-Term Memory (LSTM) is a specialized RNN architecture and the most important feature of this advanced architecture is solving the vanishing gradient problem or at least decreasing the effect of the vanishing gradient problem on training perfor-

mance. Similar to RNN, nodes in an LSTM neural network get hidden states of the previous step. However, a common LSTM unit, node, has an improved structure compared to RNN, which is the main factor providing long-term memory by decreasing the effect of the vanishing gradient.

A common LSTM unit gets an input value and generates an output value. During this operation, it uses two values, including the generated output value of the current cell and the cell state value of the previous cell that will be explained in the following paragraphs, transferred by the previous step. An LSTM unit was designed to carry out the following three tasks.

- Forget unwanted information in the current cell state through the forget gate
- Add new information to the current cell state through the input gate
- Create output of the current cell state through the output gate

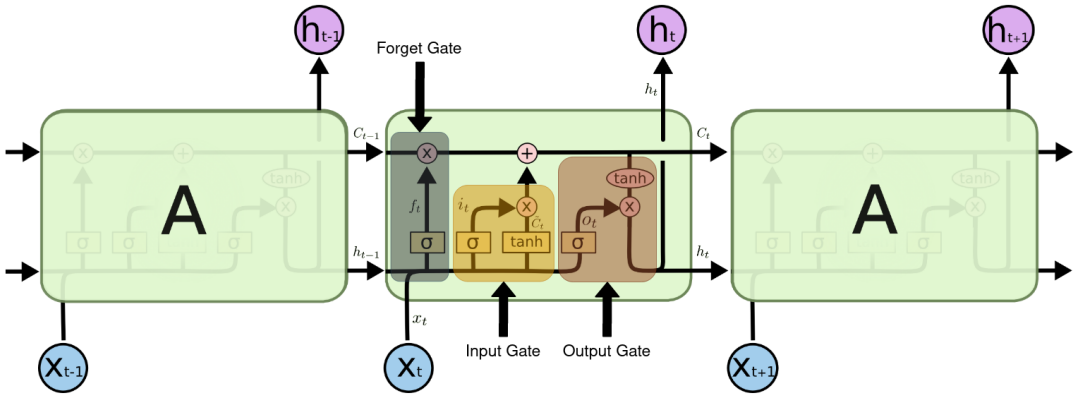

Figure 2.2: The interior design of a common LSTM cell [24]

The interior design of a common LSTM unit is shown in Figure 2.2. On the left side of the cell, the input of the current cell X_t and the output of the previous cell h_{t-1} are given to a sigmoid function to create an output f_t between 0 and 1. Then, this value, f_t, is multiplied by the previous cell state C_{t-1} to update and create the current cell state. The cell state is a value that flows through cells to carry information among them. This multiplication operation specifies which information will be forgotten and how much will be remembered in the next cells so this part of the unit is called the forget gate [26].

At the middle of the cell, there are one sigmoid and one tanh function whose outputs are multiplied. In this part, the sigmoid function again gets the input of the current cell X_t and the output of the previous cell h_{t-1} as input. As a difference from the sigmoid function in the forget gate, the output value of this sigmoid will be used to specify which value will be newly added to the current cell state. The tanh function creates an array of candidate values that will possibly be added to the current cell state. By multiplication of the output of sigmoid i_t and the output of tanh $\tilde{C}_t$, the values that will be added to the current cell state are determined out of the candidate values. With the help of add operation, the previous cell state C_{t-1} that was updated by forget gate is updated again with the new information from input to create the final current cell state. Thus, this part of the LSTM unit is called the input gate [26].

At the right part of the cell, there are a sigmoid and a tanh function that are used to specify the output value of the current cell. The tanh function gets the updated cell state, which can be called the current cell state C_t, to create the output value of the current cell between -1 and 1. On the other hand, the sigmoid function creates an output O_t which will be used to decide what parts of the current cell state will take place in the output of the current cell h_t. This part of the LSTM unit is called the output gate [26].

In summary, the current LSTM cell gets the output and the cell state of the previous cell in addition to the input of the current cell and generates the output of the current cell by updating the cell state. This sequential interior design of the LSTM architecture allows working on data consisting of long sequences with better performance, unlike basic RNN architecture.

2.3 Natural Language Processing

Natural Language Processing (NLP) is a subfield of artificial intelligence and linguistics. The main objective of NLP is to apply artificial intelligence methods to natural language to build machines that can understand the natural languages used by humans to communicate with each other. The NLP tasks include text classification, language translation, grammar check and all of them requiring building a language model to

achieve those objectives [27].

The evaluation of this field of study includes several steps, including linguistics, computational linguistics, statistical NLP, and NLP with neural networks, up until the point it reached today. In the beginning, in linguistics, there were formal methods based on mathematical rules to be able to understand and model natural languages without computational power. In the following times, computers were introduced to the field of study, linguistics, and it became popular as computational linguistics with the help of increasing computational power and large data. The data-driven methods led to the shift from classical rule-based methods to statistical methods [28].

The rise of machine learning and its applications to NLP tasks was an important point in this field of study. NLP tasks require knowledge in a variety of subjects including syntax, semantics, morphology, and pragmatics [29]. Understanding and modeling a natural language were only possible with the perception and processing of all of the knowledge together. Machine learning methods were a very good fit to overcome such a challenge since their main feature to extract and obtain such knowledge from data [29]. Thus, machine learning and statistical methods became dominant in the field and as a result, NLP also was called statistical NLP.

Today, deep learning neural networks provide a potent and effective environment for NLP tasks, which is powerful and successful compared to traditional machine learning techniques. For this purpose, a variety of neural network architectures are applied such as Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs). Recently, among those architectures, standard RNN and specialized RNN architectures show the most successful results [30]. As it is explained in previous sections, RNN architectures were designed to perform better on specifically sequenced data such as natural languages. In addition to extracting syntactic relations, RNNs are able to extract the semantic relations in data, which is an important and very useful feature to use RNN architectures on NLP tasks, particularly language modeling and text classification tasks.

Another important factor in such NLP tasks is the representation of the words (also called tokens). Word Embedding is one of the popular techniques used for word representation in NLP tasks. Word Embedding technique includes the vectorial represen-

tation of words commonly used to work on syntactic and semantic relations between words that come up together in a text from natural languages. Previously, count-based methods and vector representations like one hot encoding were commonly used but such representations produces sparse and high dimensional vectors that require so many resources to manage. Instead of approaches based on high dimensional sparse vectors, word embeddings consist of low dimensional dense vectors, which makes them computationally advantageous [31].

In this representation technique, each word in text corpora has a multiple-dimensional vector that consists of real numbers. The values in vectors are calculated by passing through sequenced data and looking at words coming up together in a specified window frame. So, the real numbers in a word's vector are formed according to the relations between the word and other words in the dictionary. By this representation, in addition to syntactic rules, a neural network can catch the semantic in a sentence from a natural language.

2.4 Relevant Work

The major challenge in today's cyber security strategies is that malware developers continuously update their methodologies, thus generating novel malware types, which are difficult to detect by the automated analysis tools. In particular, the integration of artificial intelligence and machine learning into mitigation techniques aims at developing malware detection systems with high accuracy, low false positive rates, and best performance. A review of the literature reveals three major features that have been employed for the development of automatic malware detection systems, namely opcode[1] frequency and sequence, Application Programming Interface (API) calls, and Control Flow Graphs (CFG). In our study, we examine assembly instructions obtained from run trace outputs of PE (Portable Executable) files.

Opcode frequency and sequence, which are usually obtained from the static analysis processes, comprise the backbone of any program code syntax. Therefore, they can be used as features for malware detection. The opcode sequences provide valuable

[1] In the present study, we use the term "opcode" to mean a single instruction that can be executed by the CPU.

information about *semantic* aspects of the program codes (as described within the framework of word embedding models employed for natural language processing).

Since the past decade, the study of opcodes for malware analysis has been subject to various methodological analyses. For example, Bilar [32] performed the extraction of common and rare opcodes from PE files using descriptive statistics, specifically to classify certain types of malware such as trojans and worms. Santos et al. [33] studied the incidence of opcode sequences. They investigated the relationships among the opcodes and used statistical information to detect variants of known malware families.

The use of machine learning for malware detection and classification has been popular since the past decade on various fronts. For example, in [34], the frequency of opcodes in malicious and benign software has been used as the main feature to the ML model, which was obtained from the assembly output of executables. The ML methodologies included Support Vector Machine (SVM), Random Forest (RF), Decision Tree (DT), and BOOSTING, among others, for classifying executables as malicious or benign. In addition to ML modeling by independent opcode features, Shabtai et al.'s investigation of n-gram opcode sequences has enriched automatic detection by introducing semantic aspects of opcode analysis that go beyond frequency statistics in [3]. In particular, the studies since the past decade have employed Term Frequency (TF) and Term Frequency with Inverse Document Frequency (TF-IDF) as model features. Also, the classification algorithms used in the research included SVM, Logistic Regression, Decision Trees and Random Forests, Artificial Neural Networks, Naive Bayes, and their boosted versions. The proposed method achieved 96% accuracy with a machine learning classifier, Random Forest. [35] is another study where features were extracted from *n*-gram opcode sequences and used in five machine learning classification algorithms to detect ransomware and classify ransomware families. The highest accuracy rate in this study was 91.43%. Those studies revealed high accuracy values. For example, the best accuracy in [34]was 97% with the RF algorithm. In [36], Euh et al. have focused on static feature extraction from malicious files. The obtained set of features, including opcode *n*-gram, API calls, window entropy map, were evaluated on several tree-based ensemble models such as XGBoost, AdaBoost, Random Forest. The highest average of those feature sets was 92.5 with XGBoost. Nevertheless, the feature selection process requires preliminary

steps for input data modeling, which may result in a loss of robustness, as well as having a limited scope for handling obfuscation and novel malware variants. The LSTM approach and similar approaches have the advantage of learning patterns in data by adapting the changes, which makes them robust and easy to maintain. In the present study, we employed the DL approach as a complementary approach to the previous works that have been conducted by employing classical ML algorithms.

Another major feature that has been employed in malware detection is API calls in Portable Executables (PEs). API calls can serve as a clue that may facilitate the investigation of the behavior of the PEs. The API calls are usually handled in two main forms: sequence (string) and graph. Since the past decade, several methods have been used to classify the API sequences and graphs, including string similarity, graph similarity, and machine learning classifiers. For example, in [37] the focus was on API call graphs. Since the graph matching causes problems while graphs are growing, the call graphs were converted into a new graph type called *code graph* and then the similarity between code graphs was measured using a predefined method. The proposed method achieved a 91% detection ratio which is a relatively low accuracy rate considering the amount of preprocessing, including collecting APIs, generating call graphs, and converting them into code graphs. A similar approach was proposed for API call sequences [38]. DNA sequence alignment algorithms were adopted to explore API call sequence patterns, and these patterns were used to detect malware, even their new unknown variants. The accuracy rate achieved in this study was %99.9. However, this proposed method suffers resources and time required by the DNA sequence alignment algorithms. In [39], Cheng et al. aimed to detect new malware variants by clustering malware families. They used API call dependency graphs of malware samples from the same family to create the family dependency graph of each family in their dataset (including 6 different families). The accuracy rates of the proposed method for each family varied between 88% and 98%, and the average accuracy rate was 92%.

The application of DL methods has been proposed by numerous studies. For example, Pascanu et al. implemented a two-step approach consisting of feature extraction and classification in [4]. They first employed Echo State Networks (ESNs) and Recurrent Neural Networks (RNNs) to extract features from API call sequences. Next,

at the classification step, there exists Logistic Regression and Multi-Layer Perceptrons with Rectifier Units. In contrast, Kolosnjaji et al. [40] used system call sequences to classify malicious files as malware families. They proposed a combined architecture of convolutional and recurrent LSTM layers to achieve the best results at classification. With their final model, they got an 89.4% average accuracy rate while classifying samples into families. In [41], there were three different malware classification architectures. Two of these were based on language modeling built on specialized RNN architectures, LSTM and Gated Recurrent Unit (GRU). The collected system call sequences were used to independently create language models with LSTM and GRU. The third architecture based on Character-level Convolutional Neural Networks (CNN) was also proposed by the authors as a method of classifying malware. The model with the best performance of the three was observed as LSTM. Unlike the previous API call studies, in [42] and [43], the API calls and the input parameters used during the calls were worked together. In [42], an LSTM model proposed by other researchers was expanded to categorize files into malicious and benign categories. Zhang et al. [43] extracted features from API calls and their associated parameters. The API data was passed through multiple Gated-CNNs to select important and relevant information. Then the output of Gated-CNNs was concatenated and given bidirectional LSTM to learn patterns in API calls. The highest accuracy rate in this study was 95.33%.

There are several recent studies in which deep learning techniques are employed to detect malicious software by focusing on opcode and assembly code. In [44], Khan et al. focused on cancer prediction and malware detection tasks together by using Convolutional Neural Networks (CNN). While collected X-Ray images were being used for cancer prediction, opcode pictures were generated by opcode sequences of malicious and benign binary files for malware detection. They have done each classification task on 4 different ResNet models and while the best accuracy for cancer prediction was 98%, it was 88.36% for malware detection. In [45], Khan et al. investigated GoogleNet and 5 different ResNet models by using images produced from opcodes of binary files. Histogram standardization enlargement and disintegration techniques were used to upgrade images to make the differences between malicious and benign opcode images easily detectable. The accuracy rate of GoogleNet was

74.5% and the best accuracy rate among ResNet models was 88.36%. In [46], Kumar et al. employed Convolutional Neural Networks for the classification of malware opcode images. The accuracy rate of correctly classified binary files was 98%. However, Convolutional Neural Networks are not strong enough against small changes in images as shown in [47] and [48]. Thus, obfuscation techniques used by malware can easily change the images generated from malware opcodes, which may cause a problem on this type of detection methods. In the present study, we preferred to apply text classification approaches which are more resistant against changes in data.

Furthermore, in [49], Jahromi et al. proposed a stacked LSTM method with pretraining of the neural network to avoid random problems caused by random initialization. The final proposed LSTM model, consisting of four layers, evaluated on 6 different malware datasets including static and dynamic features of Windows, Android, and IoT malware. One of the datasets was statically collected opcodes of Windows binaries and the proposed method achieved an 88.51% accuracy rate on this dataset. As a difference from this study, we investigated the success of dynamically generated data and, instead of working on opcode, we used whole assembly output to train our neural network. While the data from the dynamic analysis in our study is increasing the detection rate, using assembly output in ISM without preprocessing and in BSM with small preprocessing to change the format of data decreases the required time to classify a binary file. In [50], Tang et al. proposed a 2-layer LSTM architecture and used the whole assembly code without just picking opcodes. They changed the representation of the binary data stream by transforming every 8 bits into an unsigned integer. Then, they trained and tested the neural network on the integer sequences. The model achieved an 89.6% accuracy rate. This is the closest study to our study since they focused on the whole assembly code and used LSTM to create models. The difference in our study is that we focused on dynamic analysis data instead of statically collected data. We did not do any preprocessing after collecting the data in ISM and did a small amount of preprocessing to put the data in a different format. Also, we achieved a similar accuracy rate with our ISM model and got better results with the BSM model.

A further review of the literature reveals that LSTM models may have a better performance than standard RNN models in NLP applications [8] and [9]. An LSTM

model of English and French languages achieved 8% improvement in perplexity over standard RNN language models [8]. In [9], Sundermeyer et al. compared count-based models to feedforward, recurrent, and LSTM neural networks on two large-vocabulary recognition tasks. As a result of the comparison, Feedforward Neural Networks were outperformed by RNNs, and standard RNNs were outperformed by LSTM with a surprisingly 14% reduction in English development data. In addition, [51] employed LSTM architecture on "Google's One Billion Word Benchmark" dataset. The best improvement achieved in this study is a reduction in perplexity from 51.3 to 30 with combined CNN and LSTM architectures. Furthermore, the best LSTM architecture exhibited the best performance on rare words compared to other models implemented in the study. In another language modeling exploration [52], a language modeling was conducted using Czech spontaneous phone calls and the Wall Street Journal corpus to compare results with well-known data.

2.5 Summary

This section presented background information about malware analysis, neural networks, and natural language processing as well as the relevant works that have been done about malware detection. Malware analysis and natural language processing are the two main parts of this study. Since we focus on malware detection by employing NLP techniques, it is important to understand certain parts of malware analysis and NLP. In addition, we explained neural networks in detail, in particular how Long Short-Term Memory (LSTM) works since it is the specific method that we used to model assembly code. We shared the relevant works related to malware detection including earlier statistical approaches as well as machine learning-based and deep learning-based methods which are popularly used nowadays. In the scope of this study, we investigated the assembly language to apply NLP techniques.

Specifically, we collected the run trace output of Windows executable files and obtained assembly code as our dataset. Then, we performed language modeling on the obtained dataset by employing LSTM and created the first model named Instruction as a Sequence Model (ISM). As the second part of this study, we investigated that how the format of assembly code affects the success of the language model gener-

ated. For this purpose, we examined basic blocks of assembly language that are a bigger structure than assembly instructions. We proposed the second model named Basic Block as a Sequence Model (BSM). In the next section, we present the details of our methodology.

CHAPTER 3

METHODOLOGY

In this section, we first describe the approach of our malware detection method. Then, we describe how we collected the run trace data and the format of our datasets. Finally, we introduce the setup of the environment that was used to create language models and the details of our architecture.

3.1 Approach

The malware detection methods can be mainly categorized into two classes as static approaches and dynamic approaches. While static approaches are working on PE files without executing them, dynamic approaches focus on data dynamically generated by malware during execution. In the early times of malware detection, the traditional studies in which dynamic approaches were employed have investigated dynamically generated data to extract signatures which can be in different forms such as string and graph. However, techniques such as obfuscation made traditional methods ineffective. In addition, the growing populations of malicious software required automated systems to detect malware. The focus shifted from traditional detection methods to Machine Learning (ML) classifiers in the studies in which dynamic approaches were used. Even if ML classifiers made it possible to create automated systems to detect malware, those methods were still limited. Dynamic approaches with machine learning classifiers suffered from feature extraction, which caused the proposed dynamic approach methods to be less effective against new malware variants and obfuscated versions of known ones. In this study, we apply dynamic approaches to neural networks which allow us to create an automated system. As a difference from traditional

and machine learning dynamic approaches, our proposed method does not require signature or feature extraction which makes it a better candidate to detect new malware variants and obfuscated ones.

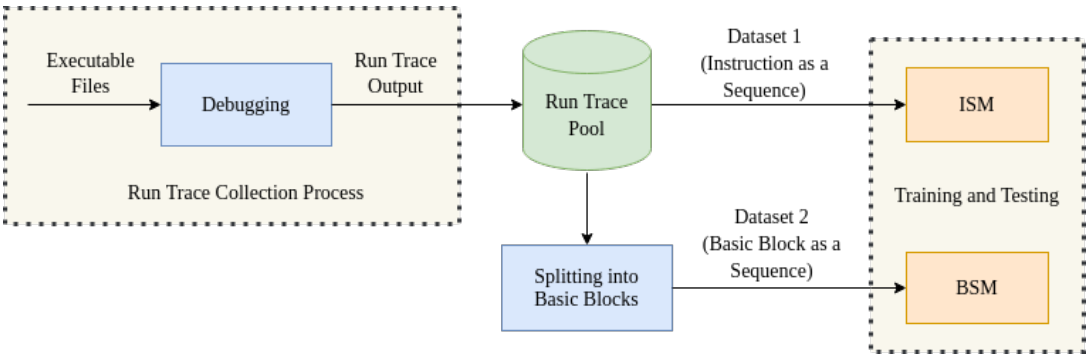

Figure 3.1: The data processing pipeline

We focus on assembly instructions[1] processed during execution for malware detection. The first step of our proposed methodology is to execute each benign/malicious file in a debugger to obtain run trace outputs. Next, the outputs are saved in plain text files such that each line includes one assembly instruction, namely "Instruction as a Sequence Model" (viz. ISM). Then, the first dataset is processed and a second dataset with one basic block[2] per line is obtained. We call this model "Basic Block as a Sequence Model" (viz. BSM). Finally, we feed our LSTM (Long Short-Term Memory) language modeling architecture [6] with our datasets. The overall processing pipeline is presented in Figure 3.1.

The rationale behind the present methodology is to apply deep learning methodologies that have been used for NLP (Natural Language Processing) modeling for classifying run traces of executable files as benign or malicious. There exist similarities between a natural language and the assembly language that allow the application of NLP techniques for the modeling of run traces of the assembly language. More specifically, certain grammatical rules of natural language exhibit similarities to complex patterns exhibited by assembly instructions. The common meaningful unit in many natural languages is the concept of word, which has a functional role similar to the opcodes

[1] In the present study, we use the term "assembly instruction" to mean expressions consisting of opcodes and operands.

[2] In the present study, we use the term "basic block" to mean a piece of straight-line code which has no branch in or out except entry and exit point of a block.

and operands[3] in the assembly language. Instructions of the assembly language perform certain, modular operations in a similar way that words convey meaningful units through modular phrases and sentences in a natural language. Moreover, paragraphs of a natural language may be conceived as sharing certain characteristics with basic blocks of the assembly language. The rationale behind focusing on the basic block structure of assembly code requires a slight explanation. A basic block consists of one or generally more assembly instructions that are executed sequentially. So, it can be said that assembly instructions in a basic block are functionally related to each other. Grouping such related assembly instructions provides advantages to neural networks to extract the meaning and pattern in the data. Also, longer sequences of basic blocks compared to just an assembly instruction provide an additional advantage during the learning process. In our study, we investigated both the assembly instructions (in the ISM model) and the basic blocks (in the BSM model) in two separate datasets. In the following section, we present the datasets.

3.2 The Datasets

For the purpose of the study, we designed and developed two datasets, one being the sequences of instructions (for the ISM model) and the other one, the basic blocks (for the BSM model). We obtained native x86 PE files from Windows operating systems (Microsoft Windows 8.1 Pro (OS Build 9600), Microsoft Windows 10 Pro 19.09 (OS Build 18363.418), and Commando VM v-2.0 [53]). Malicious executables were downloaded from the VirusShare website [54]. Since we aim at detecting malware, we randomly chose the malicious samples including various types of malware, such as virus, worm, and trojan.

The datasets consist of run trace outputs, which are the sequences of the assembly instructions resulting from executing the Portable Executable (PE) files. Table 3.1 reveals that a total of 290 PEs were used to design a language model and conduct experiments. We processed 141 malicious PEs that consisted of 188 million instructions

[3] In the present study, we use the term "operand" to mean the arguments of an instruction. The data for a source operand can be found in the following locations: a register, a memory, an immediate value, and an I/O port. When an instruction returns data to a destination operand, it can be returned to: a register, a memory location, and an I/O port.

Table 3.1: Characteristics of datasets (M is the abbreviation for million)

	Malicious	**Benign**	**Total**
Number of Instructions in Dataset 1 for ISM	188 M	151 M	339 M
Number of Basic Blocks in Dataset 2 for BSM	43 M	14 M	57 M
Number of Files	141	149	290

and 43 million basic blocks. As for the benign files, 149 PEs consisted of 151 million instructions and 14 million basic blocks. We observed that the run trace output of malicious executable files includes more branches than benign executable files.

3.2.1 Run Trace Collection

Figure 3.2 depicts the run trace collection process that was employed in the present study. For creating the dataset, we collect run trace outputs of each binary file by executing them on a debugger in a 32-bit Windows XP Professional Service Pack 3 virtual machine. For this, a Windows XP virtual machine is initially prepared and a snapshot is taken with VirtualBox (version 5.2.34) [55]. We wrote a bash script called `MainScript`[4] that runs on the host system. `MainScript` handles all PE files one by one from the input folder and repeats the following steps for each file as shown in Figure 3.2.

The x64dbg debugger [56] is seen as ready to use when the virtual machine is restored from the initial snapshot and started. We keep the benign and malicious PE files on the Linux host machine (Ubuntu 18.04.4 LTS). An `x64dbgScript` is generated for the corresponding executable file. Then, the executable file and corresponding `x64dbgScript` are moved into the shared folder, which serves as a bridge between the host machine and the virtual machine. The virtual machine is restored from the snapshot and started. The `MainScript` goes on standby on the host computer until the x64dbg debugger window is closed on the virtual machine. At this point, we

[4] The script will be shared upon request. (https://github.com/sirlanci/malware-detection-runtrace.git)

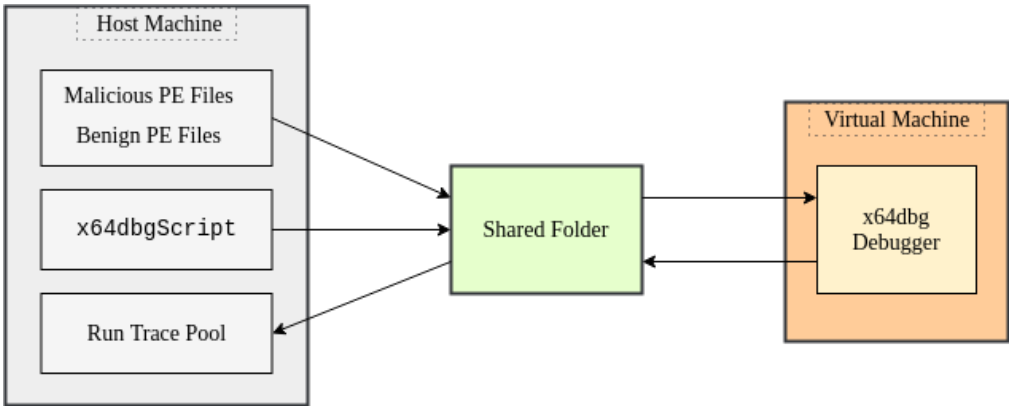

Figure 3.2: The run trace collection process (`MainScript`)

manually load the `x64dbgScript` to the debugger from the shared folder. After the script is run on the debugger, we wait until the executed PE file halts or the maximum executed instruction limit is reached, which is set as 10 million. While the `x64dbgScript` is running, there might be exceptional situations such as invalid PE files, so we observe the process and intervene if it is necessary. Before moving to the next file, the generated run trace output saved in a text file is moved from the shared folder into the run trace pool on the host machine. The `MainScript` continues until all files in the input folder are processed.

During the execution of the PEs, some of the instructions come from the system libraries and some others from the user code space. Since the instructions from the system libraries are common for both malicious and benign files, we only process the code section instructions from each PE file.

The x64dbgScript

The script that was run on x64dbg debugger to collect run trace output of each Portable Executable file is shown in Figure 3.3.

The operation of executing debugger on the virtual machine is done in `MainScript`. With the command that is used to execute the debugger, the name of the PE file is given as a parameter. So, when the debugger is up, the current file whose run trace output will be collected is automatically loaded. At that point, we intervene to load `x64dbgScript` since it could not be automatized to be automatically loaded and

```
1 cmp $pid,0
2 je end_branch
3 rtu
4 rtu
5 StepInto
6 SetTraceLogFile F:\filename_trace.txt
7 TraceSetLog {i:cip}, eip >= 00000000 && eip < 20000000
8 TraceIntoConditional 0,.10000000
9 end_branch:
```

Figure 3.3: Commands in `x64dbgScript`

run.

When the `x64dbgScript` starts to run, the first instruction executed is a compare operation, which checks the "pid" variable of the debugger to specify that there is a PE file successfully loaded on the debugger. If the "pid" is zero, then the "je" command in the second line in Figure 3.3 causes a jump into the branch at the end of the script and the script has stopped without doing any run trace collection operation. We need to use this mechanism to avoid creating empty run trace output files since some of the PE files caused an error, such as invalid PE file, when trying to load them to the debugger.

The "rtu" command in the third and fourth lines of the script is used to move the program counter to the user code. Without the "rtu" command the execution operation gets stuck at the beginning of the debug operation and only collects the instructions from that part which includes a kind of preparation instructions. Normally, one "rtu" command was enough to achieve this goal, however, some PE files required a second "rtu" command to move the program counter to user code. So, to be sure, we used the second "rtu" command. After the rtu commands, we need to use the "StepInto" command in the fifth line before using the trace collection commands. Otherwise, the debugger gets stuck at the beginning of the user code.

In the sixth line, with the "SetTraceLogFile" command, we specify the text file location to write run trace outputs into. The "F:\" path corresponds to the shared folder between the host and virtual machine. So, the run trace outputs are saved into text files in a folder on the host machine by using the PE file name. In the seventh line, the following command, "TraceSetLog", is used to specify the format of the run trace

output and to restrict the address space where the run trace outputs are collected. We did not include any specific information other than the executed instruction for the run trace output format. So, the run trace output only includes the assembly instructions, including opcodes and operands. On the other hand, the address space restriction is required to limit the output so that the run trace output includes only instructions from user code space. In this way, we ignore the instructions executed from the system libraries, since they are common code pieces and not helpful to make a distinction between malicious and benign code pieces.

The next and last command, "TraceIntoConditional", is used to start the execution of the PE file and simultaneously save the executed instructions. There is a condition specified with this command, which shows the amount of the maximum number of instructions that can be written into the run trace output text file for the current PE file. The `x64dbgScript` continues to run until the PE file stop running or the maximum number of instructions is reached.

3.2.2 Dataset Formats

We analyzed two different formats of the same run trace output (one for ISM and the other for BSM). For ISM, we worked on the plain version of the run trace output, i.e. per-line instruction. Preprocessing was not required for this format, as the run trace outputs obtained from the debugger were used directly in the modeling phase. Sample lines from the dataset of the ISM are shown in Figure 3.4.

```
1 mov edi, eax
2 add esp, 0xC
3 test edi, edi
4 jne 0x00428817
5 mov eax, edi
```

Figure 3.4: Sample lines from the ISM dataset

Next, we converted the first version (for ISM) to the basic block per sequence format to get our second dataset for BSM. Sample basic blocks from the second dataset are shown in Figure 3.5.

We wrote a Python script to parse the run trace output into basic blocks. The assem-

```
1 mov esi, dword ptr ds:[0x00401180] mov edx, eax lea ecx, ss:[ebp-0x30] call esi
2 mov ebx, dword ptr ds:[0x00401168] lea ecx, ss:[ebp-0x34] push eax push ecx call ebx
3 mov eax, dword ptr ds:[0x0040A550] or eax, eax je 0x00402387 jmp eax
4 mov dword ptr ss:[ebp-0x48], eax call dword ptr ds:[0x00401054]
5 lea edx, ss:[ebp-0x34] lea eax, ss:[ebp-0x30] push edx push eax push 0x2 call dword ptr ds:[0x00401148]
```

Figure 3.5: Sample lines from the BSM dataset

bly instructions of the ISM dataset were given as input to the script. The script splits the run trace output text file from the basic block endpoints (i.e. branching points) by scanning it from beginning to end. Our data consist of only assembly instructions without including any unnecessary lines such as comments or labels for jump commands. When we split the run trace output from basic block endpoints, the next assembly instruction becomes the beginning point of the next basic block. So, we do not need to do any additional operations for the basic block beginning points. This process was repeated for each run trace output text file of PEs. Three categories of opcodes, which were used for terminating a basic block were unconditional and conditional branches (e.g., "jmp, jz, jnz, jb, jl, jle, jnb, jbe, jge, ja, jns, js, je"), return instructions (e.g., "ret"), and function calls (e.g., "call").

In summary, we processed the two different datasets obtained from the run trace outputs for two different models: ISM and BSM. The proposed models are the subject of the next section.

3.3 The Model

This section introduces technical details of our study including the setup of the train and test environment, used libraries and modules, and the details of the LSTM training and testing pipeline. Before giving the details about the technical parts of our study, we should shortly explain why we prefer to use LSTM over other neural network architectures such as standard RNN, specialized RNN, or different neural network architectures like CNN.

Among the neural network architectures, RNNs are preferred for Natural Language Processing (NLP) tasks since they show strikingly better performance than other types of neural network architectures like CNN. So, basic RNN and specialized RNN ar-

chitectures are better choices to apply the NLP approach to malware detection. In addition, the standard RNN architecture differs from its predecessors due to its ability to remember previous situations. However, since it has a short memory, performance problems are encountered when using RNN while processing long sequences. There are also vanishing and exploding gradient issues in the standard RNN architecture. LSTM, a special kind of RNN architecture, solves the gradient problems and improves standard RNN by modifying the cell structure. Several studies (e.g [8] and [9]) showed that LSTM is successful in extracting semantics from sequential data. Also, in malware detection, opcode sequences and API sequences are modeled by LSTM and achieved good results (e.g. [40] and [57]).

In our study, a text classification neural network, such as an LSTM on sequential data, is the best choice since we approach the malware detection problem from an NLP perspective. As previously noted, among RNN and its specialized types, LSTM shows better performance by minimizing the vanishing and exploding gradient issues. Thus, we applied the LSTM architecture to the datasets presented above. In particular, we employed LSTM for modeling the assembly codes of the Portable Executable (PE) files that already exhibit sequential structures that involve semantic relations.

3.3.1 Setup of The Environment

In this study, we trained and tested our models on a machine with an Intel Xeon E5-2690 2.90GHz CPU. The machine had a CentOS Linux 7 operating system. Also, we implemented and run our models in Python programming language with version 3.6.6. In the Python environment, we used the libraries with their specified versions in Table 3.2.

3.3.2 Imported Libraries and Modules

In this section, we will give a short explanation of the python libraries and modules that are used in the scope of our study.

- **import os**

Table 3.2: Required Python libraries to build the LSTM train and test pipeline

Tensorflow	2.4.0
Keras	2.4.3
scikit-learn	0.22.2
NumPy	1.19.0
Pickle	4.0
MatPlotLib	3.3.0
Seaborn	0.10.1

The module, os, is used at the beginning of our pipeline to take the data from folders on the host machine that training and testing operations will be performed on.

- **import NumPy**

  Tensorflow 2.4.0 version required to use NumPy arrays to give the training, validation, and testing splits to the training and testing operations. So, we convert the python lists into NumPy arrays before the training and testing phase. To be able to use NumPy arrays, we import the numpy library.

- **import pickle**

  For later use without repeating the tokenization operation, we need to save the dictionary, which is obtained as the tokenizer's output, on the disk, and the Pickle library provides a way to achieve this purpose. Pickle stores python objects efficiently with its compact binary representation so that we can save the memory while storing the dictionary on the disk.

- **from keras.preprocessing.text import Tokenizer**

  We used the Tokenizer module from Keras to tokenize the text content in our dataset and extract the tokens. Also, the functionality provided by the Tokenizer module to encode the data from text to integer values is used in the course of our study.

- from **keras.preprocessing.sequence import pad_sequences**

Pad_sequences module from the Keras library is used to pad the sequences that are shorter than our fixed sequence length.

- **from keras.models import Sequential**

  Sequential module from the Keras library is used to build the plain stack of layers.

- **from keras.layers import Embedding, LSTM, Bidirectional, Dense, Dropout, GlobalMaxPool1D**

  Embedding, LSTM, Bidirectional, Dense, GlobalMaxPool1D, and Dropout modules from the Keras library are used to add each of those layers into our layered architecture in the neural network.

- **from keras.optimizers import Adam**

  We used the Adam module from the Keras library as the optimizer required to use while training the neural network.

- **from sklearn.model_selection import train_test_split**

  The module named train_test_split is required to separate the dataset into several proportions as train, validation, and test to use for different purposes during the training and testing operations.

- **from tensorflow.math import confusion_matrix**

  Confusion_matrix module is used to create the confusion matrix of the given test set.

- **from matplotlib import pyplot**

  We use the module named Pyplot to plot the loss and accuracy graphs of the training phase.

- **import seaborn**

  The Seaborn library is used to create a confusion matrix that has graphical components.

3.3.3 LSTM Train and Test Pipeline

In our study, we build a pipeline to take the dataset, to prepare the data for modeling, to train and test the neural network. To create the pipeline, we implemented Algorithm 1 on Python 3.6.5 by using TensorFlow [58] and Keras [59] open-source libraries. In the following of this subsection, we will give the details of the pipeline[5].

As we discussed in the datasets section, the run trace outputs of Portable Executable files are stored in separate text files. Our LSTM pipeline takes those text files as input and starts the operation by reading each file and merging the benign and malicious run trace outputs in two separate text files named “merged-benign” and “merged-malicious”. We preferred to merge the content of run trace output text files in the two merged text files to use the data later by reading from those merged text files without opening and closing each run trace output file to read the data. The assembly instructions coming from a malicious file are conceived as malicious. So, they were labeled "malicious" to feed the neural network even if they include both malicious and benign assembly instructions.

We use two counters while taking the data line-by-line from run trace output text files. One of them is used to limit the total number of lines that will be taken from each category as malicious and benign. The other counter is used to limit the number of lines that will be taken from each run trace output file. The run trace output text files include lines up to 10 million and without the counter control mechanism, the majority of the data could be taken from only a small amount of files. Instead of that, we prefer to take run trace output of different PE files so we used the counter to balance the unequal situation.

As a result of the first step in the pipeline, we obtain two text files, named “merged-benign” and “merged-malicious”, which include the number of lines specified by the first counter, that are taken from the run trace output files of PEs.

At the next step, the data is taken from the two merged text files and put into Python data structures. Also, the labels of the lines are added to the data structure accord-

[5] The Python codes of the LSTM train and test pipeline will be shared upon request (https://github.com/sirlanci/malware-detection-runtrace.git)

Algorithm 1: Algorithm for Modeling

input : $runTracePool = \{f_1, f_2, ..., f_N\}$ where $N = 290$

output: $accuracyRate, loss$

for $f \in runTracePool$ **do**
    $lines \leftarrow f$.readLines ();
    if f *is malware* **then**
        $mergedMalicious$.writeFile ($lines$);
    else
        $mergedBenign$.writeFile ($lines$);

for $l \in mergedMalicious$ **do**
    $sequences.value \leftarrow$ Append (l) ;
    $sequences.label \leftarrow$ Append (0) ;
    /* "0" is used to label malicious sequences */

for $l \in mergedBenign$ **do**
    $sequences.value \leftarrow$ Append (l) ;
    $sequences.label \leftarrow$ Append (1) ;
    /* "1" is used to label benign sequences */

for $s \in sequences$ **do**
    $ts \leftarrow$ Tokenize (*s*) ;
    $tokenizedSequences \leftarrow$ Append (*ts*) ;

for $ts \in tokenizedSequences$ **do**
    $es \leftarrow$ Encode (*ts*) ;
    $encodedSequences \leftarrow$ Append (*es*) ;

for $es \in encodedSequences$ **do**
    $ps \leftarrow$ Pad (*es*) ;
    $paddedSequences \leftarrow$ Append (*ps*) ;

$(trainSet, testSet) \leftarrow$ Split (*paddedSequences*) ;

$(trainSet, validationSet) \leftarrow$ Split (*trainSet*) ;

$model$.Train (*trainSet, validationSet*) ;

$(accuracyRate, loss) \leftarrow model$.Test (*testSet*) ;

ing to which merged file the corresponding line comes from. For example, if a line is taken from merged-benign, its label value becomes “1” and if it is taken from merged-malicious, its label value becomes “0”. At this point, the run trace output lines, sequences, are stored in a python list and the corresponding labels of these lines are stored in another python list.

At the next step, tokenization operation is performed on the sequences. For this process, we used the tokenizer from Keras preprocessing library. To make it more specific, we explain the tokenization process by taking the following sequence from sequences in Figure 3.5 and tokenizing it.

➔ mov esi, dword ptr ds:[0x00401180] mov edx, eax lea ecx, ss:[ebp-0x30] call esi

To obtain tokens in the sequence above, the sequence is divided into pieces by the space character and punctuation characters, and those characters are removed from the sequence. So, as the output of this process, we obtain the following tokens in this specific order.

➔ mov, esi, dword, ptr, ds, 0x00401180, mov, edx, eax, lea, ecx, ss, ebp, 0x30, call, esi

After the tokenization is completed and our dictionary is built, the pipeline continues with the encoding and padding operations. The encoding operation is necessary since our data consists of strings but the LSTM requires integer values to work on. So, the encoding process assigns a unique integer value to each token in the dataset. Sample encoded sequence of the sequence tokenized above can be seen below.

➔ 12, 27, 8, 46, 17, 35, 12, 22, 41, 54, 38, 15, 24, 59, 10, 27

After the encoding operation is done, we obtain a python list including sequences consisting of integer values similar to the sample sequence above. The sequences are in different lengths, however, the LSTM requires fixed-length sequences. To overcome this problem, we specified a maximum sequence length for each of the models (8 for ISM and 30 for BSM). If a sequence is bigger than the maximum sequence

length, its first part is taken up until the maximum sequence length is reached and the remaining part is discarded. If a sequence is smaller than the maximum sequence length, the padding operation is performed to complete the sequence to the maximum sequence length. In the padding operation, the required amount of zero integer value, which does not correspond to any token in the dictionary, is added to the end of the sequence. As a result, we obtain a python list including fixed-length sequences consisting of integer values.

At the next step, the dataset will be separated into three splits, which are named as train, validation, and test, to use during the training and testing process. Even if there are no strict rules to specify the train, validation, and test proportions, there is an accepted common opinion, which is separating %60 of the dataset for training, %20 of the dataset for validation, and %20 of the dataset for testing. So, we preferred to split the dataset according to the common opinion. Up to this point, we performed operations to take the data from text files into python data structures and to prepare the data for the training and testing phase. Next, we explain the layered architecture of our neural network.

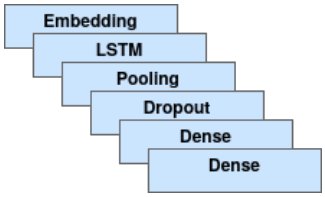

Figure 3.6: The layers of our proposed architecture

We propose a six-layer architecture for the two models (ISM and BSM) in a sequential structure, as shown in Figure 3.5. The first layer of the architecture is the embedding layer. At this layer, the word embedding vectors are created. The second layer is the bidirectional LSTM layer. After the LSTM layer, Global Max Pool is used in the pooling layer to reduce the size of the vectors. The dropout layer is added after the pooling layer to activate selected nodes in the network to increase learning efficiency. The first dense layer reduces 128 dimensions to 64 dimensions. The second dense layer reduces 64 dimensions to 2 dimensions. At this last dense layer, a sigmoid is used as an activation function since the problem we work on requires to perform binary classification.

3.3.4 Parameters for Training and Testing Processes

In order to find the best values for the parameters (*maximum sequence length*, *dropout rate*, *optimizer*, and the *number of LSTM nodes*) in the language modeling task, we tried several different values. While all other parameters are fixed, the maximum sequence length is the only parameter that takes different values for both ISM and BSM. Trial values and the best results (shown in bold) obtained from testing these values are shown in Table 3.3.

Table 3.3: Trial values for ISM and BSM

	Value 1	**Value 2**	**Value 3**	**Value 4**	**Value 5**
Sequence Length for ISM	4	6	**8**	10	-
Sequence Length for BSM	12	18	24	**30**	36
Dropout Rate for ISM and BSM	**0.2**	0.5	0.8	-	-
Optimizer for ISM and BSM	**Adam**	RMSprop	Adagrad	SGD	SGD with momentum
LSTM Output Nodes for ISM and BSM	32	**64**	128	256	-

In the beginning, we created 4 different trials for ISM, fixing other parameters (dropout rate, optimizer, and the number of LSTM output nodes), using 4, 6, 8, and 10 as the maximum sequence length. The loss was lower and the accuracy was better for 8 and 10 than 4 and 6. There were almost any loss and accuracy differences between 8 and 10. Since the training time was shorter for 8, we chose 8 as the maximum sequence length for the final architecture. On the other hand, for BSM, we trained 5 models with the maximum sequence length of 12, 18, 24, 30, and 36, again fixing other parameters. From 12 to 30, the model loss was constantly reduced, but from 30 to 36, there was no difference in loss and accuracy rate. Therefore, we decided to use 30

for the value of the maximum sequence length in BSM, as there is a difference in training hours as in ISM. The remaining parameters explained below are the same for two models: ISM and BSM.

According to the study in [60], there is no optimal dropout rate parameter that can fit all neural network architectures, and this idea has been demonstrated by some experiments on different data sets. So, we trained 3 neural networks with dropout rates 0.2, 0.5, and 0.8 to find the best one for our data by keeping other parameters fixed. A significant increase in loss and decrease in accuracy rate was observed compared to the others with a dropout rate of 0.8. The 0.2 dropout rate is a bit better than 0.5 dropout rate in the matter of loss and accuracy rate so we chose 0.2 as the dropout rate. Also, we trained on several known optimizers with their default configurations such as Adam [61], RMSprop [62], Adagrad [63], Stochastic Gradient Descent (SGD) [64], and SGD with momentum [65], fixing other parameters. We found that Adam and RMSprop achieved similar results as it is also claimed in [64]. We preferred to use Adam since it showed slightly better performance than RMSprop.

Moreover, we decided the output number of the LSTM layer by modeling with different numbers of output nodes and comparing the loss and accuracy rates of their results while the other parameters are kept fixed. With the 32 output nodes, the model loss was seriously higher than others. Using 128 and 256 output nodes did not cause a serious decrease in loss, so we chose 64 as the output nodes of the LSTM layer, as the lesser parameter allows for faster training.

Finally, we modeled malware and benign languages with these best parameters for 20 epochs, which was the highest number of epoch we used, to specify the epoch number. According to our observations during all of our experiments and the last training with 20 epochs, the neural network learns our data mostly at the first three epochs. After three epochs, there is no striking decrease in loss and an increase in the accuracy rate. In addition, the lesser number of epochs decreases the risk of overfitting, so we chose to train the neural network for 3 epochs in our final model.

3.4 Summary

In this section, we presented the details of our methodology, particularly our approach, the dataset collection process, the format of the datasets, the setup of the neural network training environment, required libraries and modules, the modeling pipeline, and the parameters used in the modeling process.

To sum up, we created a semi-automated process to collect run trace output of Windows executable files. Then, we collected the dynamically generated assembly code output named run trace to obtain our first dataset (named Instruction as a Sequence Model) and put the assembly code in the first dataset into a different format to obtain our second dataset (Basic Block as a Sequence Model). We used language modeling techniques from natural language processing (NLP) as an approach to classify assembly codes from malicious and benign files. To generate language models on assembly codes, we employed a specialized recurrent neural network (RNN) architecture named long short-term memory (LSTM). To implement our approach, we used modules from TensorFlow and Keras libraries on Python programming language as well as modules from Pickle, Numpy, Sklearn, Matplotlib, and Seaborn libraries. Lastly, we presented the values tried on the parameters required in the training and testing process.

CHAPTER 4

RESULTS

In this section, we present the results of the two models, namely the ISM and the BSM.

4.1 The ISM (Instruction as a Sequence Model)

We conducted a total of 16 experiments for the first model, ISM by manipulating 4 values for the *sequence length*, 3 values for *dropout rate*, 5 values for *optimizer* and 4 values for the *number of LSTM nodes*. The resulting number of correctly and incorrectly classified samples are shown in a confusion matrix (Table 4.1).

Table 4.1: Confusion matrix of test set from the evaluation process of ISM where TN is the number of true negatives, FN is the number of false negatives, FP is the number of false positives, and TP is the number of true positives

		Predicted Class	
		Malware	**Benign**
Actual Class	**Malware**	TP 34,810,727	FN 2,951,034
	Benign	FP 5,544,678	TN 24,691,072

The number of true negatives TN in the confusion matrix refers to correctly recognized instructions as benign instructions, whereas the number of true positives TP

refers to correctly recognized instructions as malicious instructions. The number of false positives FP shows benign instructions recognized as malicious, whereas the number of false negatives FN shows malicious instructions recognized as benign.

The true positive rate TPR is calculated by (4.1) as 92.19% and the false positive rate FPR is calculated by (4.2) as 18.34%. Accuracy rate ACC is calculated by (4.3) as 87.51%.

$$TPR = \frac{TP}{(TP + FN)} \tag{4.1}$$

where TP is the number of True Positive cases and FN is the number of False Negative cases.

$$FPR = \frac{FP}{(FP + TN)} \tag{4.2}$$

where FP is the number of False Positive cases and TN is the number of True Negative cases.

$$ACC = \frac{(TP + TN)}{(TP + TN + FP + FN)} \tag{4.3}$$

4.2 The BSM (Basic Block as a Sequence Model)

For the second model, namely BSM, we conducted 17 experiments by manipulating 5 values for the *sequence length*, 3 values for the *dropout rate*, 5 values for the *optimizer* and 4 values for the *number of LSTM nodes*.

Table 4.2: Confusion matrix of test set from evaluation process of BSM where TN is the number of true negatives, FN is the number of false negatives, FP is the number of false positives, and TP is the number of true positives

		Predicted Class	
		Malware	**Benign**
Actual Class	**Malware**	TP 8,705,965	FN 13,816
	Benign	FP 70,625	TN 2,628,383

After the BSM was trained, we evaluated it on the test set which consisted of 11 million basic blocks approximately. The resulting number of correctly and incorrectly classified samples are shown in Table 4.2 in the confusion matrix.

The true positive rate TPR, calculated using (4.1), was 99.84% and the false positive rate FPR, calculated using (4.2), was 2.62%. Finally, the correctly classified percentage of samples, accuracy rate ACC, calculated using (4.3), was 99.26%.

4.3 Comparison of The Models

The findings of the two models are summarized in Table 4.3.

Table 4.3: Comparison of the models

	$\mathbf{TPR}(\%)$	$\mathbf{FPR}(\%)$	$\mathbf{ACC}(\%)$
ISM	92.19	18.34	87.51
BSM	99.84	2.62	99.26

The only factor that led to the differences between the two proposed models was the format of data processing. The assembly instructions are the basic part of an executable's source code. It includes meaningful information and patterns and allows the ISM to achieve an 87.51% accuracy rate. However, the basic blocks in assembly code consist of more than one instruction, which are functionally related. In addition to the relation of words in instruction, there are also relations between different instructions in a basic block. Thus, the basic blocks with their longer and more complex structures include more meaningful information and more patterns than instructions, resulting in the BSM to achieve a 99.26% accuracy rate. In summary, the results of the final experiments on the two models suggest that the basic block as a sequence model (BSM) representation exhibits a better structure for LSTM modeling compared to the instruction as a sequence model (ISM) representation.

4.4 Discussion

At the earlier times of information technology (IT), malware detection methods were mostly based on signatures generated statically by analyzing malware to protect information systems. However, various techniques to bypass such signature-based methods were discovered and are still in development by malware authors. Also, the excessive growth of IT requires so much effort to analyze each new malware. The obfuscation techniques and the increasingly required effort made automated malware detection systems a necessity. In the past decades, artificial intelligence (AI) was introduced to academic research on malware detection.

In this scope, machine learning (ML) classification algorithms such as Random Forest (RF), Support Vector Machine (SVM), and Decision Tree (DT) were popularly used to classify the data from malicious and benign files. The studies employing ML classification algorithms require feature extraction. Opcodes and API calls are mostly used features of malicious and benign software for classification purposes. However, these AI-based methods still require a domain expert and a certain effort for feature extraction. In the following studies, the focus of academic research on malware detection is shifted from ML classification to deep neural networks (deep learning).

Deep learning-based methods have an advantage over ML classification methods, which is not requiring feature extraction since the deep neural networks perform it internally. First, convolutional neural networks (CNNs) are popularly used to propose malware detection methods. CNNs work on image data to learn patterns and classify given untitled images. In most of the CNN methods proposed in the literature such as [31], [32], and [33], opcode sequences or assembly instructions of malicious and benign software are converted into images and the neural network is trained on those images.

Recurrent neural networks (RNNs) work on sequential data to extract patterns that are representing the data so it performs better on tasks including sequential data such as natural language classification and speech recognition. Also, long short-term memory (LSTM) is a specialized RNN architecture that achieves better results in language modeling tasks compared to RNN and other specialized RNN architectures. The

studies in the literature employing LSTM such as [36] and [37] for malware detection purpose focuses on opcode sequences instead of the whole assembly code obtained. However, we think that assembly language shows similarities to natural languages by including semantic and syntactical elements between opcodes and operands as well as between sequential instructions. So, the assembly code without excluding any part, which comes from malicious and benign software, possibly shows differences that can lead to making a distinction between the assembly code of the two classes.

The accuracy rate of our proposed methods is in terms of assembly code sequence classification instead of binary file classification since in this study we aimed to be able to separate assembly sequences as benign and malicious. So, even if the meaning of the accuracies between other methods and our proposed methods are different, we wanted to compare them to evaluate the performance/success of our proposed methods. The studies that focused on opcode sequences and assembly code to detect malicious software can be separated into three main categories: machine learning classifiers, convolutional neural networks (CNNs), and recurrent neural networks (RNNs). Also, the majority of those studies focus on opcode sequences while a few others investigate assembly code as a whole.

Table 4.4: Evaluation of our proposed methods

Method	**Data Format**	**ACC**(%)
Random Forest [3]	Opcode	96.00
Random Forest [34]	Opcode	97.00
Random Forest [35]	Opcode	91.43
CNN - ResNet 152 [44]	Opcode	88.36
CNN - GoogleNet [45]	Opcode	74.5
CNN [46]	Instruction	98.00
4-layer LSTM [49]	Opcode	88.51
2-layer LSTM [50]	Opcode	89.6
ISM	Instruction	87.51
BSM	**Basic Block**	**99.26**

The accuracy rates shown in Table 4.4 are the results obtained in the corresponding studies with their dataset. In [3], [34] and [37] shown in Table 4.4, a set of machine learning classifiers were evaluated and Random Forest got the highest accuracy rate in each of the studies. There is just a slight difference between the accuracy rate obtained in [34] and our proposed method BSM. However, machine learning studies focusing on opcodes suffer from the feature extraction process which puts an extra burden on the detection process. Also, they are not strong against the obfuscation methods which change the statistical information in opcodes. This feature extraction operation is done by a neural network without requiring additional feature extraction so neural networks with similar success (e.g accuracy rate) can be thought of as a better robust method in comparing to machine learning classifiers. On the other hand, our proposed method BSM improves the accuracy rates of CNN - ResNet 152 [44] and CNN - GoogleNet [45] by focusing on assembly code sequences, which include more semantic relationships, instead of opcode sequences. Our proposed method BSM achieved a slightly better accuracy rate of CNN method proposed in [46] which is also focused on not just opcodes but whole assembly code. However, in previous studies [47] and [48], it was shown that slight differences in assembly code could bypass image detection methods. Thus, text classification neural networks can be a better choice against obfuscation techniques. Also, text classification neural networks do not require converting opcode or assembly code into an image as CNN, which reduces the required preprocessing time.

In [49] and [50], two LSTM architectures are built by using 4 and 2 hidden LSTM layers respectively. Our proposed method ISM achieved similar accuracy rates with the models proposed in [49] and [50]. Our second method BSM improved the accuracy rate by virtually %10 by achieving %99.26 accuracy rate. In addition, our LSTM architecture includes just one hidden LSTM layer, which makes the architecture less complex and keeps the required resources (e.g processing power) for the training of the neural network lower.

Finally, we created a method to detect malicious code by extending previously proposed methods in several aspects. The dynamically generated data, run trace output, decreases the effectiveness of obfuscation techniques so our method is capable of detecting malware using obfuscation techniques. Also, the ability of neural networks

to adapt to the changes in data makes our method stronger against the obfuscation methods. Working on assembly code without preprocessing in ISM and with small preprocessing in BSM keeps the time minimum spent for preprocessing. In addition, using just one LSTM layer decreases the number of resources required for training the neural network.

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

Malware detection methods have been evolving since information systems became an important part of people's life. The sheer growth of information technology requires faster and more efficient malware detection methods. Also, the anti-detection methods such as obfuscation techniques developed by malware authors increase the need for smart and fully automated malware detection methods. In the light of these needs, the advancements in artificial intelligence made AI-based methods the best candidate to develop better malware detection methods.

In the first AI-based studies, machine learning (ML) classification algorithms were popularly used to classify the data obtained from malicious and benign software. However, ML classification algorithms do not provide fully automated methods since they require time and effort for feature selection and extraction. So, in the following studies, the focus was shifted from ML-based methods to deep learning (DL) based methods since deep neural networks simulated the learning process better and provided smarter and faster agents. Nowadays, deep neural network architectures, particularly convolutional neural networks (CNNs) and recurrent neural networks (RNNs), are widely employed in academic researches to classify malicious and benign software.

In this study, we proposed an approach to classify malicious and benign code pieces. We worked on assembly language. Unlike other studies, we implemented our approach on dynamic analysis data instead of static analysis and focused on assembly code as a whole instead of focusing on just opcodes. With the deep learning architec-

ture LSTM, which is a specialized RNN, we modeled malicious and benign software run traces like natural languages.

The neural networks were trained on two datasets with different formats of the same run trace output data. In one of the datasets, sequences are structured as instructions and in the other, sequences are structured as basic blocks. The ISM model was trained on the dataset that is constructed as an instruction per sequence. Then we designed the BSM model, which we aim to achieve better accuracy. Since the basic blocks in the assembly code consist of multiple functions that are functionally related, we processed the run trace outputs by splitting them into basic blocks for the BSM.

We selected optimum parameter values for our neural network architectures based on our experimental results. The resulting accuracy rate (87.51%) with the ISM shows that it is possible to classify malicious and benign assembly codes by LSTM. When our improved model BSM was used, 99.26% accuracy and 2.62% false positive rate were achieved, better than the case in which ISM was used and most of the previous studies suggested. Our proposed framework for the dynamic analysis of run trace data also makes the approach resistant to polymorphic and metamorphic malware.

5.2 Limitations and Future Work

A major limitation of the study is that certain steps in the processing pipeline require human intervention and this makes the run trace collection process a bit slower. The coverage of the malware and benign files is limited to a few hundred files, despite the rich dataset obtained by the help of the run trace collection process. Future research should address improvements of the data processing pipeline, in particular developing an API for the x64dbg debugger to be able to collect data automatically for a given x86 Windows executable file as well as improving the run trace collection process by modifying the executable files to run on multiple CPUs in parallel. Future research should also address moving our current detection process from the code level to the file level and the application of our proposed method for classifying different types of malware, such as worms, trojan horses at the OS level both for desktop and mobile operating systems. Also, the future work should address the performance comparison

between file level detection version of our proposed method and a signature based detection mechanism.

The neural network architecture, LSTM, allows us only to create black box models since we do not have control over its internals. So, as in every other study employing deep neural networks, being a black box at the model level puts a limitation on this study. In addition, while we were specifying parameters, required for language modeling, such as sequence length and the number of nodes in the LSTM layer, we tried several values and picked the best ones that showed the best performance on our data. However, these assumptions are limited to this study since, in deep learning research, the dataset difference is able to lead to different results between similar studies. As a result, the nature of deep neural network architectures presents a limitation and the specified parameters in deep learning studies might be limited to the dataset used in the corresponding study. In this respect, future research depends on studies focused on explainable deep learning and a future success achieved in this subject might allow us to remove such architectural and dataset-related limitations.

Malicious software has mechanisms to avoid analysis such as avoiding execution in case of detecting a debugger. So, malicious files with such mechanisms cause problems in the dynamic data collection process. This limitation should be addressed in future work by investigating anti debugging techniques. Also, the robustness of our models against obfuscation methods is not certain. Even if our models have resistance against the obfuscation techniques trying to change the static appearance of malicious files, new obfuscation methods can be developed to change the look of dynamic assembly code of malicious files. Thus, in future work, the effect of static obfuscation methods and if available, the dynamic obfuscation methods on the proposed detection method should be investigated.

REFERENCES

[1] AV-TEST. [Online]. Available: https://www.av-test.org/en/statistics/malware, Accessed on: Aug 2020.

[2] VirusTotal. [Online]. Available: https://www.virustotal.com/en/statistics, Accessed on: Aug 2020.

[3] A. Shabtai, R. Moskovitch, and C. Feher *et al.*, "Detecting unknown malicious code by applying classification techniques on opcode patterns," *Secur. Inform.*, vol. 1, no. 1, p. 1, 2012, doi: 10.1186/2190-8532-1-1.

[4] R. Pascanu, J. W. Stokes, H. Sanossian, M. Marinescu, and A. Thomas, "Malware classification with recurrent networks," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Brisbane, QLD, Australia, 2015, pp. 1916-1920, doi: 10.1109/ICASSP.2015.7178304.

[5] Z. Markel and M. Bilzor, "Building a machine learning classifier for malware detection," in *2014 Second Workshop on Anti-malware Testing Research (WATeR)*, Canterbury, 2014, pp. 1-4, doi: 10.1109/WATeR.2014.7015757.

[6] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735-1780, 15 Nov. 1997, doi: 10.1162/neco.1997.9.8.1735.

[7] C. Acarturk, M. Sirlanci, P. G. Balikcioglu, D. Demirci, N. Sahin, and O. A. Kucuk, "Malicious Code Detection: Run Trace Output Analysis by LSTM," *IEEE Access*, vol. 9, pp. 9625-9635, 5 Jan. 2021, doi: 10.1109/ACCESS.2021.3049200.

[8] M. Sundermeyer, R. Schlüter, and H. Ney, "LSTM neural networks for language modeling," in *Thirteenth annual conference of the international speech communication association*, 2012.

[9] M. Sundermeyer, H. Ney, and R. Schlüter, "From Feedforward to Recurrent LSTM Neural Networks for Language Modeling," in *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 23, no. 3, pp. 517-529, March 2015, doi: 10.1109/TASLP.2015.2400218.

[10] J. Aycock, "Definitions and Timeline" in *Computer viruses and malware*, 1st Ed., New York, USA, Springer Science & Business Media, 2006, pp. 11.

[11] M. Sikorski and A. Honig., "Basic Static Techniques" in *Practical Malware Analysis: The Hands-on Guide to Dissecting Malicious Software*, 1st Ed., San Francisco, USA, No starch press, 2012, pp. 9.

[12] M. Sikorski and A. Honig., "Basic Dynamic Analysis" in *Practical Malware Analysis: The Hands-on Guide to Dissecting Malicious Software*, 1st Ed., San Francisco, USA, No starch press, 2012, pp. 40.

[13] [Online]. Available: https://www.thesslstore.com/blog/polymorphic-malware-and-metamorphic-malware-what-you-need-to-know/, Accessed on: Dec 2020.

[14] J. Landage and M. P. Wankhade, "Malware and malware detection techniques: A survey," *International Journal of Engineering Research and Technology*, vol. 2 no. 12, pp. 61-68, Dec 2013.

[15] B. M. Mehtre, "Advances In Malware Detection-An Overview,", 2021, arXiv:2104.01835. [Online]. Available: https://arxiv.org/abs/2104.01835

[16] Z. Bazrafshan, H. Hashemi, S. M. H. Fard, and A. Hamzeh, "A survey on heuristic malware detection techniques," *5th Conference on Information and Knowledge Technology*, Shiraz, Iran, May 2013, pp. 113-120, doi: 10.1109/IKT.2013.6620049.

[17] D. Gibert Llauradó, C. Mateu Piñol, and J. Planes Cid, "The rise of machine learning for detection and classification of malware: Research developments, trends and challenge," *Journal of Network and Computer Applications*, vol. 153, pp. 102526, March 2020, doi: https://doi.org/10.1016/j.jnca.2019.102526.

[18] J. Singh and J. Singh, "Challenge of malware analysis: malware obfuscation techniques," *International Journal of Information Security Science*, vol. 7, no. 3, pp. 100-110, 2018.

[19] I. You and K. Yim, "Malware obfuscation techniques: A brief survey," in *2010 International conference on broadband, wireless computing, communication and applications*, Fukuoka, Japan, Nov. 2010, pp. 297-300, doi: https://doi.org/10.1109/BWCCA.2010.85.

[20] [Online]. Available: https://resources.infosecinstitute.com/topic/simple-malware-obfuscation-techniques/, Accessed on: Dec 2020.

[21] [Online]. Available: https://en.wikipedia.org/wiki/Artificial_neural_network, Accessed on: Dec 2020.

[22] [Online]. Available: https://hmkcode.com/ai/backpropagation-step-by-step/, Accessed on: Dec 2020.

[23] I. Goodfellow, Y. Bengio, and A. Courville, "Sequence Modeling: Recurrent and Recursive Nets," in *Deep learning*, vol. 1, no. 2, Cambridge, MIT Press, 2016, pp. 367.

[24] [Online]. Available: https://towardsdatascience.com/understanding-rnn-and-lstm-f7cdf6dfc14e, Accessed on: Dec 2020.

[25] Illustrated Guide to Recurrent Neural Networks. [Online]. Available: https://towardsdatascience.com/illustrated-guide-to-recurrent-neural-networks-79e5eb8049c9, Accessed on: Dec 2020.

[26] [Online]. Available: https://colah.github.io/posts/2015-08-Understanding-LSTMs/, Accessed on: Dec 2020.

[27] [Online]. Available: https://en.wikipedia.org/wiki/Natural_language_processing, Accessed on: Dec 2020.

[28] [Online]. Available: https://machinelearningmastery.com/natural-language-processing/, Accessed on: Dec 2020.

[29] R. Mitkov, "Machine Learning" in *The Oxford Handbook of Computational Linguistics*, 1st Ed., Oxford, England, Oxford University Press, 2004, pp. 377

[30] [Online]. Available: http://karpathy.github.io/2015/05/21/rnn-effectiveness/, Accessed on: Jan 2021 .

[31] [Online]. Available: https://machinelearningmastery.com/what-are-word-embeddings/, Accessed on: Dec 2020.

[32] D. Bilar, "Opcodes as predictor for malware," *Int. J. Electron. Secur. Digit. Forensic*, vol. 1, no. 2, pp. 156–168, May 2007, doi: 10.1504/IJESDF.2007.016865.

[33] I. Santos *et al.*, "Idea: Opcode-Sequence-Based Malware Detection," in *Proc 2nd International Symposium on Engineering Secure Software and Systems*, Pisa, Italy, 2010, pp. 35-43, doi: 10.1007/978-3-642-11747-3_3.

[34] A. Yewale and M. Singh, "Malware detection based on opcode frequency," in *2016 International Conference on Advanced Communication Control and Computing Technologies (ICACCCT)*, Ramanathapuram, 2016, pp. 646-649, doi: 10.1109/ICACCCT.2016.7831719.

[35] H. Zhang, X. Xiao, F. Mercaldo, S. Ni, F. Martinelli, and A. K. Sangaiah, "Classification of ransomware families with machine learning based on N-gram of opcodes," *Future Generation Computer Systems*, vol. 90, pp. 211-221, 2019, doi: 10.1016/j.future.2018.07.052.

[36] S. Euh, H. Lee, D. Kim, and D. Hwang, (2020) "Comparative Analysis of Low-Dimensional Features and Tree-Based Ensembles for Malware Detection Systems," *IEEE Access*, vol. 8, pp. 76796-76808, April 2020, doi: 10.1109/ACCESS.2020.2986014.

[37] J. Lee, K. Jeong, and H. Lee, "Detecting metamorphic malwares using code graphs," in *Proc. of the 2010 ACM Symposium on Applied Computing (SAC '10)*, New York, NY, USA, 2010, pp. 1970–1977, doi: 10.1145/1774088.1774505.

[38] Y. Ki, E. Kim, and H. K. Kim, "A novel approach to detect malware based on API call sequence analysis," *International Journal of Distributed Sensor Networks*, vol. 11, no. 6, June 2015, Art no. 659101, doi: 10.1155/2015/659101.

[39] B. Cheng, Q. Tong, J. Wang, and W. Tian, "Malware clustering using family dependency graph," *IEEE Access*, vol. 7, pp. 72267-72272, May 2019, doi: 10.1109/ACCESS.2019.2914031.

[40] B. Kolosnjaji, A. Zarras, G. Webster, and C. Eckert, "Deep learning for classification of malware system call sequences," in *Australasian Joint Conference on Artificial Intelligence*, Hobart, TAS, Australia, Dec 2016, pp. 137-149, doi: 10.1007/978-3-319-50127-7_11.

[41] B. Athiwaratkun and J. W. Stokes, "Malware classification with LSTM and GRU language models and a character-level CNN," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, New Orleans, LA, 2017, pp. 2482-2486, doi: 10.1109/ICASSP.2017.7952603.

[42] R. Agrawal, J. W. Stokes, M. Marinescu, and K. Selvaraj, "Neural Sequential Malware Detection with Parameters," in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Calgary, AB, 2018, pp. 2656-2660, doi: 10.1109/ICASSP.2018.8461583.

[43] Z. Zhang, P. Qi, and W. Wang, "Dynamic Malware Analysis with Feature Engineering and Feature Learning," in *Proc. 34th the AAAI Conference on Artificial Intelligence*, New York, NY, USA, 2020, pp. 1210-1217, doi: 10.1609/aaai.v34i01.5474.

[44] R. U. Khan, X. Zhang, R. Kumar, and E. O. Aboagye, "Evaluating the performance of resnet model based on image recognition," in *Proceedings of the 2018 International Conference on Computing and Artificial Intelligence*, Chengdu, China, March 2018, pp. 86-90, doi: 10.1145/3194452.3194461.

[45] R. U. Khan, X. Zhang, and R. Kumar, "Analysis of ResNet and GoogleNet models for malware detection," *Journal of Computer Virology and Hacking Techniques*, vol. 15, no. 1, pp. 29-37, August 2018, doi: 10.1007/s11416-018-0324-z.

[46] R. Kumar, Z. Xiaosong, R. U. Khan, I. Ahad, and J. Kumar, "Malicious code detection based on image processing using deep learning," in *Proceedings of the 2018 International Conference on Computing and Artificial Intelligence*, Chengdu, China, March 2018, pp. 81-85, doi: 10.1145/3194452.3194459.

[47] A. Nguyen, J. Yosinski, and J. Clune, (2015). "Deep neural networks are easily fooled: High confidence predictions for unrecognizable images," in *Proceedings*

of the IEEE conference on computer vision and pattern recognition, Boston, MA, June 2015, pp. 427-436, doi: 10.1109/CVPR.2015.7298640.

[48] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," in *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, Abu Dhabi, UAE, April 2017, pp. 506-519, doi: 10.1145/3052973.3053009.

[49] A. N. Jahromi, S. Hashemi, A. Dehghantanha, R. M. Parizi, and K. K. R. Choo, "An enhanced stacked LSTM method with no random initialization for malware threat hunting in safety and time-critical systems," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 4, no. 5, pp. 630-640, June 2020, doi: 10.1109/TETCI.2019.2910243.

[50] Y. Tang, X. Liu, Y. Jin, H. Wei, and Q. Deng, "A Recurrent Neural Network-based Malicious Code Detection Technology," in *2019 IEEE 8th Joint International Information Technology and Artificial Intelligence Conference*, Chongqing, China, May 2019, pp. 1737-1742, doi: 10.1109/ITAIC.2019.8785580.

[51] R. Jozefowicz, O. Vinyals, M. Schuster, N. Shazeer, and Y. Wu, "Exploring the limits of language modeling," 2016, arXiv:1602.02410. [Online]. Available: https://arxiv.org/abs/1602.02410

[52] D. Soutner and L. Müller, "Application of LSTM Neural Networks in Language Modelling," in *16th International Conference on Text, Speech and Dialogue*, Pilsen, CZE, Sept. 2013, pp. 105-112, doi: 10.1007/978-3-642-40585-3_14.

[53] FireEye Commando VM 2.0. [Online]. Available: https://github.com/fireeye/commando-vm, Accessed on: 2020.

[54] VirusShare. [Online]. Available: https://www.virusshare.com, Accessed on: 2019.

[55] Oracle VM VirtualBox. [Online]. Available: https://www.virtualbox.org, Accessed on: 2020.

[56] x64dbg. [Online]. Available: https://x64dbg.com, Accessed on: 2020.

[57] R. Lu, “Malware Detection with LSTM using Opcode Language,” 2019, arXiv:1906.04593. [Online]. Available: https://arxiv.org/abs/1906.04593

[58] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane,R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zhengg, “TensorFlow: Large-scale machine learning on heterogeneous systems,” 2015.

[59] F. Chollet *et al.*, “Keras.” [Online]. Available: https://keras.io, 2015.

[60] J. Yoder, “Determining Optimum Drop-out Rate for Neural Networks,” *The Bridge, The Magazine of IEEE-Eta Kappa Nu*, vol. 115, no. 2, pp. 10-17, June 2018.

[61] D. P. Kingma, and J. Ba, “Adam: A Method for Stochastic Optimization,” 2014, arXiv:1412.6980. [Online]. Available: https://arxiv.org/abs/1412.6980

[62] T. Tieleman, and G. Hinton, (2012). “Lecture 6.5-Rmsprop: Divide The Gradient by a Running Average of Its Recent Magnitude,” *COURSERA: Neural Networks for Machine Learning*, vol. 4, no. 2, pp. 26-31, 2012.

[63] J. Duchi, E. Hazan, and Y. Singer, “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization,” *Journal of Machine Learning Research*, vol. 12, no. 7, pp. 2121-2159, July 2011.

[64] S. Ruder, “An Overview of Gradient Descent Optimization Algorithms,” 2016, arXiv:1609.04747. [Online]. Available: https://arxiv.org/abs/1609.04747

[65] N. Qian, “On The Momentum Term in Gradient Descent Learning Algorithms,” *Neural Networks*, vol. 12, no. 1, pp. 145-151, Jan 1999, doi:10.1016/S0893-6080(98)00116-6.

APPENDIX A

GRAPHS

A.1 The Graphs of Experimental Models on ISM

In this Appendix section, we share the results of 16 ISM and 17 BSM models which were generated during our study to specify optimal values for 4 parameters required for language modeling.

In Figure A.1 below, the relation between sequence length and accuracy in ISM models is shown.

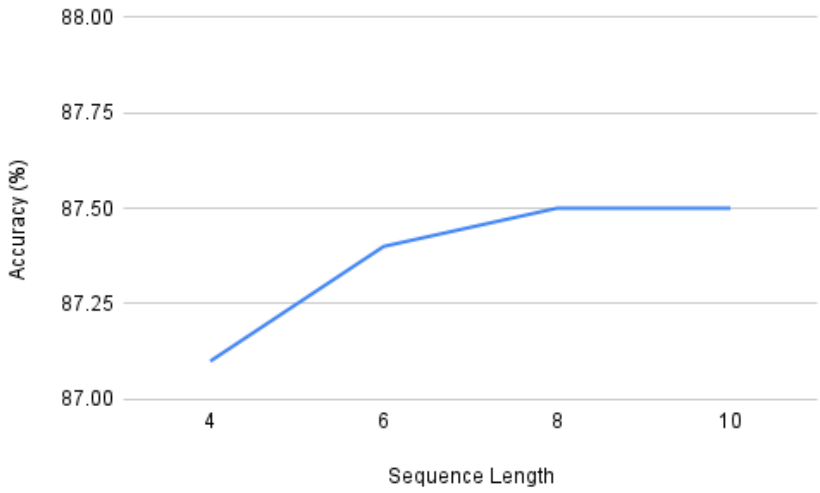

Figure A.1: Accuracy-Sequence Length

In Figure A.2 below, the relation between dropout rate and accuracy in ISM models is shown.

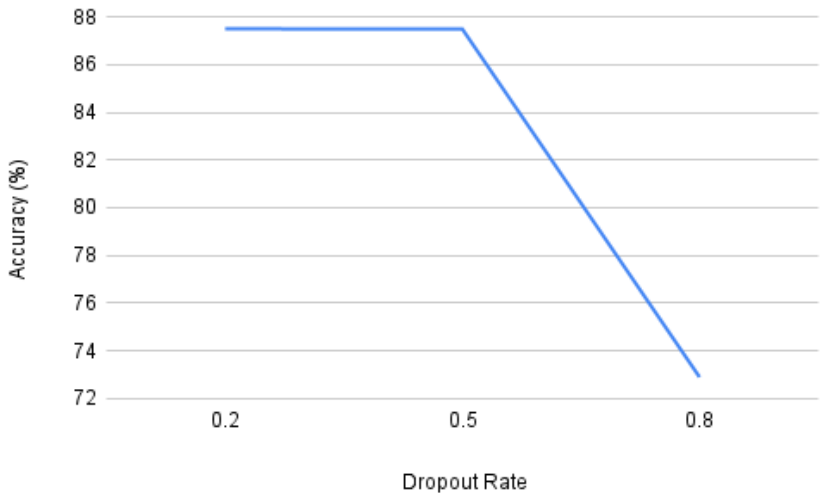

Figure A.2: Accuracy-Dropout Rate

In Figure A.3 below, the relation between type of optimizer and accuracy in ISM models is shown.

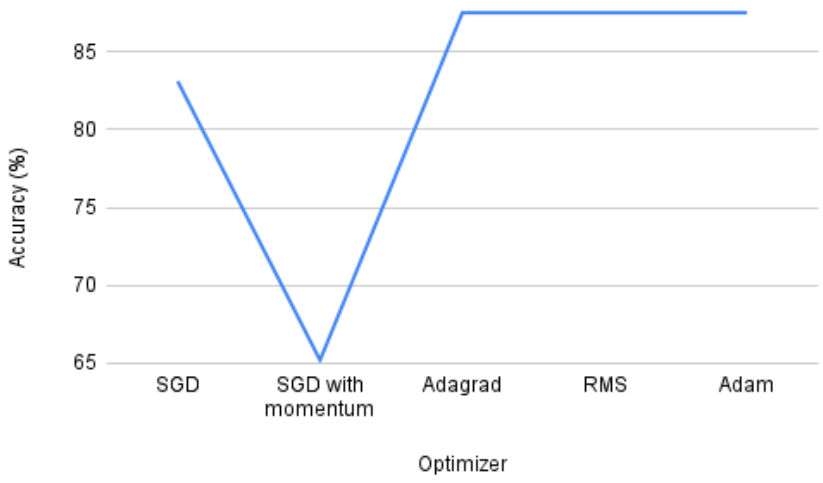

Figure A.3: Accuracy-Optimizer

In Figure A.4 below, the relation between number of nodes of LSTM layer and accuracy in ISM models is shown.

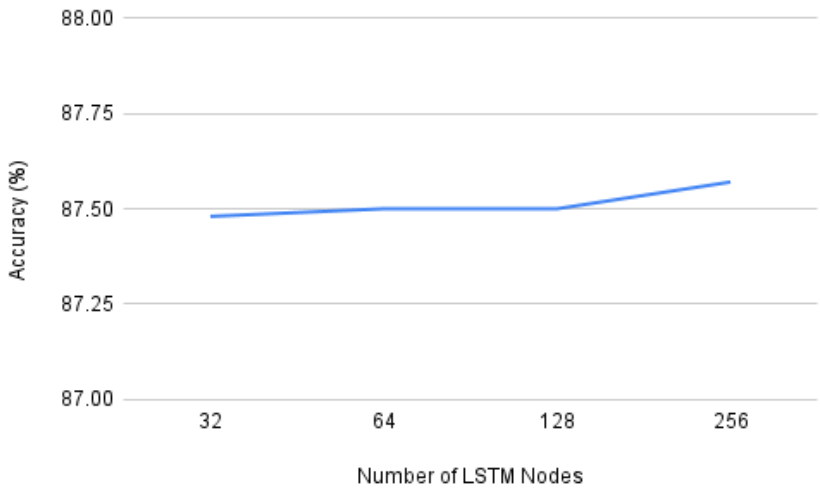

Figure A.4: Accuracy-Number of LSTM Nodes

A.2 The Graphs of Experimental Models on BSM

In Figure A.5 below, the relation between sequence length and accuracy in BSM models is shown.

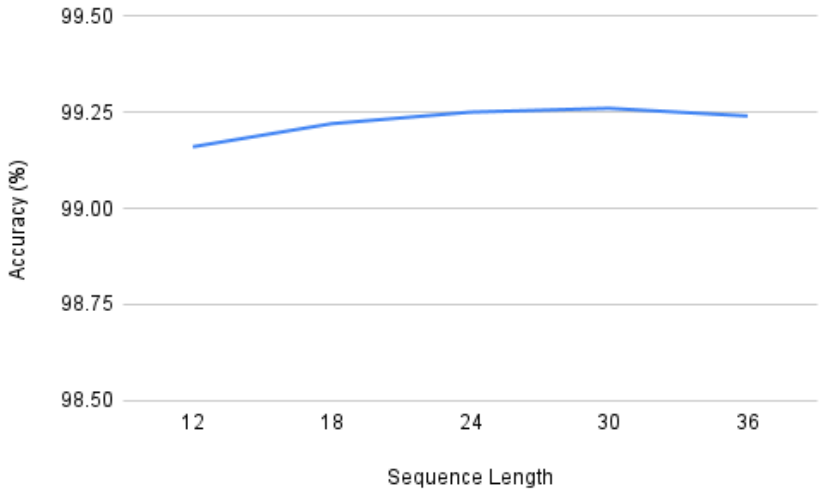

Figure A.5: Accuracy-Sequence Length

In Figure A.6 below, the relation between dropout rate and accuracy in BSM models is shown.

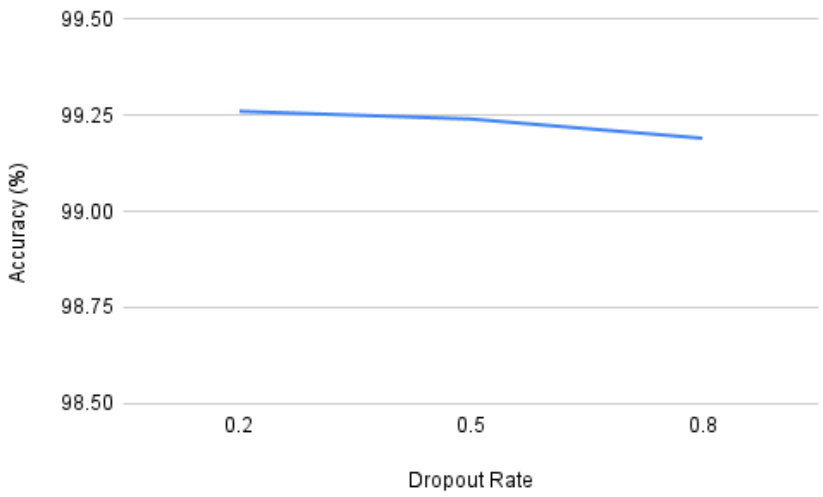

Figure A.6: Accuracy-Dropout Rate

In Figure A.7 below, the relation between type of optimizer and accuracy in BSM models is shown.

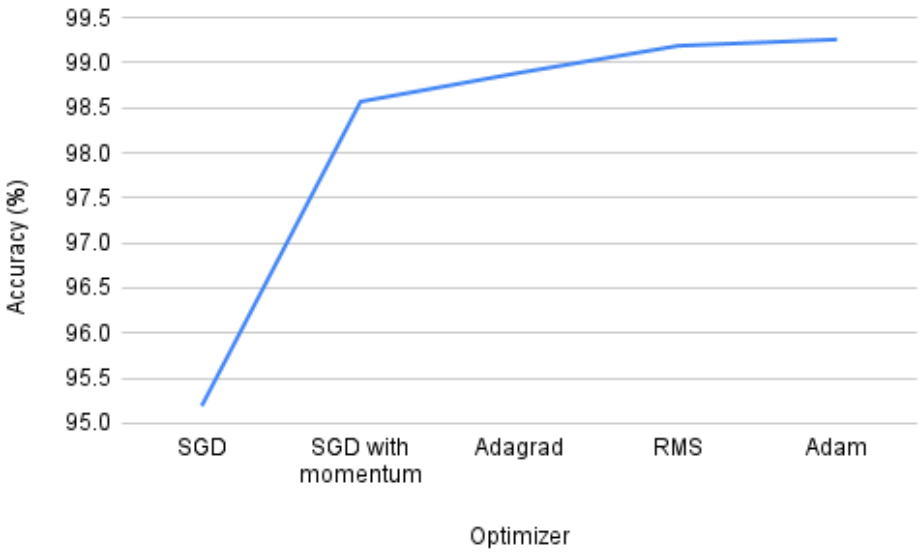

Figure A.7: Accuracy-Optimizer

In Figure A.8 below, the relation between number of nodes of LSTM layer and accuracy in ISM models is shown.

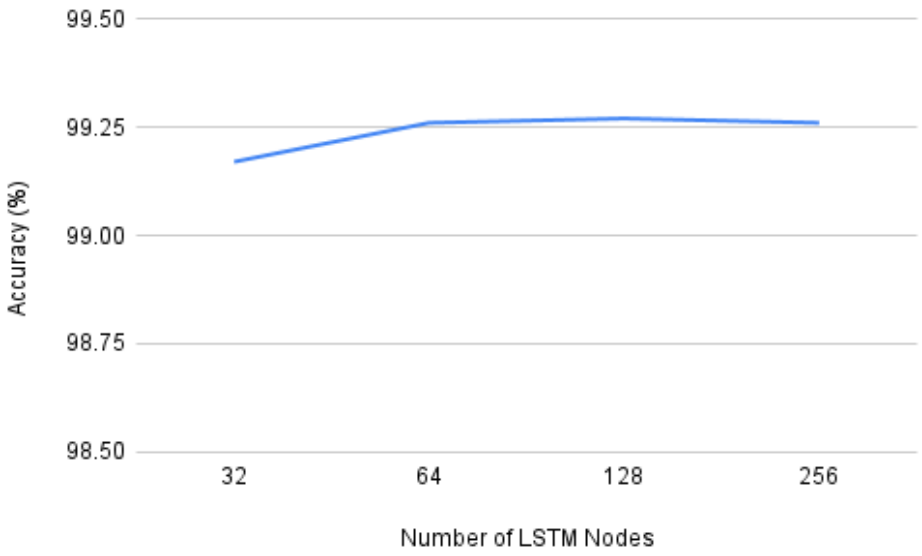

Figure A.8: Accuracy-Number of LSTM Nodes