

STATIC MALWARE DETECTION USING STACKED BI-DIRECTIONAL LSTM

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

BY

DENİZ DEMİRCİ

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

AUGUST 2021



Approval of the thesis:

**STATIC MALWARE DETECTION USING STACKED BI-DIRECTIONAL  
LSTM**

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



---

---

---

---

---

---

Date: 19.08.2021





**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: Deniz Demirci

Signature :

## **ABSTRACT**

### **STATIC MALWARE DETECTION USING STACKED BI-DIRECTIONAL LSTM**

Demirci, Deniz

M.S., Department of Cyber Security

Supervisor: Assoc. Prof. Dr. Cengiz Acartürk

August 2021, 63 pages

The recent proliferation in the use of the Internet and personal computers has made it easier for cybercriminals to expose Internet users to widespread and damaging threats. In order protect the end users against such threats, a security system must be proactive. It needs to detect malicious files or executables before reaching the end-user. To create an efficient and low-cost malware detection mechanism, in the present study, we propose stacked bidirectional long short-term memory (Stacked BiLSTM) based deep learning (DL) language model for detecting malicious code. We developed language models using assembly instructions from .text sections of malicious and benign Portable Executable (PE) files. We created our first dataset from assembly instructions obtained from static analysis of the PE files. The dataset was composed of text documents, and it was used in Document Level Analysis Model (DLAM). By splitting the first dataset into single instructions, we obtained the second dataset, which was then used in a Sentence Level Analysis Model (SLAM). We treated each instruction as a sentence, and .text sections as documents. We labeled each document and sentence by their corresponding malicious and benign tags. The experiments showed that the Document Level Analysis Model (DLAM), and the Sentence Level Analysis Model (SLAM) achieved 98,3% and 70.4% F1 scores, respectively.

**Keywords:** Malware Detection, static analysis, opcode, Stacked BiLSTM, NLP

## ÖZ

### YIĞINLANMIŞ ÇİFT YÖNLÜ UZUN-KISA SÜRELİ BELLEK KULLANARAK ZARARLI YAZILIM TESPİTİ

Demirci, Deniz

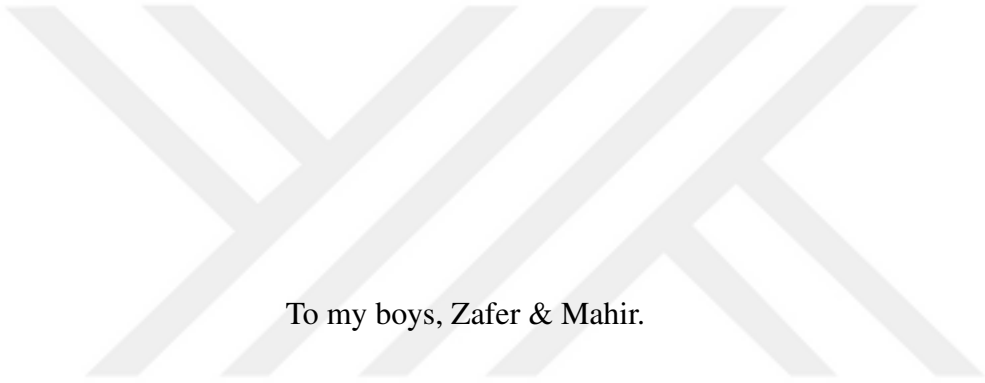
Yüksek Lisans, Siber Güvenlik Bölümü

Tez Yöneticisi: Doç. Dr. Cengiz Acartürk

Ağustos 2021 , 63 sayfa

İnternetin ve kişisel bilgisayarların son zamanlarda yaygınlaşması, siber suçluların İnternet kullanıcılarını yaygın ve zarar verici tehditlere maruz bırakmasını kolaylaştırdı. Son kullanıcıları bu tür tehditlere maruz bırakmamak için kullanılan güvenlik sistemlerinin proaktif olması, zararlı yazılımları ve çalıştırılabilir dosyaları son kullanıcıya ulaşmadan önce algılaması beklenir. Verimli ve düşük maliyetli bir zararlı yazılım algılama mekanizması oluşturmak için, bu çalışmada, yığılanmış çift yönlü uzun kısa süreli bellek (Yığılanmış BiLSTM) tabanlı derin öğrenme (DL) dil modeli öneriyoruz. Zararlı ve zararsız çalıştırılabilir dosyalardan elde ettiğimiz assembly talimatlarını kullanarak modeller geliştirdik. Bu modellerde, assembly talimatlarını cümle, kod bölümlerini ise belge olarak ele aldık. Yapılan denemeler, Belge Düzeyinde Analiz Modelinin (DLAM) ve Cümle Düzeyinde Analiz Modelinin (SLAM) sırasıyla 98,3% ve 70.4% F1 doğruluk puanına ulaştığını gösterdi.

Anahtar Kelimeler: Zararlı Yazılım Tespit, statik analiz, opcode, yığılanmış BiLSTM, NLP



To my boys, Zafer & Mahir.

## ACKNOWLEDGEMENTS

First, I would like to thank Assoc. Prof. Dr. Cengiz Acartürk, my supervisor, for his guidance, support, encouragement, and most of all, his patience throughout the entire process.

I would like to thank the members of our malware analysis research group, Melih Şırlancı, and Nazenin Şahin for their support in the scope of this study.

I truly appreciate my commander, Col. Adnan Gürbüz, who encouraged me to start this journey. Lastly, I would like to thank my beloved wife, Özge, for being there for me during the challenges in the graduate school, my parents for their support and encouragement.



## TABLE OF CONTENTS

|                                                 |      |
|-------------------------------------------------|------|
| ABSTRACT . . . . .                              | iv   |
| ÖZ . . . . .                                    | v    |
| ACKNOWLEDGEMENTS . . . . .                      | vii  |
| TABLE OF CONTENTS . . . . .                     | viii |
| LIST OF TABLES . . . . .                        | xi   |
| LIST OF FIGURES . . . . .                       | xiii |
| LIST OF ABBREVIATIONS . . . . .                 | xv   |
| CHAPTERS                                        |      |
| 1 INTRODUCTION . . . . .                        | 1    |
| 1.1 Motivation and Problem Definition . . . . . | 2    |
| 1.2 Research Questions . . . . .                | 3    |
| 1.3 Structure of the Thesis . . . . .           | 3    |
| 2 BACKGROUND AND RELEVANT WORK . . . . .        | 5    |
| 2.1 Text Vectorization . . . . .                | 5    |
| 2.1.1 One Hot Encoding . . . . .                | 5    |
| 2.1.2 Count Vectorizer . . . . .                | 6    |
| 2.1.3 TF-IDF Vectorizer . . . . .               | 8    |
| 2.1.4 Keras TextVectorization . . . . .         | 8    |

|         |                                                                       |    |
|---------|-----------------------------------------------------------------------|----|
| 2.1.5   | Word Embedding . . . . .                                              | 9  |
| 2.1.6   | Word2Vec . . . . .                                                    | 9  |
| 2.1.7   | Document Vectorization . . . . .                                      | 11 |
| 2.1.8   | Sentiment Analysis and Text Classification . . . . .                  | 11 |
| 2.1.9   | RNN, LSTM vs BiLSTM . . . . .                                         | 12 |
| 2.2     | Malware Detection Methods . . . . .                                   | 14 |
| 2.2.1   | Signature-Based Detection Methods . . . . .                           | 14 |
| 2.2.2   | Behavior-Based Detection Methods . . . . .                            | 15 |
| 2.2.3   | Machine Learning and Deep Learning Based Detection Methods            | 16 |
| 2.3     | Learning Methods in Malware Detection . . . . .                       | 17 |
| 2.4     | Summary . . . . .                                                     | 19 |
| 3       | METHODOLOGY . . . . .                                                 | 21 |
| 3.1     | The Approach . . . . .                                                | 21 |
| 3.2     | Data Collection and Formatting . . . . .                              | 22 |
| 3.2.1   | Obtaining Assembly Instructions . . . . .                             | 22 |
| 3.2.2   | Standardization and Tokenization . . . . .                            | 23 |
| 3.2.3   | Splitting data into the training set, testing set, and validation set | 26 |
| 3.3     | The Proposed Model . . . . .                                          | 28 |
| 3.3.1   | Setup Environment . . . . .                                           | 28 |
| 3.3.2   | Imported Libraries and Modules . . . . .                              | 28 |
| 3.3.3   | Training and Testing Pipeline . . . . .                               | 30 |
| 3.3.4   | Parameters and Tuning . . . . .                                       | 31 |
| 3.3.4.1 | Dropout and Regularization Methods . . . . .                          | 31 |

|         |                                                        |    |
|---------|--------------------------------------------------------|----|
| 3.3.4.2 | Optimizers and loss functions . . . . .                | 32 |
| 3.3.4.3 | Pooling Layer . . . . .                                | 32 |
| 3.3.4.4 | Overcoming The Overfitting . . . . .                   | 32 |
| 3.4     | The Document Level Analysis Model (The DLAM) . . . . . | 32 |
| 3.5     | The Sentence Level Analysis Model (The SLAM) . . . . . | 39 |
| 3.6     | Summary . . . . .                                      | 43 |
| 4       | RESULTS . . . . .                                      | 45 |
| 4.1     | Evaluation Criteria . . . . .                          | 45 |
| 4.2     | SLAM (Sentence Level Analysis Model) . . . . .         | 46 |
| 4.3     | DLAM (Document Level Analysis Model) . . . . .         | 47 |
| 4.4     | Comparison . . . . .                                   | 48 |
| 4.5     | Discussion . . . . .                                   | 49 |
| 4.6     | Open Problems . . . . .                                | 50 |
| 5       | CONCLUSION AND FUTURE WORK . . . . .                   | 53 |
| 5.1     | Conclusion . . . . .                                   | 53 |
| 5.2     | Limitations and Future Work . . . . .                  | 54 |
|         | REFERENCES . . . . .                                   | 55 |
|         | APPENDICES . . . . .                                   | 60 |
| A       | RESERVED SECTIONS IN PE . . . . .                      | 61 |
| B       | TEXT VECTORIZATION . . . . .                           | 63 |
| B.1     | Sample Text Vector . . . . .                           | 63 |

## LIST OF TABLES

### TABLES

|            |                                                                        |    |
|------------|------------------------------------------------------------------------|----|
| Table 2.1  | One-Hot Encoding Sample . . . . .                                      | 6  |
| Table 2.2  | Count Vectorizer Sample . . . . .                                      | 7  |
| Table 2.3  | Count Vectorizer with n-grams Sample . . . . .                         | 7  |
| Table 3.1  | Sample from the SLAM Dataset . . . . .                                 | 25 |
| Table 3.2  | Total Number of Samples for models . . . . .                           | 26 |
| Table 3.3  | Dataset Sizes for training, validation and testing . . . . .           | 27 |
| Table 3.4  | Required Python libraries . . . . .                                    | 29 |
| Table 3.5  | Model summary for the initial DLAM. . . . .                            | 33 |
| Table 3.6  | Parameters For The DLAM . . . . .                                      | 34 |
| Table 3.7  | The effects of BiLSTM Layers on Validation Loss Change . . . . .       | 34 |
| Table 3.8  | The effects of Sentence Length on Validation Losses . . . . .          | 35 |
| Table 3.9  | The effects of n-grams on Validation Losses . . . . .                  | 36 |
| Table 3.10 | Model summary for the proposed DLAM. . . . .                           | 37 |
| Table 3.11 | Parameter Values for The SLAM . . . . .                                | 40 |
| Table 3.12 | Model summary for the initial SLAM. . . . .                            | 41 |
| Table 3.13 | The effects of Maximum Sentence Lengths on Validation Losses . . . . . | 41 |

|                                                                              |    |
|------------------------------------------------------------------------------|----|
| Table 3.14 The effects of Embedding Dimensions on Losses . . . . .           | 42 |
| Table 3.15 Best Results for TextVectorization parameters on the SLAM . . . . | 42 |
| Table 3.18 Effects of Word2Vec parameters on the SLAM . . . . .              | 42 |
| Table 3.16 Word2Vec Parameters for the SLAM . . . . .                        | 43 |
| Table 3.17 Model summary for the initial SLAM with Word2Vec. . . . .         | 43 |
| Table 3.19 Model summary for the SLAM. . . . .                               | 44 |
| Table 3.20 Effects of DistilBERT parameters on the SLAM . . . . .            | 44 |
| Table 4.1 F1 Score Calculation of The SLAM . . . . .                         | 47 |
| Table 4.2 F1 Score Calculation of The DLAM . . . . .                         | 48 |
| Table 4.3 Performances of The Models . . . . .                               | 48 |
| Table 4.4 Comparison of Models Used to Detect Malware . . . . .              | 50 |
| Table 4.5 Seed Value Effect on Performance . . . . .                         | 52 |
| Table A.1 Reserved Sections in PE Header . . . . .                           | 61 |

## LIST OF FIGURES

### FIGURES

|            |                                                                                |    |
|------------|--------------------------------------------------------------------------------|----|
| Figure 2.1 | WordEmbedding, redrawn based on (Bengio et al., 2000). . . . .                 | 9  |
| Figure 2.2 | CBOW & SkipGrams in Word2Vec . . . . .                                         | 10 |
| Figure 2.3 | RNN Basic Architecture, redrawn based on (Donahue et al., 2014) . . . . .      | 13 |
| Figure 2.4 | LSTM Basic Architecture, redrawn based on (Graves & Jaitly, 2014). . . . .     | 14 |
| Figure 2.5 | BiLSTM Basic Architecture, redrawn based on (Cornegruta et al., 2016). . . . . | 15 |
| Figure 3.1 | Language Modeling Pipeline. . . . .                                            | 23 |
| Figure 3.2 | Sample Document from DLAM Dataset. . . . .                                     | 24 |
| Figure 3.3 | Text Processing for the DLAM. . . . .                                          | 24 |
| Figure 3.4 | Text Processing for the SLAM. . . . .                                          | 26 |
| Figure 3.5 | Directory Structure for Dataset . . . . .                                      | 27 |
| Figure 3.6 | Convergence Without Variational RNN in the DLAM . . . . .                      | 35 |
| Figure 3.7 | Convergence With Variational RNN in the DLAM . . . . .                         | 36 |
| Figure 3.8 | Overfitting Sample on the DLAM . . . . .                                       | 38 |
| Figure 3.9 | Sentence Length Distribution for the SLAM . . . . .                            | 39 |

|            |                                             |    |
|------------|---------------------------------------------|----|
| Figure 4.1 | The Confusion Matrix for reference. . . . . | 46 |
| Figure 4.2 | The SLAM Confusion Matrix. . . . .          | 46 |
| Figure 4.3 | The DLAM Confusion Matrix. . . . .          | 47 |



## LIST OF ABBREVIATIONS

|         |                                             |
|---------|---------------------------------------------|
| ANN     | Artificial Neural Network                   |
| CNN     | Convolutional Neural Network                |
| RNN     | Recurrent Neural Network                    |
| LSTM    | Long-Short Term Memory                      |
| BiLSTM  | Bidirectional Long-Short Term Memory        |
| DNN     | Deep Neural Network                         |
| DL      | Deep Learning                               |
| ML      | Machine Learning                            |
| NLP     | Natural Language Processing                 |
| SA      | Sentiment Analysis                          |
| OM      | Option Mining                               |
| ADAM    | Adaptive Moment Estimation                  |
| RMSProp | Root Mean Square Propagation                |
| TF-IDF  | Term Frequency - Inverse Document Frequency |
| BOW     | Bag Of Words                                |
| CBOW    | Continuous Bag Of Words                     |
| DBOW    | Distributed Bag of Words                    |
| SGD     | Stochastic Gradient Descent                 |
| PV-DM   | Paragraph Vector—Distributed Memory         |
| NIDS    | Network-based Intrusion Detection Systems   |
| HIDS    | Host-Based Intrusion Detection Systems      |
| PE      | Portable Executable                         |
| COFF    | Common Object File Format                   |
| API     | Application Programming Interface           |

|       |                                                         |
|-------|---------------------------------------------------------|
| DLL   | Dynamic-link Library                                    |
| CFG   | Control Flow Graphs                                     |
| NDFS  | Nonnegative Discriminative Feature Selection            |
| CGSSL | Clustering-guided Sparse Structural Learning            |
| AUC   | Area Under Curve                                        |
| SLAM  | Sentence Level Analysis Model                           |
| DLAM  | Document Level Analysis Model                           |
| GloVe | Global Vectors for Word Representation                  |
| Bert  | Bidirectional Encoder Representations from Transformers |



## CHAPTER 1

### INTRODUCTION

The recent increase in Internet usage and personal computers has made it easier for cybercriminals to expose Internet users to widespread and damaging cyber threats. The exposition of the users has led cybercriminals to make profits or create damage on a massive scale. Users have faced viruses, worms, spyware, adware, and ransomware, constituting subclasses of malicious software. Those threats compromise the user systems using various attack vectors, including email phishing, software vulnerabilities, freeware, email attachments. Antiviruses, antimalware, host-based intrusion detection systems (HIDS), and network-based intrusion detection systems (NIDS) are widely deployed malware mitigation techniques (Moon et al., 2014). However, due to malware's dynamic nature and growing sophistication, mitigations are usually insufficient to provide sustainable protection. "Proactive protection", which means to detect malicious files or executables before reaching the end-user, may be a solution to prevent infection stemmed from malicious files in IT systems. In daily settings, we frequently use the term 'malware' to address files that perform malicious activities. Those activities include violations of security policies of use, such as unauthorized access or authentication, privilege escalation, and unauthorized disclosure of information about the target machine, its users, or components (Elisan, 2012). Despite the general conceptualization of the term 'malware' as malicious files, malicious code pieces are usually parts of a file rather than the file itself. In other words, malware code pieces are typically wrapped into executable files (Scarfone & Souppaya, 2013), namely payloads. From a system-level perspective, malicious lines of a software code are basic units that run a series of instructions at the machine level. In other words, malware refers to the commands that run instructions for malicious purposes. These instructions may perform system calls for input-output functions and a set of functions that operate computer memory and file systems. Accordingly, the foremost challenge in malware detection is identifying pieces of code in a file so that the suspected file that includes the malware has malicious functionality. In practice, malware detection methods rely on signature databases and YARA<sup>1</sup> rules. The signature databases are used to match against a signature generated from a newly encountered executable. Nevertheless, the malware's self-modifying abilities limit the detection capabilities of these methods so as not to confuse it for a benign file. In order not to miss malicious activities, it is crucial to focus on lines of codes which are the parts that express more functionality in the suspected files. However, since source codes for executable files are not usually available in compiled form, the assembly instructions are the best candidate to unveil malicious functionality in a suspected file. Researchers and practitioners have proposed various techniques that may be classified into three major

---

<sup>1</sup> YARA: <https://virustotal.github.io/yara/> (retrieved on 19 Mar 2019)

groups to address such threats: static, dynamic, and hybrid analysis (Vinayakumar et al., 2021). Dynamic analysis executes the file for malware detection, while static analysis aims to detect malware by scanning the entire file without running the executable. Static analysis has some drawbacks against dynamic analysis (Acarturk et al., 2021) in resisting malicious deformation techniques such as obfuscation and dynamic code loading. However, it consumes fewer resources, identifies malware efficiently, and mitigates it before reaching end-users or servers. Moreover, static analysis is scalable and usable when facing batch unknown malware detection and may traverse all possible execution paths of the executable file. With the maturity of machine learning, Natural Language Processing (NLP) practices, and open-source software, tools have been developed and made accessible (e.g., TensorFlow<sup>2</sup>, OpenAI<sup>3</sup>, Gensim<sup>4</sup>, NLTK<sup>5</sup>, and OpenNLP<sup>6</sup>). Those tools may help malware researchers use and configure NLP techniques in malware detection. Briefly, NLP refers to analyzing texts by automated means. Thus, in the present study, we propose using NLP techniques, particularly document and sentence level polarity detection on assembly instruction<sup>7</sup> sequences with a deep learning model using stacked bidirectional long short-term memory (BiLSTM) to detect malicious files.

## 1.1 Motivation and Problem Definition

Technological advances have provided a suitable environment for cybercriminals to devise new malware functionalities, types, and increasing effects in daily life. The creation and distribution of intelligent and sophisticated malware have become easier through Smartphones, IoT devices, and, more generally, connected devices on the Internet. Nowadays, users come across malware and ransomware with malicious purposes, such as consuming the victims' computing power to mine crypto-coins or locking all the files on the infected system for ransom (Richardson & North, 2017). To reduce the risks due to malware, antimalware developers and researchers have to keep up with the development speed and expansion of malware propagation. For this, researchers have been building detection systems that do not rely only on the experts' knowledge of the malware domain (cf. signature-based systems) but also adaptive learning systems that rely on Machine Learning (ML) techniques. Since machine learning created a profound shift in many areas, including cybersecurity (Salloum et al., 2020), AI-powered antimalware tools have a high potential to detect modern malware types and attacks, improve scanning engines, reinforce overall cybersecurity, and create proactive systems (Gibert et al., 2020). A proactive system that will impede an infection before it reaches end-users must identify the threats, then mitigate them efficiently. To achieve this goal, we considered using NLP techniques in the detection phase of malware mitigation. In the present study, since malicious and benign executables contain the same grammar, semantics, syntax, and vocabulary to express their intentions, we treat malware detection as a polarity detection problem in terms

<sup>2</sup> Tensorflow: <https://www.tensorflow.org/> (retrieved on 19 Mar 2019)

<sup>3</sup> OpenAI: <https://openai.com/> (retrieved on 19 Mar 2019)

<sup>4</sup> Gensim: <https://github.com/RaRe-Technologies/gensim> (retrieved on 19 Mar 2019)

<sup>5</sup> NLTK: <https://www.nltk.org/> (retrieved on 19 Mar 2019)

<sup>6</sup> OpenNLP: <https://opennlp.apache.org/> (retrieved on 19 Mar 2019)

<sup>7</sup> An assembly instruction is human readable(mnemonic) format of binary opcode and operands in assembly language. Such as *cmp ebx, eax*

of sentiment analysis in NLP. With the help of Bidirectional wrapper over LSTM, we aim to create a language model using assembly instructions. This model is designed to effectively learn and extract the features and characteristics of assembly language and may be used to classify files. Overall, language models in NLP have numerous applications in use, including speech recognition, machine translation, tagging, optical character recognition, and sentiment analysis (Torfi et al., 2020). In a sentiment analysis or sentiment classification task, the goal is to resolve a judgment's polarity in a document, sentence, or feature/aspect level, whether positive, negative, or neutral. Similarly, we disassembled both benign and malicious files in the same routine, and both shared the exact instructions. We treat benign and malicious files as if they are written in the same language though with different intentions, as in sentiment analysis. We statically gathered assembly instructions in the present study and used text processing methods to extract assembly language features and characteristics. Since BiLSTM is more advantageous in multiple ways, such as resolving issues relevant to vanishing gradients and exploding gradients (Sherstinsky, 2018a), (Pascanu et al., 2012), we chose it over LSTM and RNN (Recurrent Neural Network) systems to create an assembly language model. We then used the language model to classify files and sentences as malicious or benign.

## **1.2 Research Questions**

The major research question of the present study is to investigate the potential of deep learning language models for the identification of malicious code. For this, we statically collected assembly instructions from malicious and benign (Portable Executable) PE files. The goal of the models is to identify the intention of the code both at the instruction (sentence) level and the file (document) level. More specifically, our goal is to investigate the effects of instruction-level and document-level datasets in terms of the models' malware detection performance.

## **1.3 Structure of the Thesis**

Our study consists of five chapters. We present the background to shed light on the concepts associated with our research. We offer the pertinent findings in malware detection and language modeling in Chapter 2 (p. 5). Then, we propose our malware detection pipeline and architecture in Chapter 3 (p. 21). Later we report our results in Chapter 4 (p. 45), and finally, we report the limitations of our research and present our conclusion in Chapter 5 (p. 53).



## CHAPTER 2

### BACKGROUND AND RELEVANT WORK

This chapter first presents the methods applied in preparing text content for machine learning, such as text vectorization and word embedding. Then, we describe the tasks performed using NLP, for example, text classification and sentiment analysis. After that, we give a brief description of RNN, LSTM, and BiLSTM. Lastly, we summarize the approaches developed for detecting malware.

#### 2.1 Text Vectorization

Machine learning models operate on numeric features by taking vectors (arrays of numbers) as input. In this array, rows contain instances, and columns contain features. To apply machine learning algorithms to assembly instructions, we need to transform our assembly code into vector representations. The transforming process is the feature extraction step (Lewis, 2000), and it is essential for creating a language model. The text used to form a language model may be a set of documents (corpus), a single document, or words of different lengths. The feature is each property of the vector representation (John, 2017). For the present study, features represent assembly instructions and the relation between the opcode and operands. To extract the features, we applied text analysis and text preprocessing techniques to our raw data. Hence, our input assembly code features define a feature space specific to the assembly language model on which we later apply machine learning methods. There are alternative ways to represent text in numeric form. We examine one-hot encoding, count vectorizer, TF-IDF vectorizer, hash vectorizer, *word2vec* and *TextVectorization* in the Keras<sup>1</sup>.

##### 2.1.1 One Hot Encoding

One-hot encoding creates a vector representation of the words used in the text without ordering. Since the ordering is highly related with context, the loss of order causes the context to vanish. The vector representation created by one-hot encoding consists of binary values. To illustrate, we used a simple function prologue<sup>2</sup> and represented the one-hot encoding form of it. The typical function prologue created by Gnu Compiler

---

<sup>1</sup> The Keras: <https://keras.io> (retrieved on 19 Mar 2019)

<sup>2</sup> Function Prologue: [https://en.wikipedia.org/wiki/Function\\_prologue](https://en.wikipedia.org/wiki/Function_prologue) (retrieved on 20 May 2020)

Collection (GCC<sup>3</sup>) in assembly language consists of a sequence of *'mov ebp, esp. push ebp. sub esp 0xd'*. To create one-hot encoding we need each word in a sentence, so, we tokenize the instruction and get an array of words, like *['0xd', 'ebp', 'esp', 'mov', 'push', 'sub']*. Then we assign each word an integer value, to create integer encoded representation. For example, in American Standard Code for Information Interchange (ASCII<sup>4</sup>) order of words, we may assign 0 for *0xffff*, 1 for *ebp*, etc. The final integer encoded form of our input becomes, *[3, 1, 2, 4, 1, 5, 2, 0]*. Since one-hot encoding is a sparse representation, and there are eight words, and the total unique word count is six, we create a two-dimensional array with eight rows and six columns. We represent this table in Table 2.1. There are two significant points to deduce from

Table 2.1: One-Hot Encoding Sample

|       | words | 0xd | ebp | esp | mov | push | sub |
|-------|-------|-----|-----|-----|-----|------|-----|
| words | Index | 0   | 1   | 2   | 3   | 4    | 5   |
| mov   | 0     | 0   | 0   | 0   | 1   | 0    | 0   |
| ebp   | 1     | 0   | 1   | 0   | 0   | 0    | 0   |
| esp   | 2     | 0   | 0   | 1   | 0   | 0    | 0   |
| push  | 3     | 0   | 0   | 0   | 0   | 1    | 0   |
| ebp   | 4     | 0   | 1   | 0   | 0   | 0    | 0   |
| sub   | 5     | 0   | 0   | 0   | 0   | 0    | 1   |
| esp   | 6     | 0   | 0   | 1   | 0   | 0    | 0   |
| 0xd   | 7     | 1   | 0   | 0   | 0   | 0    | 0   |

the table 2.1. First, the sparse representation may be used to construct the assembly sequence from top to bottom. Therefore, using one-hot encoding, it is possible to represent sentences or documents. Second, each word may be represented as an array of integers. Such as, *mov* opcode becomes, *[0, 0, 0, 1, 0, 0]* and *ebp* operand becomes *[0, 1, 0, 0, 0, 0]*. We created the samples in this subsection using the code<sup>5</sup> in github repository.

### 2.1.2 Count Vectorizer

Count vectorizer is another way to represent a group of text documents and build a vocabulary of known words. As in one-hot encoding, a column represents each word, a row represents each text from the document, in the vector representation. The number of occurrences of a word is represented in each cell in the matrix. As

<sup>3</sup> GCC: <https://gcc.gnu.org> (retrieved on 20 May 2020)

<sup>4</sup> ASCII: <https://en.wikipedia.org/wiki/ASCII> (retrieved on 20 May 2020))

<sup>5</sup> one\_hot\_encoding.py: [https://github.com/d\protect\discretionary{\char\hyphenchar\font}{}{}demirci/binary\\_classification/blob/master/one\\_hot\\_encoding.py](https://github.com/d\protect\discretionary{\char\hyphenchar\font}{}{}demirci/binary_classification/blob/master/one_hot_encoding.py)

in one-hot encoding, the words in the text are not stored as strings; each word is given a particular index value. Hence, in Table 2.2 below, it is possible to say that File1 contains two *push*, one *mov*, one *sub* opcode with two *ebp*, two *esp*, and a *0xd* operand. But because the ordering and other semantic information are lost, it is not always true to assert that File1 consists of '*mov ebp, esp. push ebp. sub esp 0xd*'. This representation is also known as a sparse matrix.

Table 2.2: Count Vectorizer Sample

|       | words            | 0xd | ebp | esp | mov | push | sub |
|-------|------------------|-----|-----|-----|-----|------|-----|
| words | Integer Encoding | 0   | 1   | 2   | 3   | 4    | 5   |
| File1 | 0                | 1   | 2   | 2   | 1   | 1    | 1   |
| File2 | 1                | 0   | 1   | 1   | 1   | 0    | 0   |

Although vector size increases, the usage of n-grams in count-vectorizer, as in Table 2.3, provides the necessary baseline to predict the next word, hence understanding the context.

Table 2.3: Count Vectorizer with n-grams Sample

|       |          | mov ebp | ebp esp | esp push | push ebp | ebp sub | sub esp | esp 0xd |
|-------|----------|---------|---------|----------|----------|---------|---------|---------|
|       | Int.Enc. | 0       | 1       | 2        | 3        | 4       | 5       | 6       |
| File1 | 0        | 1       | 1       | 0        | 1        | 0       | 1       | 1       |
| File2 | 1        | 1       | 1       | 0        | 0        | 0       | 0       | 0       |

With the help of Table 2.3, it is possible to infer that File1 contains opcode sequence like '*mov ebp, esp. push ebp. sub esp 0xd*', and File2 contains '*mov ebp, esp*' using chaining method. Nevertheless, the inability to identify more critical or less essential words, considering the words plenteous in a corpus as the most statistically meaningful word, and the failure to identify relationships among words are the downsides of count-vectorizers. We created the samples in this subsection using the code<sup>6</sup> in github repository.

<sup>6</sup> count\_vectorizer.py: [https://github.com/d-demirci/binary\\_classification/blob/master/count\\_vectorizer.py](https://github.com/d-demirci/binary_classification/blob/master/count_vectorizer.py)

### 2.1.3 TF-IDF Vectorizer

Term Frequency - Inverse Document Frequency (TF-IDF) is a statistic calculated using the number of occurrences of a word in the corpus with a numerical exemplification of the significance of a word. Unlike Count Vectorizers, the TF-IDF focuses on the frequency of words and keeps the importance of the terms. This way, TF-IDF helps us decide which words are less important during our analysis and what to ignore. Hence, TF-IDF makes models less resource-demanding by reducing the dimensions of input. TF-IDF scores the words according to the statistics calculated. TF-IDF gives low scores to the terms that are abundant or too rare so the importance of the words in a corpus is not related to the abundance or the rareness of the word. A higher value of TF-IDF means higher importance of the words in the corpus, while lower values are of lesser extent. While the efficiency of TF-IDF compared to other vectorization methods depends on multiple factors, the comparative study (Shahmirzadi et al., 2018), claims that TF-IDF is well suited for the text similarity detection tasks with its efficiency and manageable aspects. The next subsection describes our preferred method, Keras TextVectorization, which also includes TF-IDF implementation to represent document in the present study.

### 2.1.4 Keras TextVectorization

The Keras TextVectorization class has options to manage text in a deep learning model constructed using the Keras framework. It facilitates the transformation of a batch of strings into a list of token indices or a dense representation. To create a vocabulary list using the set of strings, we may use the adapt method in TextVectorization class. This method analyzes the input and calculates the frequency of individual string values also, it uses the number of unique values while creating vocabulary set. But, if the dataset contains more unique values than the maximum vocabulary size parameter, unlike TF-IDF, by default, it uses the most frequent terms to create vocabulary. It is also configurable with parameters to use TF-IDF to create vector space models of the documents. By employing TextVectorization class, it is possible to process each sample for standardization purposes applying the following steps.

- Standardize each sample generally by lowercasing and stripping the punctuation,
- Split each sample into words using whitespace characters unless told otherwise,
- Recombine the words into tokens by using n-grams,
- Create a unique integer value for each token and associate with it,
- Transform each sample using this integer value, index, to a vector representation consisting of integer values or to a dense float vector.

Briefly, the standard methods needed to preprocess text in a corpus are compactly provided by this class.

### 2.1.5 Word Embedding

Word embeddings in text analysis mean a vector representation of words. It differs from previous vectorization methods by addressing semantic relations between words based on their distributional properties. The closeness of words forms the basis of this technique. In 2003, researchers used the term word embeddings initially and trained word embeddings in a neural language model. Since then, word embeddings have become widely used in NLP tasks such as the sentiment analysis process. The neural language model proposed by Bengio (Bengio et al., 2000) consisted of the layers such as embedding layer, intermediate layer, and SoftMax layer, as in Figure 2.1.

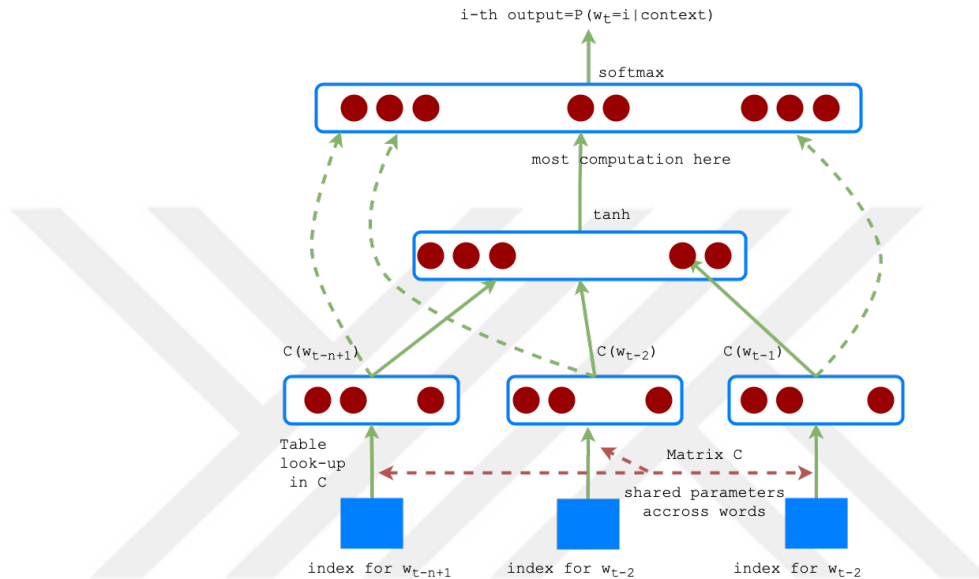


Figure 2.1: WordEmbedding, redrawn based on (Bengio et al., 2000).

The SoftMax layer, which outputs a probability distribution over all unique words in corpus and since the number of words may be millions, was the main bottleneck of the model. Therefore, in 2008, Collobert and Weston (Collobert & Weston, 2008) showed that word embeddings trained on a large dataset could hold syntactic and semantic meaning and contribute to performance gains on NLP tasks. Their solution to the expensive SoftMax calculation was to use an alternative objective function instead of cross-entropy. This way, given the previous words, they maximized the accuracy in predicting the next term. The next subsection gives a brief description about Word2Vec, which we use to create the sentence level analysis model.

### 2.1.6 Word2Vec

By definition, a Word Vector is a layered artificial neural network that is capable of learning how to represent each word with a real number vector by conserving its semantic features, thus allowing similar words to group into a single vector (Rong, 2014). Word2Vec offers a range of models to represent words in an n-dimensional

space. The original paper of Word2Vec (Mikolov et al., 2013) describes the approaches for creating Word2Vec representations of a text corpus. Those methods are CBOW (Continuous Bag of Words) and Skip Grams. In the simplest form of Bag-Of-Words (BOW), a text (document, sentence, or assembly instructions) is represented as the multiset of its words without considering grammar and word order. With the Bag-of-words model, the term frequency, explicitly, the number of times an expression appears in the corpus, total unique expression may calculated. In CBOW, to represent the continuity of relationships among the words, the window size is utilized to use both the (n) words before and after the target word to predict it, the center word, as shown in Figure 2.2. The second approach in Word2Vec, to create word embedding, is Skip Grams (Figure 2.2). On the contrary to CBOW, which focus on the surrounding terms to predict the center word, Skip Gram focuses on the central word to predict surrounding words.

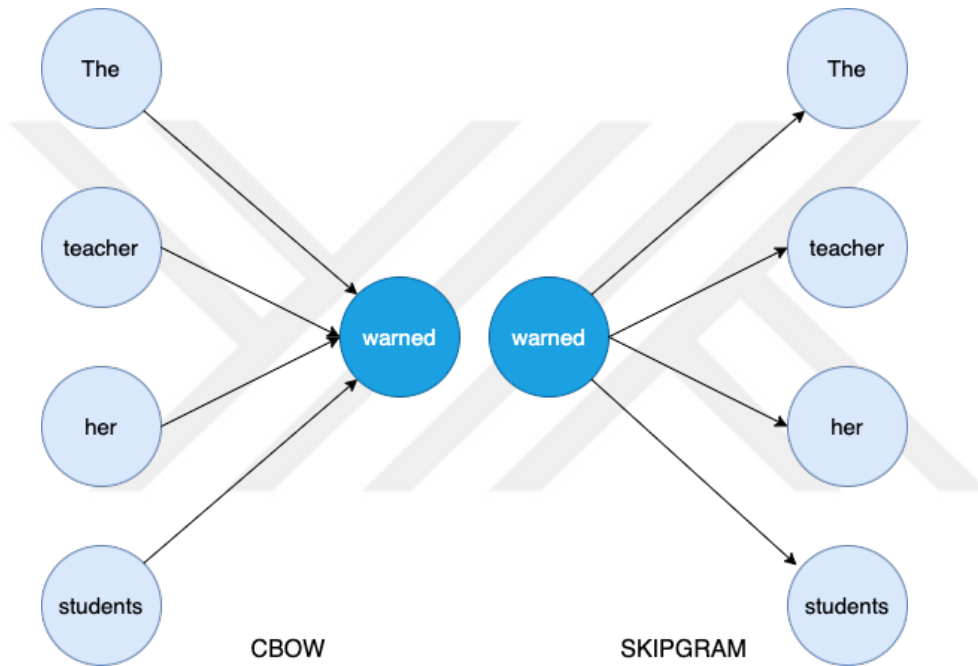


Figure 2.2: CBOW & SkipGrams in Word2Vec

One of the main contributions of Word2Vec is the simplicity of reducing or increasing the dimension of feature vectors. Since the primary purpose of feature selection is to collect the most practical features rather than redundant or irrelevant features, Word2Vec offers some parameters to select feature dimensions in advance and be adjusted accordingly. Although considering its architecture, researchers argue about whether Word2Vec is genuinely deep learning or not (Rong, 2014), it is still one of the most frequently used word embedding techniques in the literature. To illustrate the usage of word2vec with one of the smallest samples of assembly code in our study, such as, `"['push ebp', 'mov ebp,esp', 'sub esp,addr']"`, we used the following parameters. Two for the window size, so that each word will be related to one word before and one word after. Because our sample is too small, we didn't want to downsample this eight-word set, so we provided one for the minimum count. With a total of six unique words, we created the embeddings. Then, we calculated the most seman-

tically similar word to *mov* by using *most\_similar* function provided in *wordvector* model, `w2v_model.wv.most_similar(mov, topn=1)`, we understand that *push* opcode is the most similar with a distance of 0.990742564201355 to *mov* opcode. We created the samples in this subsection using the code<sup>7</sup> in github repository. Recently, strong alternatives to word2vec and similar methods have started to emerge. One of them is transformers. It is claimed to be superior to other vectors, mainly in short sentences. In essence, transformers work with the attention mechanism and treat all inputs at the byte level. It works by marking the beginning of the sentence, the beginning of the word, the end of the sentence with its own token logic. Although in this study we treated assembly instructions as sentences, after creating the word vectors with textvectorization and word2vec, we examined the model success by using transformers, in particular DistilBERT (Sanh et al., 2019) in our sentence level analysis.

### 2.1.7 Document Vectorization

The approaches employed for transforming words or sentences are also used on entire documents to extract features and represent them as n-dimensional vectors. Those vectors may be used to detect similar documents, distinguish the topics of the documents and context. One of the methods applied to documents for vectorization purposes is the n-gram model. n-grams are contiguous sequences of predefined (n) items from a given series of text. We mentioned 2-grams on assembly instruction in the Count Vectorization subsection. When applied to an entire document or a corpus rather than to the sentences as in our example, each tuple of 'n' grams, either characters or words, is represented by a unique bit in a bit vector. When aggregated for a body of text, those bits form a sparse vectorized representation of the text in n-gram occurrences. The TF-IDF, which is also used word-level, creates vector representation of the document to calculate document similarity. On the other hand, Doc2Vec (Le & Mikolov, 2014), an extension of the Word2vec, learns the representation of documents through two models, namely Distributed Bag of Words (DBOW), which is equivalent to Skip-Gram in Word2vec, and Distributed Memory (DM), which is equivalent to CBOW. For vector representation of a document, Distributed Bag of Words (DBOW) method needs an identifier or label, such as a topic. Then, DBOW randomly predicts a probability distribution of words in a document resulting in an n-dimensional vector using the document's identifier. The order of words is lost in DBOW. During training, the document vector and word weights are randomly initialized and updated using stochastic gradient descent (SGD). The second model, namely, Paragraph Vector—Distributed Memory (PV-DM), unlike DBOW, predicts a word from the context of the document. It takes a set of words of a paragraph randomly and a document identifier as input and tries to predict a central word.

### 2.1.8 Sentiment Analysis and Text Classification

Language: with its structure, semantics, and phonetics is a widely studied structured system of communication using symbols. With those symbols, human beings express

---

<sup>7</sup> word2vec\_sample.py: [https://github.com/d\protect\discretionary{\char\hyphenchar\font}{}{}demirci/binary\\_classification/blob/master/word2vec\\_sample.py](https://github.com/d\protect\discretionary{\char\hyphenchar\font}{}{}demirci/binary_classification/blob/master/word2vec_sample.py)

themselves. Linguistics, a scientific study, mainly focuses on structure, semantics, morphology, and phonetics of grammar in human languages. Computational linguistics, a subcategory of linguistics, deals with human languages using a computational perspective to formulate those grammatical aspects. Additionally, it is concerned with the computational modeling of natural language. On the other hand, NLP focuses on understanding natural languages utilizing computers. Understanding is to accurately extract information and insights contained in the documents and categorize or organize the documents. Although recently NLP and Computational Linguistics are used interchangeably, Natural language processing may be summarized as the set of methods for making human languages accessible to computers.(Eisenstein, 2019) . The development of fast computers and the emergence of big data made it possible to process previously unhandled documents. Also, the processing power of today’s computers made it possible to expose unknown or hidden information from documents. Such as hidden meanings and intentions from those documents. One of the problem domains in natural language processing is sentiment analysis. Sentiment Analysis (SA), Opinion Mining(OM) or Polarity Detection aims to distinguish positive or negative sentiment in each text; hence may be considered as a classification process. It is a computational study of people’s opinions, attitudes, and emotions toward events and even towards individuals. There are three classification levels in SA: document-level, sentence-level, and aspect-level SA. Document-level SA is to categorize a document as articulating a positive or negative opinion or emotion. It studies the whole composition as a basic block of information. Sentence-level SA aims to classify sentiment expressed in each sentence. Since each sentence may be considered as a short document, the categorization of text at document-level or sentence-level may not differ in the matter of preprocessing. Aspect-level SA, known as word level, is the most concise classification since both positive and negative sentiments may be expressed for the same entity. Various types of SA models focus on different aspects of an entity, such as polarity, feelings, emotions, and intentions. The polarity-focused SA includes categories like very optimistic, positive, neutral, negative, and very damaging. Some studies refer to this kind of classification as fine-grained sentiment analysis (Zirn et al., 2011), such as 5-star rating reviews (Sharma et al., 2015). Detecting emotions like happiness, frustration, anger, sadness is another focus of SA. These SA models take advantage of lexicons or complex machine learning algorithms. Since natural language is flexible and people express emotions in various ways, the downside of using lexicons<sup>8</sup> originates from using the same word to demonstrate both anger and happiness. In the present study we focused on polarity detection in sentence-level and document-level assembly instructions to classify PE files as malicious or benign.

### 2.1.9 RNN, LSTM vs BiLSTM

The Recurrent Neural Networks (RNNs), based on (Rumelhart et al., 1986), is an extended version of the conventional Feed-Forward neural networks to work with variable-length sequence inputs. The extension lies in new gates added to store the previous inputs and leverage sequential information from the RNNs model’s previous inputs. Those gates increase RNNs memory and give the RNNs the ability to predict what input to expect in the input data sequence (Figure 2.3). Although RNNs seem

---

<sup>8</sup> lexicon: <https://en.wikipedia.org/wiki/Lexicon> (retrieved on 20 Jul 2021)

to leverage preceding sequential information for long series, due to RNNs' memory limitations, the size of the sequential information is reduced to a few stages back. This drawback is known as "vanishing gradients". This problem makes it difficult for RNNs to capture the long-term dependencies, and as such, the training of RNNs will be highly challenging. Another problem that makes RNNs hard to train is known as "exploding gradients,"

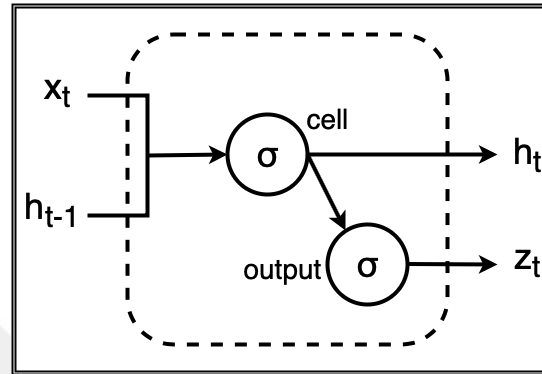


Figure 2.3: RNN Basic Architecture, redrawn based on (Donahue et al., 2014)

The LSTM (Hochreiter & Schmidhuber, 1997) model, (Figure 2.4), extends the RNNs model by increasing RNNs' memory to store and learn long-term dependencies of inputs to address the vanishing gradient problem (Sherstinsky, 2018b). This memory extension, "gated" cell, gives RNNs Models the ability to recall information and decide whether to store or discard the information. During the training process, the weight values assigned to the information affect this decision. Hence LSTM models store data for a longer time than RNNs Models. The weights make an LSTM model learn what information deserves to be preserved or removed.

The bidirectional LSTMs (Figure 2.5) are an extension of the described LSTM models in which two LSTMs are applied to the input data. In the first phase, the forward layer, the input sequence is fed into the LSTM model. In the second round, the backward layer, an LSTM model is applied to the reverse form of the input sequence. This approach improves learning long-term dependencies, makes the model know what words immediately follow and precede a word in a sentence, and thus enhances the model's accuracy.

RNN, LSTM, and BiLSTM Networks are used for learning and analyzing the patterns across time in a long sequence of data. These networks infer short- and long-term dependencies or time-based variances.

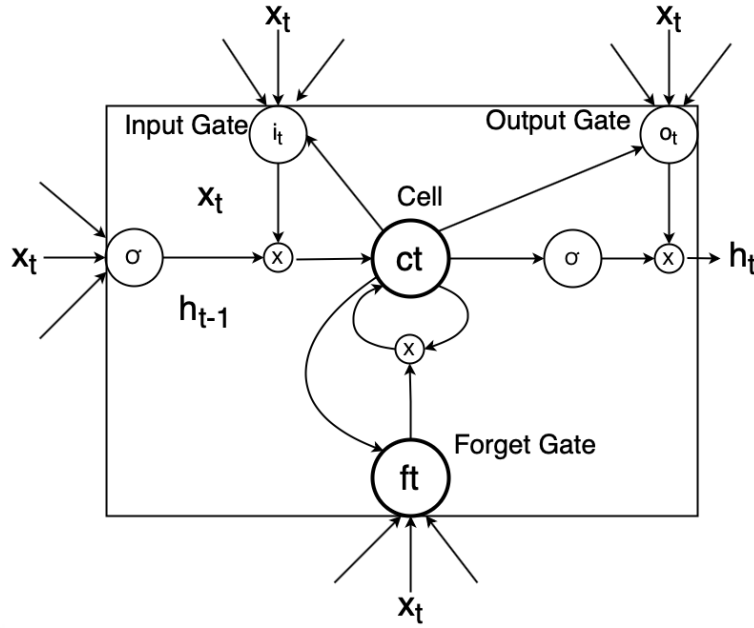


Figure 2.4: LSTM Basic Architecture, redrawn based on (Graves & Jaitly, 2014).

## 2.2 Malware Detection Methods

Malware detection methods rely on analysis techniques applied to malicious PE executables. Those techniques may be classified as static analysis and dynamic analysis techniques. Static analysis is a type of analysis that is performed using the static information acquired from the target executable. Such as headers, imported libraries, strings, hash values. Dynamic analysis, on the other hand, is based on monitoring the malware in question by running it in a controlled environment. Analysts employ those techniques to extract features from executable to identify it.

### 2.2.1 Signature-Based Detection Methods

Signature-based detection methods rely on storing previously generated signatures of known malware samples and matching these stored signatures with a signature generated from a newly encountered executable. A signature consists of a sequence of bytes that are uniquely present in the malware. Therefore, signatures are malware-specific and detect only known malware samples. On the other hand, some signatures are created using the specific content to families or variants of malware. Such signatures are called generic signatures. Since signature extraction is done chiefly manually, the experts analyze the malware and determine the signature; it is naturally time-consuming. However, some automatic signature extraction systems have been developed successfully to speed up the signature extraction process (Griffin et al., 2009). Due to its simplicity, signature-based detection is a widely used method, not only on the end-user side but also in perimeter defense. While antiviruses, anti-malwares, and host-based intrusion detection systems use signatures in the end-user

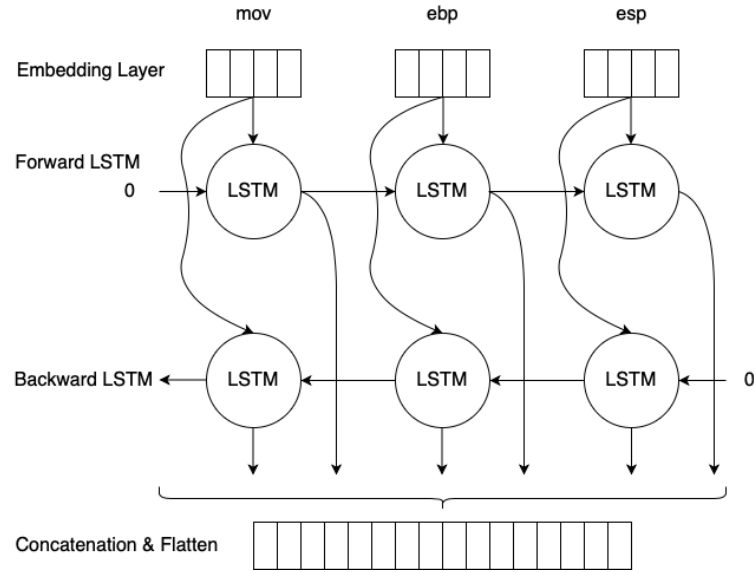


Figure 2.5: BiLSTM Basic Architecture, redrawn based on (Cornegruta et al., 2016).

domain, network-based intrusion detection systems (NIDS), next-generation firewalls (Next-GEN FW) with IDS capability use signatures in the network domain (Khraisat et al., 2019; Sourì & Hosseini, 2018). The significant disadvantages of signature-based malware detection methods may be listed as the inability to recognize newly produced malware, the troublesome signature generation and distribution processes, and the unmanageable growth of signature databases.

### 2.2.2 Behavior-Based Detection Methods

Behavior-based detection methods focus on the activities of malware on an infected system. These activities may be system calls, file activities, registry activities, API calls, communications between the infected system and a remote server, or a decryption loop in an assembly code. To identify the activities, behavior-based methods use both dynamic and static analysis of malware. Statically performed behavior-based detection involves using a template to identify behavioral traits such as detecting a decryption loop in polymorphic malware, or a loop to search for email addresses in directories without running malware (Christodorescu et al., 2005). On the other hand, in the dynamic behavior-based detection method, the malware is executed in a controlled environment, and its behavior is monitored. For example, system calls constitute malware behavior monitored dynamically and subsequently used for malware detection (Lin et al., 2014). The execution of malware and monitoring those activities contributes to malware detection rate of behavior-based mechanisms. On the other hand, when malware runs in virtual machines or sandboxes, it is possible that malware is environment-sensible, so detects the environment and may evade the sandboxes. Additionally it takes time to run, and the needed resources may be intensive. Moreover, false positives are often a concern due to the misclassification of benign software because benign executables may exhibit behavior like malware. (Mosli et al., 2017)

### 2.2.3 Machine Learning and Deep Learning Based Detection Methods

The increase in the number of novel malware variants requires automation of malware detection as they may no longer be detected by resources primarily depending on analysts. Therefore, there is a need for automated detection methods for malware with little or no human intervention. For this reason, the recent studies in malware detection have evolved from signature-based and behavior-based detection methods to machine learning-based detection methods. In machine learning-based methods, there are two stages, such as feature extraction and identification. In the feature extraction phase, various data collected from malicious files are used. These data types include opcode sequences, API Calls, system Calls, Control Flow Graphs (CFGs) (Gibert et al., 2020). Malware detection methods based on machine learning require human control over feature extraction and feature selection. Thus, in those approaches, supervised learning is employed at the feature selection phase. The supervised learning models may require certain levels of expertise to extract and select features accurately. So, the algorithms that eliminate the need for knowledge during feature selection, feature engineering, dimension reduction, or feature reduction (Yan et al., 2013) processes could also be used before supervised learning algorithms. Those algorithms aim to make a better representation of the extracted features. Some examples of unsupervised feature selection methods used are Laplacian Score (X. He et al., 2005), Nonnegative Discriminative Feature Selection (NDFS)(Li et al., 2012), and clustering-guided sparse structural learning (CGSSL)(Li et al., 2014). In the second stage, the collected features are fed into a supervised learning or classification algorithm. The same classification algorithm may be used to detect malicious files as in binary classification problems or to distinguish malware families. Some examples of such machine learning classification algorithms used to detect malware in academic studies primarily include Logistic Regression, Naive Bayes Classifier, Support Vector Machine, Decision Trees, Boosted Trees, and Random Forest (Gibert et al., 2020). The clustering algorithms are related to the natural grouping of data and found labels associated with each group. In the malware detection studies focusing on machine learning clustering, the k-means clustering algorithm is the most used machine learning clustering algorithm. Since training supervised learning models requires too much time, effort, and domain expertise in extracting and selecting essential features (Rathore et al., 2019) and also suffer from human errors; Deep Learning-based techniques are starting to attract a lot of attention to malware detection. Deep learning techniques, in other words, neural networks, come in because they may learn from data without feature extraction by humans. Malware detection studies using deep learning techniques have various approaches. Those approaches may be summarized as the training of recurrent neural networks (RNN) and convolutional neural networks (CNN) and, lastly, attention mechanisms (Vaswani et al., 2017). CNN's are mainly used for image recognition tasks, in malware detection domain, it employs feature extraction by converting malware into images. However, 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. Attention mechanisms allow the modeling of dependencies without regard to their distance in the input or output sequences. As in machine learning studies, the data collected from malicious files, including API calls, system calls, opcodes, opcode graphs, and many other features, are used to train a neural network detecting malware. Hence, the effectiveness of these methods highly depend on the features extracted from the sam-

ples in datasets. The studies employed deep learning techniques, which generally are close to becoming automatic with reducing human intervention, show better results in some cases, showing similar performances with machine learning classification techniques in some other. We are exploring those studies in the upcoming section.

### **2.3 Learning Methods in Malware Detection**

Any portable executable has common aspects, such as included libraries, strings, and code patterns. Thus, researchers generally use the extracted features using those aspects in malware analysis and malware detection, whether the method is signature-based, behavior-based, machine learning-based, and deep learning-based. However, determining those identifying features is not always an easy task because malware developers constantly update their methodology and create new types of malwares. Therefore, artificial intelligence and machine learning integration aim to develop end-to-end malware detection systems with high accuracy, low false-positive rates, less human intervention, and best performance. One of the successful sub-fields of this integration is deep learning. However, as in machine learning techniques, the effectiveness of the deep learning methods is affected by the input features extracted from the dataset. The features generally consist of API calls, opcodes, opcode graphs, and many other aspects in executable files. We also examine assembly instructions obtained from disassembled PE (Portable Executable) files.

ML classification algorithms require a methodological approach that contains mainly two steps which are feature related and classification related. Feature related step consists of feature extraction, feature selection and reduction processes. Classification related step includes deciding the best algorithm that may classify or detect the family of executable. The researchers in (El Merabet & Hajraoui, 2019) compared the steps required for machine learning-based malware detection, including feature extraction, selection, and reduction. By applying different feature extraction techniques, they observed the effects of signatures, dll functions, binary string, and pe headers. Then they compared principal component analysis(PCA) and random forest algorithms for feature reduction. Finally, they discussed three algorithms, support vector machine(SVM), random forest, and artificial neural network, and compared feature selection and accuracy based on those algorithms. Feature selection methods may depend on the researchers approach to the problem. For example; Bilar (Bilar, 2007) stated that the difference between malware files and benign files was statistically significant in opcode frequency distributions. Furthermore, they used rare opcodes as a predictor for malware detection. Santos et al. (Santos et al., 2010) studied the incidence of opcode sequences. They investigated the relationships among the opcodes and used statistical information to detect variants of known malware families. Later, the method in the study (Santos et al., 2011) was based on analyzing the appearance of frequency of opcode sequences to create semi-supervised machine learning classifier using a set of labeled and unlabeled data to detect novel malware. On the other hand, in (Anderson & Roth, 2018), Anderson et al., used an extensive dataset contained nine hundred thousand malicious data. They divided the dataset into training set, validation set, and testing set, and each includes three thousand malicious data which later they open sourced, known as the EMBER dataset. They used a cross

platform library (LIEF<sup>9</sup>) to parse malicious and benign executables in their dataset to extract features. The features consisted of eight groups of information. Five of them were the output of LIEF such as general file information, header information, section information, imported and exported function in json format. The other three features obtained from raw file were byte histogram, byte-entropy histogram, and string information. They employed LightGBM<sup>10</sup> on the obtained features with a detection rate of 98.2%.

Although the study (Raff et al., 2018) claimed that byte n-grams technique leads to overfitting and overestimating the accuracy in the malware detection domain, undeniably, byte n-gramming is a technique commonly used in malware classifiers since it expects none or less domain knowledge. For example, (Moskovitch et al., 2008) used n-gram of the opcodes as a feature vector for the classification process. In their study, they detect unknown malware based on the text categorization they applied. Their methodology was successful than byte sequence n-gram representation, and their results indicated 99% accuracy using a training set with a malicious file percentage lower than 15%. Besides n-gram, some studies, (Shabtai et al., 2012), applied TF and TF-IDF representations for each opcode n-grams with an accuracy rate of 95.6% to detect malware.

However, in the NLP domain, although it was shown that the TF-IDF is a richer and more successful representation for the retrieval and categorization purposes (Salton et al., 1975) in malware domain the results varied. As in (Moskovitch et al., 2008) and (Shabtai et al., 2012), they found out that the scarce vector representation of opcodes created by TF-IDF led to poor results concerning accuracy compared to n-grams. The usage of n-gram is not limited to detection of malware, it is also used to classify malware. In (Zhang et al., 2019), the researchers used five different machine learning classification algorithms to detect and classify ransomware families using extracted features from n-gram opcode sequences. Moreover, the highest accuracy rate in this study was 91.43%.

Since machine learning depends on the extracted features selected by analysts or experts to identify malware, that may cause losses of robustness and miss out on malwares never seen before. On the other hand, the DL approaches have the advantage of learning patterns in data by themselves. Thus, it provides adapting to the changes and makes them robust and easy to maintain. Deep learning methods extract valuable information about semantic source code, binary code, opcodes, and assembly code may be used as features, mapped with different word embedding techniques to word vectors. For instance, the study, (Krcál et al., 2018) treated an executable as a sequence of bytes and applied CNN for malware detection. Their CNN model contained four convolutional layers and four fully connected layers. Instead of a global max-pooling layer, they used a global mean pooling layer after the convolutional layers. And their best afford was 97.1% accuracy. There are several recent studies focus on assembly code and detect malicious software employing deep learning techniques. In (Khan et al., 2019), investigated GoogleNet (Szegedy et al., 2015) and five different ResNet(K. He et al., 2015) models using images produced from opcodes of binary files. They applied Histogram standardization enlargement and disintegration techniques to augment images. With this technique it became easier to distinguish between malicious

---

<sup>9</sup> LIEF: <https://lief.quarkslab.com> (retrieved on :17 Feb 2021)

<sup>10</sup> LightGBM: <https://github.com/microsoft/LightGBM> (retrieved on : 17 Feb 2020)

and benign opcode using images. The accuracy rate of GoogleNet was 74.5%, and the best accuracy rate among ResNet models was 88.36%. In (Kumar et al., 2018), also employed CNN to classify malware opcode images. The accuracy rate of correctly classified binary files was 98%.

Instruction2Vec (Lee et al., 2019) work uses both opcode and operand information to classify malwares. They used a nine-dimensional feature vector to resemble registers, and addresses. They split instructions and encoded each token as unique index numbers. In their setup, an opcode takes one token, a memory operand takes up to four tokens, including base register, index register, scale, and displacement. This approach represents information about opcode and operands. The researchers in (Lu, 2019) also used opcodes and operands as features. They extracted opcode and operand from their dataset which consists of 969 malware samples and 123 benign samples. Later, they applied different word embedding techniques to word vectors, and used resulted vectors to feed into their models based on LSTM. Their model attained accuracy of 97.87% for malware detection, while for classification, the model achieved an accuracy of 94.51%.

## 2.4 Summary

We presented background information about malware analysis, neural networks, natural language processing, and the relevant studies conducted by researchers on malware detection. We aim to detect malware using NLP techniques, so we put forward certain aspects of NLP. Furthermore, we explained the various methods to convert human-readable strings to machine-ready formats. We also mentioned how NLP is used to identify topics, subjects, and polarity of opinions with sentiment analysis methods. In natural language processing, the major challenge is understanding the patterns in the data sequence. Then using this pattern, we analyze the text to predict the next character, word, sentence or the main idea of a corpus. We explained that RNN, LSTM, and BiLSTM are well suited for forecasting the future using current and previous data. Therefore, those models have been widely preferred for sequential structures like time series and natural languages. We shared the applicable works associated with malware detection, involving earlier statistical approaches and machine learning-based and deep learning-based methods. In the scope of this study, we investigated NLP techniques and procedures for applying to assembly language. Purposely, we collected Windows executable files and obtained assembly instructions to form our dataset. Then, we performed data preprocessing methods and text-preprocessing techniques to create a machine-ready representation of assembly instructions. As the second part, we built the detection model consisting of different layers, including BiLSTM. In the next section, we present the particulars of our methodology.



## CHAPTER 3

### METHODOLOGY

In this chapter, first, we present our approach to malware detection. Then we describe our dataset and the process that we acquire assembly instructions from our dataset and the formats that we prepare our dataset to feed our architecture. Finally, we present the design that we use to construct the language model.

#### 3.1 The Approach

Methods for detecting malware are usually classified into three basic approaches: static, dynamic, and hybrid. While dynamic analysis and hybrid analysis depends on running the executable file in a controlled environment or in a sandbox to inspect and identify the behavior, static analysis deals with the statically obtained information and features from the executable files. Therefore, static analysis methods do not need the execution of binary. While conducting malware detection with static analysis, a fundamental approach will be to utilize the sections, imported DLLs, exported functions, resources, URLs, compilers, packers, and human-readable strings. The traditional method to detect malicious files depends on creating signatures that may be used in security systems. These signatures identify a single file or a family of malicious executables. New signatures should be created and delivered using the necessary means for each new malicious file. Although automated signature builders exist, there is still a need for human involvement in creating signatures (Griffin et al., 2009). Thus, it cannot detect zero-day attacks since there is no corresponding signature stored in the repository. Additionally, repositories of signatures cannot keep up with the evolution rate of new malware, or the databases swell quickly. One of the cornerstones of cutting-edge deep networks is the approach of end-to-end learning, or equivalently, automatic characteristic extraction where only the labels and uncooked statistics are introduced to the network with no handmade elements provided and close to no pre-processing. Our study extracts assembly code using an open-source disassembler to create an opcode sequence as output. We used the output as our raw data to create a language model assisted with word embedding, just like processing natural language. Using this language model, we aim to adopt polarity detection methods to identify the intention of an executable file using the labels as malicious and benign. Hence, we plan to detect whether it is malicious or benign with our proposed language model.

## 3.2 Data Collection and Formatting

Our dataset consists of benign and malicious executables in Portable Executable(PE)<sup>1</sup> format. The PE is a complicated structure, based on COFF (Common Object File Format)<sup>2</sup> specification, with its standard headers, optional headers, sections of various types, resources, and relocation tables. Typical COFF sections encompass code or data that linkers and Microsoft Win32 loaders do not need information about the area contents. The contents apply entirely to the software that is being linked or executed. However, some COFF sections have special meanings when found in executable files. Due to the special flags set in the section header, these sections are easily recognized by tools and loaders. The table A.1 in Appendix A includes the descriptions for the section types that are stored within executables and the section types that incorporate metadata for extensions.

In the present study, we use the *.text* section (referred to as code section throughout our study) contents of executables found in our collection. Our collection contains Win32 PE files from Windows operating systems<sup>3</sup>, and Commando VM v-2.0<sup>4</sup>, and malicious x86 executables from the sorel-20m (Harang & Rudd, 2020) database website. Since we aim at detecting malware, we randomly chose the malicious samples, including various types of malwares, such as viruses, worms, and trojan. We chose Commando VM over the rest of the versions of Windows OS, because it contains executables compiled using different compilers, such as Cygwin<sup>5</sup> and MinGW<sup>6</sup>.

We focused on assembly instructions obtained statically from the collected executables. The first step of our proposed methodology is to disassemble each benign/malicious file to get assembly instructions contained in the code section. Next, the outputs are saved in plain text files, namely "Document Level Analysis Model" (viz. DLAM). Then, the first dataset is processed and a second dataset with one instruction per document is obtained. We call this model "Sentence Level Analysis Model" (viz. SLAM). Finally, we feed our bidirectional LSTM language modeling architecture with our datasets. The overall processing pipeline is presented in Figure 3.1.

### 3.2.1 Obtaining Assembly Instructions

We first converted the executables into assembly language instructions using a slightly modified version of *bin2op.py*<sup>7</sup>. The code<sup>8</sup> used in this process is running `objdump` on binary and extracting assembly instructions in the code section from the output. In our version, we obtain instructions as a list. We treat each instruction as a sentence,

---

<sup>1</sup> PE Format: <https://docs.microsoft.com/en-us/windows/win32/debug/pe-format> (retrieved on 21 Feb 2020)

<sup>2</sup> COFF: <https://docs.microsoft.com/en-us/windows/win32/debug/pe-format> (retrieved on 21 Feb 2020)

<sup>3</sup> Microsoft Windows 8.1 Pro (OS Build 9600), Microsoft Windows 10 Pro 19.09 (OS Build 18363.418),

<sup>4</sup> Commando VM: <https://protect\leavevmode@ifvmode\kern+.2222em\relax//github.com/fireeye/commando-vm> (retrieved on 19 Mar 2019)

<sup>5</sup> Cygwin: <https://www.cygwin.com> (retrieved on 19 Mar 2019)

<sup>6</sup> MinGW: <http://mingw-w64.org> (retrieved on 19 Mar 2019)

<sup>7</sup> *bin2op.py*: [https://github.com/d-demirci/binary\\_classification/blob/master/bin2op.py](https://github.com/d-demirci/binary_classification/blob/master/bin2op.py) (retrieved on 17 Apr 2019)

<sup>8</sup> *create\_dlam\_dataset.py*: [https://github.com/d-demirci/binary\\_classification/blob/master/create\\_dlam\\_dataset.py](https://github.com/d-demirci/binary_classification/blob/master/create_dlam_dataset.py)

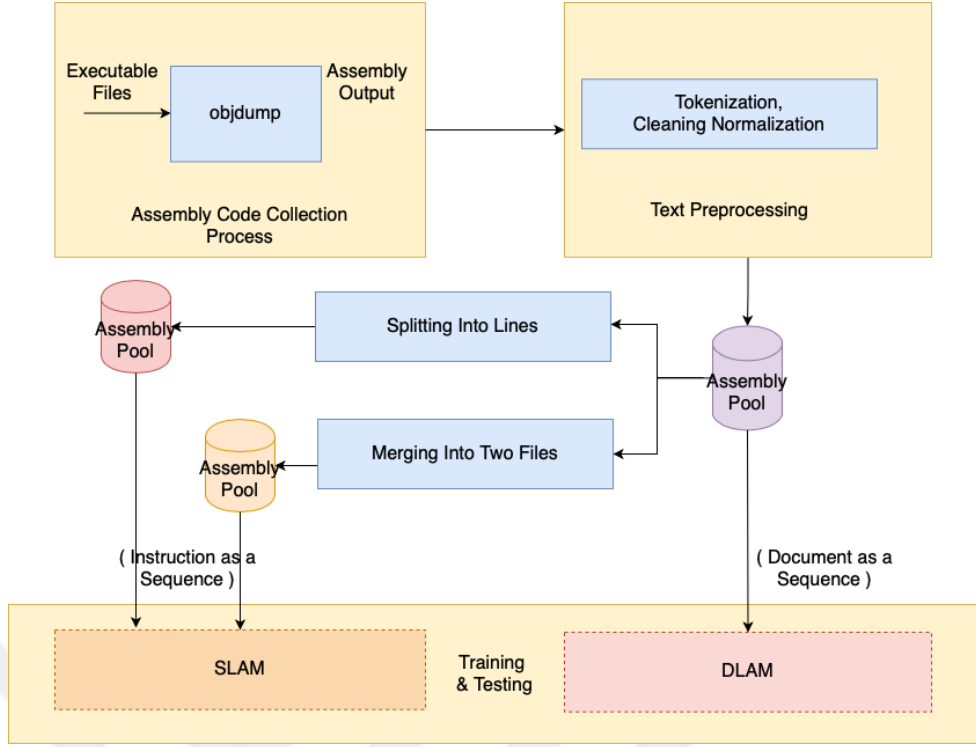


Figure 3.1: Language Modeling Pipeline.

each opcode or operand as a word, and the complete code section as a document. We present a view of samples from our dataset for the DLAM model in Figure 3.2.

On the other hand for creating *word2vec* and *DistilBERT* vector representations for the SLAM, we processed<sup>9</sup> the files in the DLAM dataset for the second time. We merged the files in *train/malw* folder in *train/malicious.txt* and the files in *train/b-ngn* folder in *train/benign.txt*. Using these files we created *pandas.DataFrame* with columns *content* and *category* as seen in Table 3.1.

### 3.2.2 Standardization and Tokenization

To prepare the dataset for feeding our architecture, we standardize, tokenize, and vectorize the data using *TextVectorization* layer in the Keras. As we mentioned in the background section, Standardization refers to preprocessing the assembly, typically to remove punctuation to simplify the dataset. Tokenization means splitting strings into tokens, in our case, splitting an assembly instruction using whitespace and commas into individual words. Vectorization means converting tokens into numbers so they can be used to train a neural network. Since we extracted assembly code from executable files, our dataset consists of subsequent assembly instructions. We wrote a

<sup>9</sup> `merge_files.py`: [https://github.com/d-demirci/binary\\_classification/blob/master/merge\\_files.py](https://github.com/d-demirci/binary_classification/blob/master/merge_files.py)

```

0x2374.and bh, BYTE PTR [esi].or eax, 0x2020200a.and BYTE PTR [eax], ah.and
BYTE PTR [eax], ah.and BYTE PTR [edx+esi*2], bh.gs jno 0x23c7.gs jae 0x23c9.gs fs
inc ebp.js 0x23bf.arpl WORD PTR [ebp+0x74], si.imul ebp, DWORD PTR
[edi+0x6e], 0x6576654c.ins BYTE PTR es:[edi], dx.and BYTE PTR
[ebp+eiz*2+0x76], ch.gs ins BYTE PTR es:[edi], dx.cmp eax, 0x49736122.outs
dx, BYTE PTR ds:[esi].jbe 0x23e2.imul esp, DWORD PTR [ebp+0x72], 0x22.and BYTE
PTR [ebp+0x69], dh.inc ecx.arpl WORD PTR [ebx+0x65], sp.jae 0x23f3.cmp
eax, 0x6c616622.jae 0x23ec.and ch, BYTE PTR [edi].ds or eax, 0x2020200a.and
BYTE PTR [eax], ah.and BYTE PTR [edi+ebp*1], bh.jb 0x23fb.jno 0x240d.gs jae
0x240f.gs fs push eax.jb 0x2409.jbe 0x240b.ins BYTE PTR es:[edi], dx.gs addr16 gs
jae 0x23e6.or eax, 0x2020200a.and BYTE PTR [edi+ebp*1], bh.jae 0x2417.arpl
WORD PTR [ebp+0x72], si.imul esi, DWORD PTR [ecx+edi*2+0x3e], 0x20200a0d.cmp
al, 0x2f.je 0x2433.jne 0x2436.je 0x240e.outs dx, BYTE PTR ds:[esi].outs dx, WORD
PTR ds:[esi].ds or eax, 0x612f3c0a.jae 0x2443.gs ins DWORD PTR es:[edi], dx.bound
ebp, QWORD PTR [ecx+edi*2+0x3e].add BYTE PTR [eax], al.add BYTE PTR
[eax], dh.add BYTE PTR [eax], al.or al, 0x0.add BYTE PTR [eax], al.pop esp.cmp
al, BYTE PTR [eax]..

```

Figure 3.2: Sample Document from DLAM Dataset.

custom standardization<sup>10</sup> function to clean our data. For this process, first, we cleaned some irrelevant opcodes from sequences such as *align*, *bad*, *int3*. Then we replaced some variants of opcodes with basic opcodes, such as replacing *movb*, *movw*, *movl*, *movq* with *mov* instructions. This process is like replacing synonyms of a word in a corpus with a word to augment data in natural language processing. But in our case, to keep the relation between opcode and operand, we did the opposite. Also, we replaced the hex representation of addresses with string *addr*. We preprocessed the assembly instructions in both datasets, the overall process applied in DLAM may be depicted as in Figure 3.3.

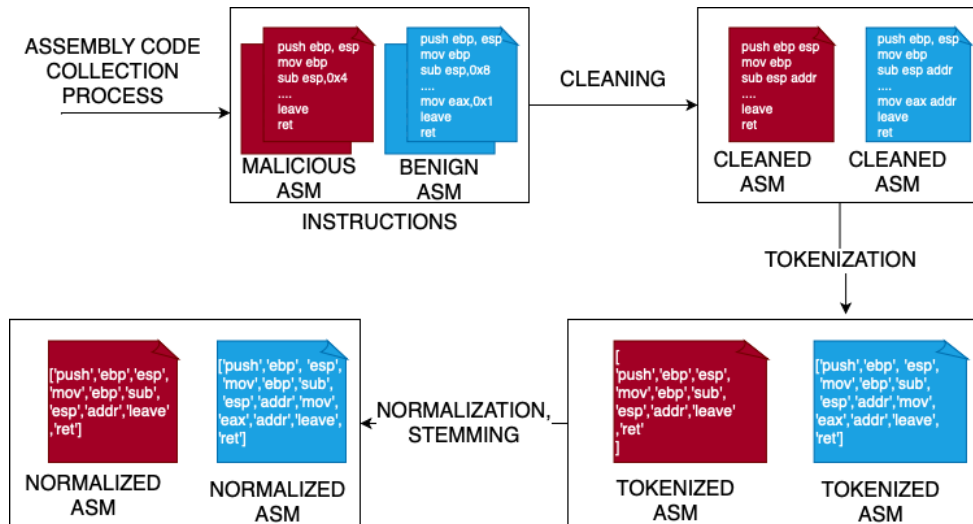


Figure 3.3: Text Processing for the DLAM.

<sup>10</sup> custom\_standardization: [https://github.com/d-demirci/binary\\_classification/blob/master/custom\\_standardization.py](https://github.com/d-demirci/binary_classification/blob/master/custom_standardization.py)

Table 3.1: Sample from the SLAM Dataset

| id      | content                          | category  |
|---------|----------------------------------|-----------|
| 0       | dec ebp                          | malicious |
| 1       | pop edx                          | malicious |
| 2       | nop                              | malicious |
| 3       | add BYTE PTR [ebx],al            | malicious |
| 4       | add BYTE PTR [eax],al            | malicious |
| ...     | ...                              | ...       |
| 7432620 | push es                          | benign    |
| 7432621 | cmp BYTE PTR [eax+0xff876ae],dh  | benign    |
| 7432622 | (bad)                            | benign    |
| 7432623 | and BYTE PTR [edi+0x6d9207a0],al | benign    |
| 7432624 | mov edi,0xaa2da6                 | benign    |

Also for single line instructions, the text-processing is applied as shown in Figure 3.4.

To prevent train/serving skew, we included the TextVectorization layer directly inside our model; hence we preprocess the data using the same methods at train and test time. We give in the upcoming chapters. We used the default split function, which splits the sentence using the whitespace character, and the custom standardization function we defined above. We determined the variables for the model, like an explicit maximum sequence\_length, which will cause our TextVectorization layer to pad or truncate sequences to exactly sequence\_length values. We tried with different sequence length in training and testing phase for creating vector representation of our assembly code. In the sample vector representation in the Appendix B.1 with a sequence\_length of 400 chosen randomly, we see that each token has been replaced by an integer as in Appendix B.1. In our dataset, on feature selection phase, we used the get\_vocabulary() function of TextVectorization layer, and we saw that a total of 1122 unique vocabulary, with the most frequent ones such as *addr*, *eax*, *dword*, *byte*, *mov*, *add*, *esi*, *al*, *edi*, *ecx*, *ebp*, *push*, *ebx*, *edx*, *ds*, *esp*, *pop*, *inc*, *es*, *dec*, and, *call*. Our dataset for the DLAM model consists of 810 malicious samples and 681 benign samples with a total of 1,491 files. On the other hand, the dataset we prepare for the SLAM model consists of 7,339,294 instructions. 4,036,612 from the malicious files and 3,302,682 from the benign files as shown in Table 3.2.

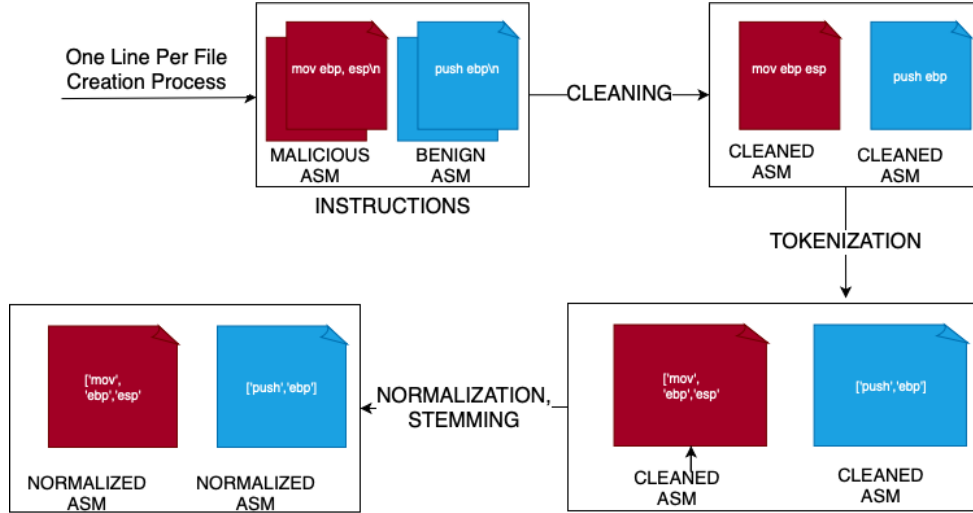


Figure 3.4: Text Processing for the SLAM.

Table 3.2: Total Number of Samples for models

| Dataset for | Malicious | Benign    | Total     |
|-------------|-----------|-----------|-----------|
| SLAM        | 4,036,612 | 3,302,682 | 7,339,294 |
| DLAM        | 810       | 681       | 1491      |

### 3.2.3 Splitting data into the training set, testing set, and validation set

To split the dataset into three, train, validation, and test is recently a best practice while constructing a machine learning model. Since we use *text\_dataset\_from\_directory* method from the Keras library, we arranged the directory as in Figure 3.5

To use in DLAM model, we arranged the directory structure for training and testing. The directory structure used for training and testing contains the class labels as directory names. There are two directories as malicious and benign for binary classification. This directory contains extracted assembly instructions from the executables. The number of samples in this directory used for training is 1,309. We used 719 samples from malware samples, and for benign samples, 590. Since we needed files that will never be used in the training phase to test our proposed model's accuracy, we put the test samples in the test directory, which contains two directories: malicious and benign. The number of total test files is 182. We subsampled the validation set from training samples, with a ratio of 80:20. With the help of *validation\_split* parameter of *keras.preprocessing.text\_dataset\_from\_directory*. Therefore we created validation data set with 261 files. To use in SLAM model, we used the documents in the DLAM model. We then split<sup>11</sup> those documents into sentences (assembly instructions) to

<sup>11</sup> prepare\_slam\_data.py: [https://github.com/d-demirci/binary\\_classification/blob/master/prepare\\_slam\\_data.py](https://github.com/d-demirci/binary_classification/blob/master/prepare_slam_data.py)

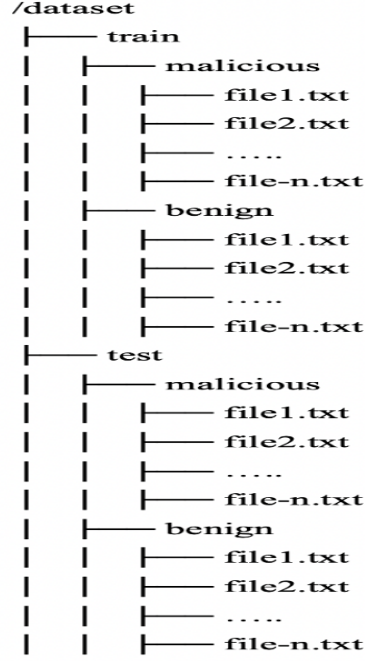


Figure 3.5: Directory Structure for Dataset

create the second dataset. As we did in the DLAM, we created two directories as malicious and benign, in both train and test directories. Those directories contain files that consist of single assembly instruction per file. The number of samples in the training directory was 6,442,918. We used 3,543,605 malware instruction samples and 2,899,913 benign instruction samples for training. While applying deep learning algorithms, the test dataset is never introduced to the model to simulate real-world scenario. Therefore, we put the test samples in the test directory, which contains two directories: malicious and benign. The number of total test files was 896,376. Since we already divided our dataset into training and testing set, we subsampled the validation set from the training set, using an 80:20 split ratio with the help of *validation\_split* parameter of *keras.preprocessing.text\_dataset\_from\_directory*. Hence, our initial model used 5,154,335 instructions for training, 1,288,583 instructions for validation, and 896,376 instructions for testing. We depict the sizes of training, validation, and test datasets for both SLAM and DLAM models as the following Table 3.3.

Table 3.3: Dataset Sizes for training, validation and testing

| Model | Training  | Validation | Testing |
|-------|-----------|------------|---------|
| SLAM  | 5,154,335 | 1,288,583  | 896,376 |
| DLAM  | 1129      | 261        | 182     |

### 3.3 The Proposed Model

This section introduces the technical details of our study, including the training and testing environment setup, used libraries and modules, and the training and testing pipeline elements. Before giving the details about the technical parts of our study, We briefly explain why we prefer to use BiLSTM over other neural network architectures such as RNN, LSTM. Among the neural network architectures, especially LSTM, are preferred for Natural Language Processing (NLP) tasks since they show better performance than other neural network architectures like RNN and CNN. RNN has a short memory to remember the previous situations, which causes performance problems 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. On the other hand, according to the study (Siami-Namini et al., 2019) the given input data is utilized in both backward and forward layers for training in BiLSTM models, and this causes a reduce in error rates by 37.78% compared to LSTM models.

#### 3.3.1 Setup Environment

We trained and tested our document level analysis model and the initial model of the sentence level analysis model on a machine with a 6-Core Intel Core i9 processor with 2.9 GHz speed and 32 GB Memory. For the training of Word2Vec and DistilBERT to create the sentence level analysis model, we used kaggle<sup>12</sup> draft session with a 13 GB ram and GPU with 16 GB Ram. Also, we implemented and ran our models in Python programming language with version 3.7.9. In the Python environment, we used the libraries with their specified versions in Table 3.4

#### 3.3.2 Imported Libraries and Modules

This section will give a brief explanation of the python libraries and modules used in our study's scope.

- from tensorflow.keras.layers import Embedding, LSTM, Bidirectional, Dense, Dropout, GlobalMaxPool1D  
Embedding, LSTM, Bidirectional, Dense, GlobalMaxPooling1D, and Dropout modules from the tensorflow.Keras library are used to add each of those layers into our layered architecture in the neural network.
- from tensorflow.keras.losses import BinaryCrossEntropy  
BinaryCrossEntropy is used to compute the cross-entropy loss between true labels and predicted labels.

---

<sup>12</sup> Kaggle: <https://kaggle.com>

Table 3.4: Required Python libraries

| Library                       | Version |
|-------------------------------|---------|
| TensorFlow                    | 2.4.0   |
| tensorflow-datasets           | 4.1.0   |
| tensorflow-estimator          | 2.4.0   |
| tensorflow-model-optimization | 0.5.0   |
| tensorflow-text               | 2.4.2   |
| scikit-learn                  | 0.24.0  |
| scipy                         | 1.5.4   |
| seaborn                       | 0.11.1  |
| numpy                         | 1.19.4  |
| Keras                         | 2.4.3   |
| Keras-Preprocessing           | 1.1.2   |
| hyperas                       | 0.4.1   |
| matplotlib                    | 3.3.3   |

- 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 confusion matrix that has graphical components.
- from tensorflow.keras import callbacks  
callbacks from TensorFlow.keras is used to prevent overfitting of our proposed model.
- from gensim.models import Word2Vec  
Word2Vec used to create vector representation and calculate weights for instructions in SLAM model, Doc2Vec used to create vector representation of documents in DLAM model.
- import re  
re used for regular expressions used for cleaning data, and removing irrelevant opcodes from dataset.

### 3.3.3 Training and Testing Pipeline

Our study designed a pipeline to take the dataset, preprocess the data for modeling, and train and test the neural network. To create the pipeline, first, we used the Dataset model found in TensorFlow to get batches of texts from the subdirectories malicious and benign, with labels 0 and 1 (0 corresponds to malicious and 1 corresponds to benign). This module requires a directory structure as in Figure 3.5.

Hence, as discussed earlier in the datasets section, we created directories to store malicious and benign assembly instructions. Using this module, we generate a dataset from directories with minimum memory usage. With the help of the *cache()* method provided by the *Dataset* class, we could keep data in memory after it's loaded off disk. That way, we make sure that our dataset does not become a bottleneck while training our model. Furthermore, we took advantage of this method to create a performant on-disk cache, which is more efficient to read than reading many files. As a result of the first step in the pipeline, we obtained text files, which contain assembly instructions extracted from executables. We applied text preprocessing techniques to assembly instructions in the dataset to create opcode sequences at the next step. We cleaned and tokenized our opcode sequences and applied TextVectorization class to create a vector representation of our input for this process. The sequences in the dataset are in different length so we specified a maximum sequence length. If a sequence is longer than the maximum sequence length, its first part is taken up until the maximum sequence length and the remaining part is discarded. If a sequence is smaller than the maximum sequence length, the padding operation is performed to complete the maximum sequence length. In the padding operation, the required amount of zero integer value is added to the end of the sequence. As a result, we obtain the vector representation of fixed-length sequences consisting of integer values. Since the dataset used for SLAM contained sentences with a maximum of 30 words (after tokenization), we saw that this method created sparse vectors filled with zeros if we increase the sequence length.

After we created the vector representation of our input data, we separated the dataset into three splits named training, validation, and testing for use during the training and testing. Then, we divided our dataset into two as 80% and 20%. We set 20% portions aside to use during the test phase. Then we divide the 80% portion into two again as 75% and 25%. We used the more extensive set for training and the latter for validation purposes. This way, we allot 60% of the dataset for training, 20% for testing, and 20% for validation. We built our initial classifier by stacking the layers sequentially as follows. Embedding layer is the first layer, which takes the integer-encoded opcode sequences and looks up an embedding vector for each word index. These vectors add a dimension to the output array, and during the training, the model learns the features represented by vectors. The resulting dimensions of the first layer are batch, sequence, and embedding. Since we want our model to learn features related to malicious and benign executables in the sequential form, we use BiLSTM as the next layer. After, BiLSTM layer we added Dropout Layer with a dropout rate for starting point 0.1. Next, a GlobalAveragePooling layer to return a fixed-length output vector for each example by averaging the sequence dimension. Then we feed, a fully connected (Dense) layer with 128 hidden units, with this fixed-length output vector. The last layer is a densely connected layer with a single output node. To calculate

weights during the training, every deep learning model needs a loss function and an optimizer. As we focus on classifying samples in two categories and the output of our model is a probability (a single-unit layer with a sigmoid activation), we used the BinaryCrossentropy loss function. Lastly, we configured the model to use an optimizer and a loss function. We preferred Adaptive Moment Estimation (Adam) (Kingma & Ba, 2017) optimizer. Since this model is our initial model with the help of hyperparameter tuning we give the details of shaping our final model with upcoming subsection, parameters and tuning.

### 3.3.4 Parameters and Tuning

Hyper-parameter tuning constitutes a crucial step in building neural networks. We chose the parameters in the present study in accordance with previous studies and popular trends in similar research. Additionally, we evaluated the performance of the different parameters on the effect of validation loss. Therefore, starting from one BiLSTM layer, we added one layer at a time and observed the performance of the model until it reached the optimum point. Other than the BiLSTM layer depth, there were other hyper-parameters such as batch size, number of epochs, filter size, optimization algorithm, dropout rate etc. A trial-and-error method may not seem a viable method for tuning the model because of the many combinations that occur using those parameters. To overcome this complexity, we employed *HParams*<sup>13</sup> library which works as a plugin of tensorboard. With the suggestions provided by HParams dashboard, automated hyperparameter optimization was possible. We manually experimented on some of the values HParams suggested with more epochs and data, to see how the models evolve and react to those parameters. we built the sentence level and document level neural network language models using the parameters that were optimum in terms of validation loss.

#### 3.3.4.1 Dropout and Regularization Methods

To make our training noisy, we trained four neural networks with dropout layers. We decided on the dropout and alternative regularization methods by examining the studies on this subject. The study in (Srivastava et al., 2014), states that dropout may break up the incorrect situations that the previous layers adapt by ignoring some of the outputs of preceding layer. On the other hand, according to the study (Yoder, 2018), there may be no optimal dropout rate parameter that may prevent overfitting neural network architectures. And it also stated that the dataset size may affect the dropout rate, the smaller datasets mean the lower the dropout rate, and vice versa. On the other hand, researchers suggested employing Variational RNN (Gal & Ghahramani, 2016) technique especially in LSTM models. Moreover, they showed the effects to the neural network model for sentiment analysis and text classification tasks as an alternative to dropout.

---

<sup>13</sup> HParams: <https://github.com/tensorflow/tensorboard/tree/master/tensorboard/plugins/hparams> (retrieved on: 17 May 2020)

### **3.3.4.2 Optimizers and loss functions**

Also, we trained on several known optimizers with their default configurations such as Adam, Adagrad (Duchi et al., 2011), RMSProp (Hinton & Swersky, 2012), and ADADELTA (Zeiler, 2012) while fixing other parameters. We found that Adam and RMSprop achieved similar results as it is also claimed in (Ruder, 2017). We preferred to use Adam since it showed slightly better performance than RMSprop in both models. By employing optimizers, we expected a decrease in the training loss with each epoch and an increase in training accuracy since, on every iteration, the optimization method is used to minimize the loss. However, we saw that this is not true for every experiment. Because of the usage of regularization methods the validation loss fluctuated slightly in some trials we believe that means the model is adjusting parameters accordingly.

### **3.3.4.3 Pooling Layer**

For the pooling layer we chose between Global Average Pooling and Global Max Pooling strategy. Since there isn't much study related to pooling methods on LSTM networks, except (Kao et al., 2020) (at the time of the present study), we tried both methods in a mutually exclusive way. Moreover, we decided the output number of the LSTM layer by modeling with different numbers of hidden cells and comparing the loss and accuracy rates of their results. With the 32 output nodes, the model loss was seriously higher than others. Using 128 and 256 output nodes did not cause LSTM Output Nodes for SLAM and DLAM a severe decrease in loss, so we chose 64 as the output nodes of both models, as the lesser parameter lets faster training.

### **3.3.4.4 Overcoming The Overfitting**

After we trained the models with our data, we tested it on the dataset which is previously unseen. We did this to see if the models generalize well enough to simulate real world scenario. If the model generalized well enough it means that the model performs well enough with new data. But if the models perform well while training but not with test data it means that the model overfits. This means that in fact it memorized the input data not learning the general features of it. Nevertheless, if the model performs bad in both with training and test data it means the model underfits. According to the study (Ruizendaal, 2017), the solutions against overfitting could be listed as early stopping, adding more data or augmenting existing data, generalizable data collection and inclusion to the model, building a simple model and regularization. Those were taken into consideration during the model building process.

## **3.4 The Document Level Analysis Model (The DLAM)**

We created a simple initial model for DLAM, starting with one BiLSTM layer. We created the summarization of DLAM using visualization utilities provided by the

Keras. Those utilities give a review about layers and trainable/non-trainable parameters as in Table 3.5 below.

Table 3.5: Model summary for the initial DLAM.

| Layer (type)                    | Output Shape      | Param |
|---------------------------------|-------------------|-------|
| embedding_1 (Embedding)         | (None, None, 64)  | 64064 |
| bidirectional_1 (Bidirectional) | (None, None, 128) | 66048 |
| dropout_3 (Dropout)             | (None, None, 128) | 0     |
| global_average_pooling1d (Glob  | (None, 128)       | 0     |
| dropout_4 (Dropout)             | (None, 128)       | 0     |
| dense_2 (Dense)                 | (None, 128)       | 16512 |
| dropout_5 (Dropout)             | (None, 128)       | 0     |
| dense_3 (Dense)                 | (None, 1)         | 129   |
| Total params : 146,753          |                   |       |
| Trainable params : 146,753      |                   |       |
| Non-trainable params : 0        |                   |       |

We added one-layer BiLSTM at a time and observed the performance of the model until it reached the optimum point. Other than the BiLSTM layer depth, there were other hyper-parameters such as batch size, number of epochs, filter size, optimization algorithm, dropout rate etc. To decide the values for those parameters, we experimented with several different values (Table 3.6). We employed EarlyStopping<sup>14</sup> from Keras with a patience of 3 to fine-tune those hyperparameters in the language modeling task. The best values for the parameters are shown bold in Table 3.6

We also used history object observe the training and validation loss for comparison, as well as the training and validation accuracy. By the help of history object, we observed the change in those parameters in each epoch. Moreover, the metrics stored in history object were also effective to use while plotting.

### Impact of the number of BiLSTM Layers

Initially, we experimented with the numbers of stacked BiLSTM layers. We fixed the hidden layer cells to 64 and, maximum sentence length to 100K, using 4 grams. We planned to increase the number of BiLSTM layers starting with one. We observed that the prediction loss decreases when the number of LSTM layers is more. Table 3.7 shows the evaluation loss values and the total parameters of the DLAM. Increasing the number of stacked BiLSTM layers makes the DLAM deeper; hence the model learns more features from the dataset, thus starts to decrease loss but increasing training time. Also, the more the BiLSTM layers mean, the more overfitting, so we used 2 stacked BiLSTM layers with the following parameters.

<sup>14</sup> Early Stopping: [https://keras.io/api/callbacks/early\\_stopping/](https://keras.io/api/callbacks/early_stopping/) (retrieved on 18 Mar 2020)

Table 3.6: Parameters For The DLAM

| Parameter Name              | Value1 | Value2  | Value3  | Value4 |
|-----------------------------|--------|---------|---------|--------|
| Embedding Dimensions        | 64     | 128     | 256     | 1024   |
| Max Sequence Length         | 50K    | 100K    | 150K    | 200K   |
| Sequence Length             | 256    | 512     | 1024    | -      |
| Dropout Rate                | 0.01   | 0.1     | 0.2     | 0.5    |
| Optimizer                   | Adam   | RMSprop | Adagrad | SGD    |
| Number of LSTM Hidden Cells | 64     | 128     | 256     | 512    |
| Number of BiLSTM Layers     | 1      | 2       | 3       |        |
| Max Features                | 1000   |         |         |        |

Table 3.7: The effects of BiLSTM Layers on Validation Loss Change

| BiLSTM Layers | Loss   | Trainable Parameters |
|---------------|--------|----------------------|
| 1             | 0.0414 | 437,249              |
| 2             | 0.0319 | 536,065              |
| 3             | 0.0206 | 634,881              |

In following experiments we used 2 stacked BiLSTM layers, with both have 64 hidden cells and decreased the number of epochs to 7 for each parameter.

#### Impact of maximum sentence length

We trained DLAM with the sentence lengths of 10K, 25K, 50K, and 100K, again fixing other parameters. With 10K, 25K, and 50K, we saw that the model loss did not reduce. Therefore, we understood that the sentences with a length of 100K was enough to represent the full document. Since, the longer the sequence length means the longer the training time and overfitting, we did not try sentence length higher than 100K, and fixed this value to 100K in DLAM.

#### Impact of Regularization and Normalization

To make our training noisy, we trained neural network with dropout rates 0.01, 0.1, 0.2, and 0.5 to find the best one for our data by keeping other parameters fixed. Since the Keras supports Variational RNNs we used 0.5 for *recurrent\_dropout* parameter for each BiLSTM layers. And with a dropout rate of 0.2. After we added dropout rate parameters in BiLSTM layers, we removed the initially added dropout layers following each BiLSTM layers. Together with Variational RNN technique and a dropout rate of 0.2 we observed late convergence of losses and accuracies as in Figure 3.7a, Figure 3.7b, we show the results for DLAM in Figures 3.6a and 3.6b below.

Table 3.8: The effects of Sentence Length on Validation Losses

| Sentence Length | Loss   | Accuracy |
|-----------------|--------|----------|
| 10K             | 0.0526 | 0.9780   |
| 25K             | 0.0371 | 0.9835   |
| 50K             | 0.0301 | 0.9835   |
| 100K            | 0.0277 | 0.9862   |

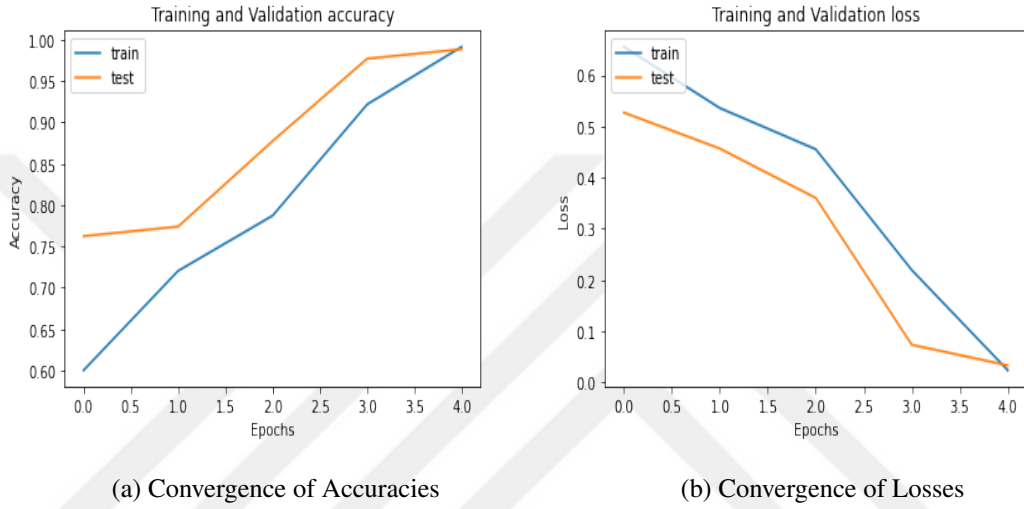


Figure 3.6: Convergence Without Variational RNN in the DLAM

As we see in Figures 3.7a, 3.7b above, the training losses are higher than validation losses. Although we expect the otherwise, according to the Keras documentation<sup>15</sup>, it may be observed with the usage of Regularization Mechanisms such as Dropout and L1/L2 weight regularization, since those mechanisms are turned off during testing period. We checked our initial loss before applying regularization methods as suggested in (Karpathy, 2019). Since our training and validation sets are divided as 80:20, for binary classification problem we expect the value as the result of the calculation.

$$-0.2\ln(0.5) - 0.8\ln(0.5) = 0.693147$$

The initial loss of the DLAM model outputs as 0.6425, with the values so close at hand, it implies that the DLAM starts learning process randomly as expected.

### Impact of Optimizers

<sup>15</sup> Losses : [https://keras.io/getting\\_started/faq/#why-is-my-training-loss-much-higher-than-my-testing-loss](https://keras.io/getting_started/faq/#why-is-my-training-loss-much-higher-than-my-testing-loss) (retrieved on: 20 Mar 2020)

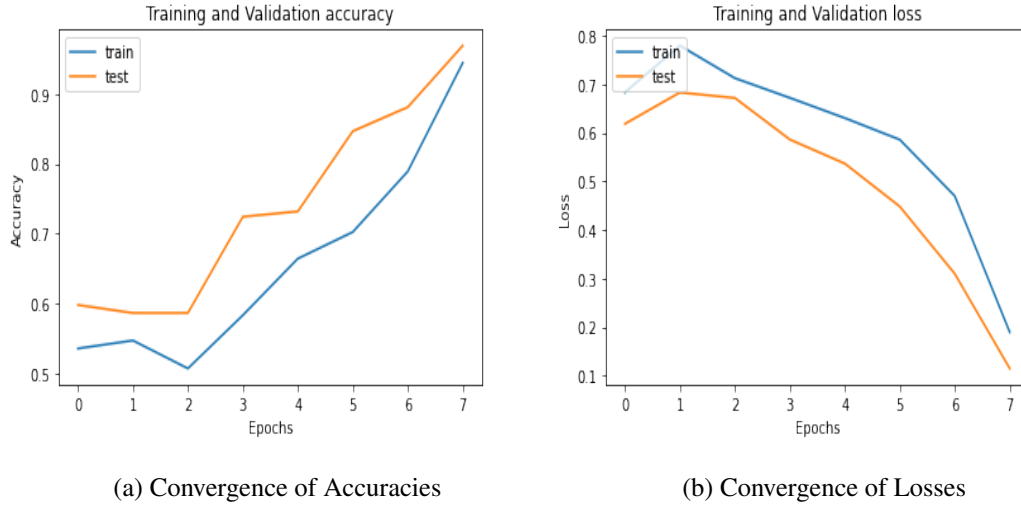


Figure 3.7: Convergence With Variational RNN in the DLAM

Also, we trained on different optimizers with their default configurations. We found that Adam and RMSprop achieved similar results. We preferred to use Adam since it showed slightly better performance than RMSprop.

#### Impact of n-grams

We observed the increase in n-grams affects in a positive way, such that the loss decreases and accuracy increases as seen in Table 3.9.

Table 3.9: The effects of n-grams on Validation Losses

| n-grams | Loss   | Accuracy |
|---------|--------|----------|
| 2       | 0.1833 | 0.9683   |
| 3       | 0.1179 | 0.9770   |
| 4       | 0.0301 | 0.9890   |

#### Impact of Pooling Methods

For the pooling layer we chose between Global Average Pooling and Global Max Pooling strategy. We observed that the Global Max Pooling outperformed the Global Average Pooling.

#### Impact of the number of LSTM Hidden Cells

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. With the 256 output nodes, the model loss was higher than others and the convergence was early in terms of epoch numbers. But did not cause a significant loss cause compared to 128, so we chose 128 as the output nodes of the DLAM

layer, as the lesser parameter allows for faster training. We also used different output modes provided by TextVectorization in the Keras, such as TF-IDF. However, with the number of 1000 maximum features, TF-IDF performed poorly and we increased the maximum features up to 10K. Increasing the maximum features significantly increased the training time and trainable parameters close to 16M. Since 4-grams are more effective in training time we preferred n-grams. We used the change of losses as a sign for the parameters to be the most suitable for the DLAM and prepared it accordingly. The proposed neural network model may be depicted as in Table 3.10.

Table 3.10: Model summary for the proposed DLAM.

| Layer (type)                    | Output Shape      | Param  |
|---------------------------------|-------------------|--------|
| embedding (Embedding)           | (None, None, 256) | 256256 |
| bidirectional_1 (Bidirectional) | (None, None, 256) | 394240 |
| bidirectional_2 (Bidirectional) | (None, None, 256) | 394240 |
| dropout_1 (Dropout)             | (None, None, 256) | 0      |
| global_max_pooling1d (Global)   | (None, 256)       | 0      |
| dropout_1 (Dropout)             | (None, 256)       | 0      |
| dense_1 (Dense)                 | (None, 128)       | 32896  |
| dropout_2 (Dropout)             | (None, 128)       | 0      |
| dense_2 (Dense)                 | (None, 1)         | 129    |
| Total params : 1,077,761        |                   |        |
| Trainable params : 1,077,761    |                   |        |
| Non-trainable params : 0        |                   |        |

After deciding the parameters we trained our model for different number of epochs to understand how our model performs.

**Impact of number of epochs** Although we employed dropout and Variational RNNs we experienced overfitting with 15 epochs. Validation accuracy seems to reach the highest accuracy before the training accuracy. As we see on Figure 3.8, the training and validation accuracies and losses converges close to the sixth epoch. After seven epochs we see that our model is adapting itself according to the test set.

For this case, to prevent overfitting we stopped the training when the validation loss was no longer decreasing. Then we trained and tested our model with five epochs to obtain maximum accuracy and minimum loss.

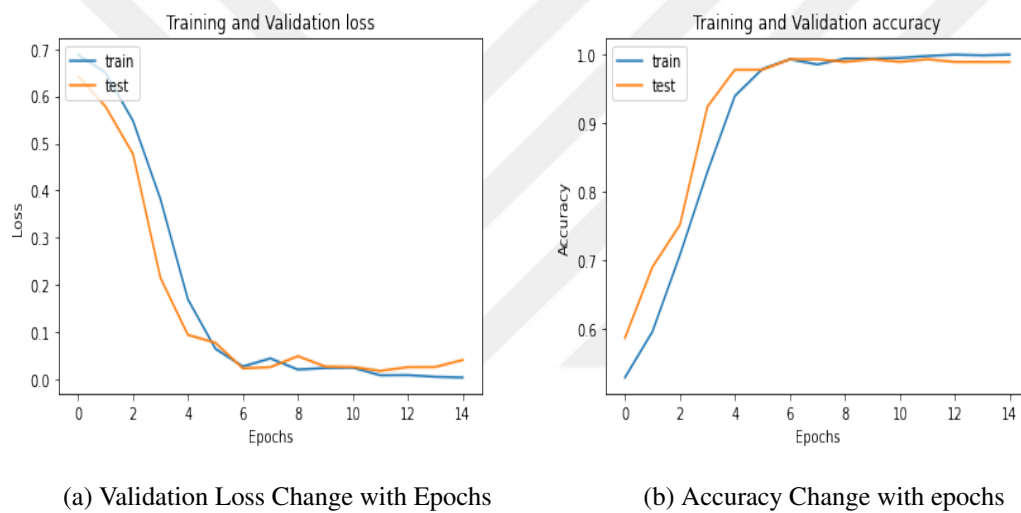


Figure 3.8: Overfitting Sample on the DLAM

### 3.5 The Sentence Level Analysis Model (The SLAM)

We began with examining dataset for the SLAM. For maximum sentence length we selected the group that represents 95% of the dataset. When we visualize the dataset as in Figure 3.9, we decided to use 16, 25, and 30 for the maximum sentence length.

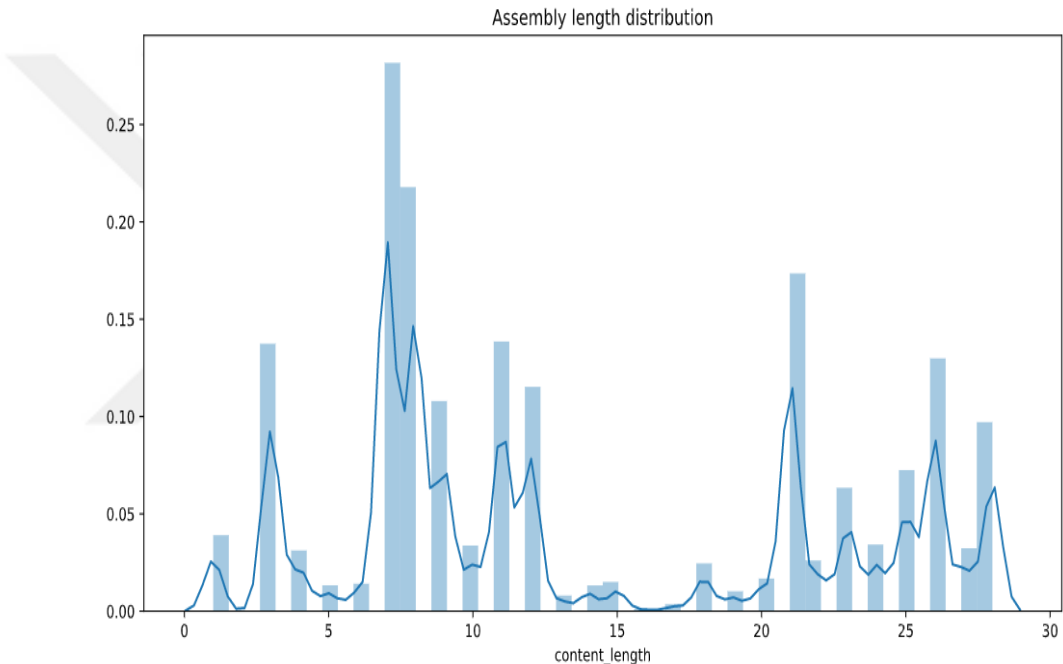


Figure 3.9: Sentence Length Distribution for the SLAM

Using the model we created in the DLAM we experimented with the values in Table 3.11 and with maximum feature count 1122 ( unique word count). We started with 2 BiLSTM layers.

Table 3.12 shows the summarization of SLAM created by the Keras with a total of 834,177 trainable parameters.

Table 3.11: Parameter Values for The SLAM

| Parameter Name             | Value1 | Value2  | Value3  | Value4 |
|----------------------------|--------|---------|---------|--------|
| Embedding Dimensions       | 128    | 256     |         |        |
| Max Sequence Length        | 8      | 16      | 32      |        |
| Sequence Length            | 16     |         |         | -      |
| Dropout Rate               | 0.01   | 0.1     | 0.2     | 0.5    |
| Optimizer                  | Adam   | RMSprop | Adagrad | SGD    |
| Number of LSTM Output Node | 128    | 256     | 512     | 1024   |
| Number of BiLSTM Layers    | 1      | 2       | 3       |        |
| Max Features               | 1122   |         |         |        |

### Impact of maximum sentence length

We trained SLAM with the sentence lengths of 9, 16 and 25. The losses are shown in Table 3.13 below.

### Impact of Optimizers

Also, we trained on several known optimizers with their default configurations such as Adam, RMSProp, Adagrad, Stochastic Gradient Descent (SGD), and SGD with momentum, fixing other parameters. We found that Adam and RMSprop achieved similar results.

### Impact of Embedding Dimension Length

With the values of 128 and 256, the SLAM didn't show any significant improvement in terms of loss as shown in Table 3.14.

### Impact of n-grams

The validation losses regarding to n-gram values for 2, 3, and 4 are in shown in Table 3.15.

The most important thing that we may infer from the outputs above is that the model's hyperparameters do not significantly affect the validation loss. Hence, the sentence-level representation of assembly instructions with the TextVectorization class may not be sufficient for this task. With this finding, for the vectorization layer, we employed different methods. Considering the assembly lengths, we examined the effectiveness of methods in Word2Vec and DistilBERT.

With experiments related to word2vec to create the vector representations of assembly instructions, we employed CBOW and Skip-Gram implementations. We fixed the value for min\_count parameter as 10 to ignore the assembly instructions that does not occur more than 10. We used different parameters for window (looking back and forward for number of window size words), and size. Using the values in Table 3.16, we trained word2vec for 8 times for 5 epochs. We fed the embedding layer in

Table 3.12: Model summary for the initial SLAM.

| Model: Initial SLAM            |                   |        |
|--------------------------------|-------------------|--------|
| Layer (type)                   | Output Shape      | Param  |
| embedding_1 (Embedding)        | (None, None, 128) | 143744 |
| bidirectional_2 (Bidirectional | (None, None, 256) | 263168 |
| bidirectional_3 (Bidirectional | (None, None, 256) | 394240 |
| dropout_3 (Dropout)            | (None, None, 256) | 0      |
| global_average_pooling1d (Glob | (None, 256)       | 0      |
| dropout_4 (Dropout)            | (None, 256)       | 0      |
| dense_2 (Dense)                | (None, 128)       | 32896  |
| dense_3 (Dense)                | (None, 1)         | 129    |
| Total params : 834,177         |                   |        |
| Trainable params : 834,177     |                   |        |
| Non-trainable params : 0       |                   |        |

Table 3.13: The effects of Maximum Sentence Lengths on Validation Losses

| Sentence Lengths | Losses |
|------------------|--------|
| 9                | 0.6646 |
| 16               | 0.6584 |
| 25               | 0.6530 |

the SLAM with the resulting vectors. Since word2vec creates and trains the word vectors, we set trainable parameter of the embedding layer to false.

After we fed the embedding layer of the SLAM with word2vec weights, using the values in the first column of Table 3.16, we had a deep neural network model as in 3.17.

We may infer from Table 3.17, that the network has more than 1M parameters but 866K parameters are trainable because of no training occurred in embedding layer. We observed that with the CBOW implementation of word2vec and when window size is 25, the loss decreased to the lowest and the accuracy increased to the highest among the other parameters. On the other hand with our setup environment, the training time doubled in time, such as one epoch took nearly three hours. We presented Table 3.18 below, which shows the effectiveness of the algorithms regarding

Table 3.14: The effects of Embedding Dimensions on Losses

| Embedding Dimensions | Losses |
|----------------------|--------|
| 128                  | 0.6462 |
| 256                  | 0.6610 |

Table 3.15: Best Results for TextVectorization parameters on the SLAM

| n-grams | Loss   | Accuracy % |
|---------|--------|------------|
| 2       | 0.6425 | 56,1       |
| 3       | 0.6673 | 54,1       |
| 4       | 0.6898 | 53,7       |

to validation losses and validation accuracies.

Table 3.18: Effects of Word2Vec parameters on the SLAM

| Algorithm | Window Size | size | Loss  | Accuracy (%) |
|-----------|-------------|------|-------|--------------|
| Cbow      | 25          | 300  | 0.554 | 61.69        |
| Skipgram  | 10          | 300  | 0.538 | 63.72        |

For the third vectorization method, we employed transformers, using DistilBERT on the same dataset. We used a pre-trained base model, *distilbert-base-uncased*, to create the embeddings for the SLAM. Since DistilBERT outputs a tuple where the first element is the *last\_hidden\_state* of the model's last layer. We first fed the BiLSTM layer using the hidden state from outputs. The model with DistilBERT embeddings is shown in Table 3.19. We used 5000 for the batch size and 25 for the maximum length. With two stacked BiLSTM layers, it took about six hours to train for one epoch. With DistilBERT, the total parameters are 67,708,673. Since we did not train the DistilBERT layers for the first experiment, the total trainable parameters are 1,345,793. 3.20 After six epochs, the model achieved 70.4% accuracy with a validation loss of 0.4340.

Table 3.16: Word2Vec Parameters for the SLAM

| Parameters  | Value 1 | Value 2   |
|-------------|---------|-----------|
| Algorithm   | CBOW    | Skip Gram |
| Window Size | 10      | 25        |
| Size        | 100     | 300       |

Table 3.17: Model summary for the initial SLAM with Word2Vec.

| Model: Initial SLAM with Word2Vec |                 |        |
|-----------------------------------|-----------------|--------|
| Layer (type)                      | Output Shape    | Param  |
| embedding_1 (Embedding)           | (None, 10, 300) | 144900 |
| bidirectional_2 (Bidirectional    | (None, 10, 256) | 439296 |
| bidirectional_3 (Bidirectional    | (None, 256)     | 394240 |
| dropout_3 (Dropout                | (None, 256)     | 0      |
| dense_5 (Dense)                   | (None, 128)     | 32896  |
| dropout_6 (Dropout) (None, 128)   | 0               |        |
| dense_6 (Dense) (None, 2)         | 258             |        |
| Total params : 1,011,590          |                 |        |
| Trainable params : 866,690        |                 |        |
| Non-trainable params : 144,900    |                 |        |

### 3.6 Summary

In this section, we presented the details of our methodology. We detailed the dataset collection process, explained the different dataset formats, the environment we used to train our neural network models, the required libraries and modules, the modeling pipeline, and the parameters used in the modeling process. Briefly, we used language modeling techniques in natural language processing (NLP) then we built the malware detection model using assembly code. To instrument our methodology, we used modules from Tensorflow and Keras libraries on Python programming language and modules from the numpy, and Matplotlib libraries. For hyperparameter tuning we used HParams and we presented the values we experimented on the parameters required in the training and testing process. Lastly, to create features of short texts, assembly instructions, we employed different vectorization methods such as TextVectorization, Word2Vec and DistilBERT.

Table 3.19: Model summary for the SLAM.

| Model: SLAM with BERT             |                  |          |
|-----------------------------------|------------------|----------|
| Layer (type)                      | Output Shape     | Params   |
| input_ids (InputLayer)            | [(None, 25)]     | 0        |
| input_attention (InputLayer)      | [(None, 25)]     | 0        |
| tf_distil_bert_model (TFDistilB   | ((None, 25, 768) | 66362880 |
| bidirectional_2 (Bidirectional    | (None, 25, 256)  | 918528   |
| bidirectional_3 (Bidirectional    | (None, 25, 256)  | 394240   |
| dropout_3 (Dropout)               | (None, 25, 256)  | 0        |
| global_max_pooling1d (Glob        | (None, 256)      | 0        |
| dropout_4 (Dropout)               | (None, 256)      | 0        |
| dense_2 (Dense)                   | (None, 128)      | 32896    |
| dropout_4 (Dropout)               | (None, 128)      | 0        |
| dense_3 (Dense)                   | (None, 1)        | 129      |
| Total params : 67,708,673         |                  |          |
| Trainable params : 1,345,793      |                  |          |
| Non-trainable params : 66,362,880 |                  |          |

Table 3.20: Effects of DistilBERT parameters on the SLAM

| Trainable Parameters | learning rate | Loss  | Accuracy % |
|----------------------|---------------|-------|------------|
| 1,345,793            | 5e-5          | 0.484 | 68,36      |
| 67,708,673           | 2e-5          | 0.434 | 70,4       |

## CHAPTER 4

### RESULTS

This section presents the results of the two models, namely the SLAM and the DLAM.

#### 4.1 Evaluation Criteria

We evaluated the performances of the models based on the F1 score. F1 score consists of precision and recall values which in themselves are performance evaluation metrics. Precision measures the ratio of correctly identified positive cases against all positive predicted cases. It is formulated as :

$$Precision = \frac{TruePositive}{(TruePositive + FalsePositive)}$$

Recall measures the ratio of correctly identified positive cases from all the actual positive cases. It is summarized as:

$$Recall = \frac{TruePositive}{(TruePositive + FalseNegative)}$$

Based on those two metrics, the F1 score is calculated. The calculation based on Precision and Recall demonstrates the harmonic mean of those metrics and is accepted as a more accurate value for model evaluation. F1 score formulated as:

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall}$$

$$F1 = \frac{2 * TruePositive}{2 * TruePositive + FalsePositive + FalseNegative}$$

The TP, FP, FN, and TN values used for calculation of F1 score are visualized in matrix created by *sklearn.metrics* package. Those values are selected according to Table 4.1 below.

|               |          | Positive         | Negative |
|---------------|----------|------------------|----------|
| Actual Values | Positive | TP               | FN       |
|               | Negative | FP               | TN       |
|               |          | Predicted Values |          |

Figure 4.1: The Confusion Matrix for reference.

## 4.2 SLAM (Sentence Level Analysis Model)

We conducted a total of 24 experiments for the SLAM, by manipulating the parameters listed in Table 3.11 and Table 3.16. We show the resulting number of correctly and incorrectly classified samples in the confusion matrix 4.2.

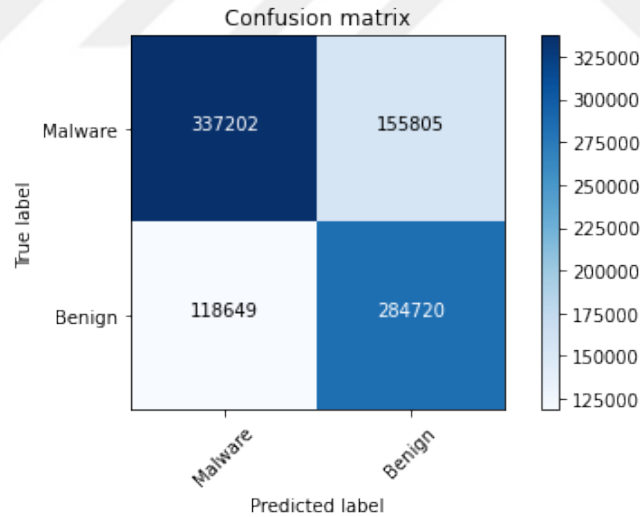


Figure 4.2: The SLAM Confusion Matrix.

The confusion matrix of the test set from the evaluation process of SLAM shows the number of true negatives (TN), the number of false negatives (FN), the number of false positives (FP), and the number of true positives (TP). Hence, TN refers to correctly recognized instructions as benign instructions. In contrast, TP refers to correctly identified instructions as malicious instructions. Also, FP shows harmless

instructions recognized as malicious, whereas the number of FN shows malicious instructions recognized as benign. Using the values from confusion matrix we may calculate precision, recall and F1 score of the SLAM in Table 4.1.

Table 4.1: F1 Score Calculation of The SLAM

| Term      | Definition                                    | Calculation                                 |
|-----------|-----------------------------------------------|---------------------------------------------|
| Precision | $\frac{TP}{TP+FP}$                            | $\frac{337202}{337202+118649} = 0.739$      |
| Recall    | $\frac{TP}{TP+FN}$                            | $\frac{337202}{337202+155805} = 0.683$      |
| F1        | $\frac{2*Precision*Recall}{Precision+Recall}$ | $\frac{2*0.739*0.683}{0.739+0.683} = 0.704$ |

### 4.3 DLAM (Document Level Analysis Model)

We conducted a total of 15 experiments for the DLAM, by manipulating the parameters listed in Table 3.6. We show the resulting number of correctly and incorrectly classified samples in a confusion matrix 4.3.

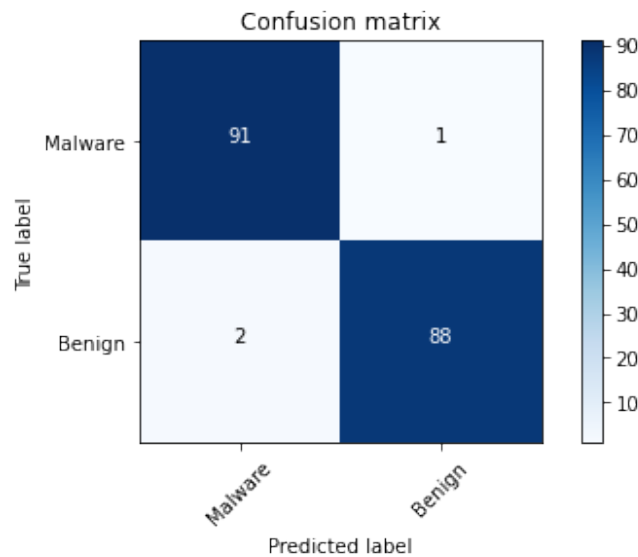


Figure 4.3: The DLAM Confusion Matrix.

The confusion matrix of the test set from the evaluation process of DLAM shows the number of true negatives (TN), the number of false negatives (FN), the number

of false positives (FP), and the number of true positives (TP). Hence, TN refers to correctly recognized documents as benign documents. In contrast, TP refers to the correctly identified document as malicious documents. Also, FP shows harmless documents recognized as malicious, whereas the number of FN shows malicious documents recognized as benign. Using the values from confusion matrix we may calculate precision, recall and F1 score of the DLAM in Table 4.2.

Table 4.2: F1 Score Calculation of The DLAM

| Term      | Definition                                    | Calculation                                 |
|-----------|-----------------------------------------------|---------------------------------------------|
| Precision | $\frac{TP}{TP+FP}$                            | $\frac{91}{91+2} = 0.978$                   |
| Recall    | $\frac{TP}{TP+FN}$                            | $\frac{91}{91+1} = 0.989$                   |
| F1        | $\frac{2*Precision*Recall}{Precision+Recall}$ | $\frac{2*0.978*0.989}{0.978+0.989} = 0.983$ |

#### 4.4 Comparison

The findings of the models are shown in Table 4.3.

Table 4.3: Performances of The Models

| Model | Precision (%) | Recall (%) | F1 (%) |
|-------|---------------|------------|--------|
| SLAM  | 73.9          | 68.3       | 70.4   |
| DLAM  | 97.8          | 98.9       | 98.3   |

The information that we can deduct from the comparison table is that The DLAM outperforms the SLAM. To explain why our initial model underperformed, we must focus primarily on the underlying and processed dataset. Due to the short text form at most nine assembly instructions used in SLAM, the applied feature selection and extraction mechanisms did not fully represent the necessary features. Since we labeled each assembly instruction as malicious or benign depending on the source, we also labeled the shared instructions between the classes that made the SLAM less performant. When the F1 score is considered, we may infer that it includes meaningful information and patterns to achieve a 70.40% F1 score. On the other hand, the documents of assembly instructions which consist of more than one instruction, are

contextually related. In addition to the relation of words in an assembly instruction, there are also relations among the instructions documents-wise. Thus, the documents with longer and more complex structures include more meaningful information and more relations than instructions, resulting in the DLAM achieving a 98.30% F1 score. In summary, the results of the final experiments on the two models suggest that the Document Level Assembly Analysis Model (DLAM) exhibits a better structure for Stacked BiLSTM based deep learning language modeling compared to the Sentence Level Assembly Analysis Model (SLAM).

## 4.5 Discussion

The evolution of the studies related to malware detection began with the signature extraction methods applied to executables. The signature databases, distributed over the internet to the end-users online or offline, grew in time. However, malware developers used innovative approaches to bypass anti-malware applications. In answering the methods applied to evade signature-based detection mechanisms, AI-based detection methods became the best candidate. Due to the advancements in artificial intelligence, studies related to AI-based detection mechanisms emerged. The early AI-based studies employed Machine learning (ML) classification algorithms to identify malicious and benign and classify malware families. Among the most studied and used classification algorithms may be listed as Random Forest (RF), Support Vector Machine (SVM), and Decision Tree (DT) algorithms. Those classification algorithms were applied to data obtained from malicious or benign files. ML classification methods require meaningful extracted and selected features in mathematical representation from the data as in any computer system. The feature extraction methods to deploy for ML classification algorithms are cumbersome due to time, domain knowledge, and effort. So, in recent studies, since deep neural networks have an advantage over ML in feature selection, extraction, and learning process, the focus was shifted from ML to DL. Nowadays, deep neural network architectures are widely researched in academic studies to identify malicious and benign software and classify malicious files to their corresponding families. Since deep learning isn't a single approach and the number of architectures and topologies is wide and varied, researchers proposed various architectures and methods such as convolutional neural networks (CNNs) and recurrent neural networks (RNNs), and attention mechanisms.

There are several recent studies focus on assembly code and detect malicious software employing deep learning techniques. Studies employing RNN, LSTM or CNN models for binary classification and also opcode, operand sequences as binary forms or word forms. The study (Lu, 2019) used the LSTM network to distinguish executables between malicious or benign with opcode sequences as words. Another study (Jahromi et al., 2020) used opcode and byte-code sequences with LSTM based neural network to detect malware in IoT systems. On the other hand, (Kumar et al., 2018) and (Khan et al., 2019) used binary form of the executables to create the image representations and applied CNN based architectures to their dataset. Since we aimed to detect sentence-level and document-level assembly sentences as benign and malicious in this study, the accuracy rates of our proposed methods are in terms of assembly instruction sequence classification rather than binary file or byte-code se-

quences classification. Also, not all the studies employ the same processes regarding feature extraction methods; some use byte-code, some used opcode, while others used complete binary forms of executables. So, even if the meaning of the accuracy of the methods and our proposed methods are different, we wanted to compare them to evaluate the performance/success of our proposed methods. The Table 4.4 below, gives information about the employed deep learning architectures and the accuracy rate of those models regarding percentage.

Table 4.4: Comparison of Models Used to Detect Malware

| Study                  | Architecture | Accuracy    |
|------------------------|--------------|-------------|
| (Kumar et al., 2018)   | CNN          | 97.1        |
| (Khan et al., 2019)    | ResNet       | 88.36       |
| (Khan et al., 2019)    | GoogleNet    | 74.5        |
| (Lu, 2019)             | LSTM         | 97,26       |
| (Jahromi et al., 2020) | LSTM         | 99,01 (AUC) |
| The SLAM               | BiLSTM       | 70.4        |
| The DLAM               | BiLSTM       | 98.3        |

## 4.6 Open Problems

While studying the recent articles, and during our study we have noticed that some points may be considered as open problems in malware detection domain. The first problem is the datasets used in the researchs. Since the studies related to malware domain doesn't have a common or benchmark datasets, each research tried to create their own dataset as we described in Section 2.3 like (Anderson & Roth, 2018; Harang & Rudd, 2020). This leads to the inability of making general comparisons of the generated models and the models' performances. For this reason, the topics that researchers should avoid in methods using deep learning have become more evident recently. Problems such as collecting datasets by researchers, making them accessible, and creating datasets that everyone may use come to the fore more. Besides, it is seen that it is possible to generalize the deep learning models only with big data or a common dataset. Therefore, it is not recommended to generalize research made with limited resources. The issues that need special attention are briefly summarized in the recently published article (Lones, 2021) on this aspect. Following the guidelines provided by the article (Lones, 2021), we shared our dataset, which we used within the scope of our study, in our Github repository<sup>1</sup>, including hash values and also the DLAM architecture as python notebook. Furthermore, in our experiments, we tried to ensure that our models produce healthy and comparable results by changing the ratios

<sup>1</sup> Repository: [https://github.com/d-demirci/binary\\_classification](https://github.com/d-demirci/binary_classification)

and places of the training set, test set, and validation sets (without data leakage among them) and by using the data sets, we obtained from different sources. For example, when we change the validation set to be used as test dataset and the test dataset as validation dataset we saw that the model accuracy changed %0.5 in a positive way as shown in files <sup>2</sup>, <sup>3</sup>. Another problem in deep learning-based models is randomly splitting the dataset into training, validation, and test sets. The studies rely on the provided algorithms by libraries such as scikit-learn and the Keras. In this study, to overcome this problem, and since it uses *np.random.RandomState(seed)*, we preferred to use the method, (*tf.keras.preprocessing.text\_dataset\_from\_directory*), provided by the Keras library, among the other methods. While using this library, it is critical to use the same seed value when separating the dataset. This way, according to the numpy documentation<sup>4</sup> the generated random value<sup>5</sup> is ensured to be the same every time, hence the focus may be on developing the model. Otherwise, the change of random number in each run will effect the consistency of the model. Especially after the model matures, we ran the model with different seed values. This approach made the model consume different samples in the dataset as training and validation sets. We experimented with 5 different values (24,49,63,75,82) for seed value and inspect the change in the accuracy of the DLAM as in table 4.5. Another method to calculate the performance of the DLAM may be as the mean of the accuracies in the table. Namely, we may calculate the sum of accuracy rates and divide it by the number of runs. The result is given in table ???. Besides, we calculated False Positive Rate of different runs. The false positive rate is calculated as  $FP/FP+TN$ , where FP is the number of false positives and TN is the number of true negatives. It means the probability of a false alarm and in the scope of the present study, it means that a benign file will be detected as malware. And also this is another open problem in the models created using deep learning since there are many ways to express the accuracy or the performance of the models. The choice of which one/ones to use depends on the concerns of the studies<sup>6</sup>. Some studies like (Jahromi et al., 2020) use AUC and some studies like (Vasan et al., 2020) use F1 Score and some studies use the accuracy rates (Kumar et al., 2018; Lu, 2019) calculated using confusion matrix. In the present study we stucked with the F1 Score as we explained in Section 4.1.

---

<sup>2</sup> Validation Set in fit method : [https://github.com/d-demirci/binary\\_classification/blob/master/00\\_last\\_text\\_classification\\_dlam\\_sorel\\_test\\_for\\_val.html](https://github.com/d-demirci/binary_classification/blob/master/00_last_text_classification_dlam_sorel_test_for_val.html)

<sup>3</sup> Testing Set in fit method: [https://github.com/d-demirci/binary\\_classification/blob/master/00\\_last\\_text\\_classification\\_dlam\\_sorel.html](https://github.com/d-demirci/binary_classification/blob/master/00_last_text_classification_dlam_sorel.html)

<sup>4</sup> NumpyPRNG:[https://numpy.org/doc/stable/reference/random/bit\\_generators/pcg64.html#numpy.random.PCG64](https://numpy.org/doc/stable/reference/random/bit_generators/pcg64.html#numpy.random.PCG64) Retrieved on : 5 Sep 2021

<sup>5</sup> Random Value Generation: [https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/keras/preprocessing/dataset\\_utils.py#L123](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/keras/preprocessing/dataset_utils.py#L123) Retrieved on: 5 Sep 2021

<sup>6</sup> Binary Classification Metrics: <https://neptune.ai/blog/evaluation-metrics-binary-classification> Retrieved on: 5 Sep 2021

Table 4.5: Seed Value Effect on Performance

| run #          | seed value | accuracy (%) | FP             | FN | TP | TN | F1 Score (%) | FPR    |
|----------------|------------|--------------|----------------|----|----|----|--------------|--------|
| 1              | 24         | 98,3         | 3              | 0  | 87 | 92 | 98,3050847   | 0,0315 |
| 2              | 49         | 99,4         | 0              | 1  | 89 | 92 | 99,4413408   | 0      |
| 3              | 63         | 99,4         | 1              | 0  | 90 | 91 | 99,4475138   | 0,0108 |
| 4              | 75         | 98,9         | 0              | 2  | 88 | 92 | 98,8764045   | 0      |
| 5              | 82         | 98,9         | 0              | 2  | 88 | 92 | 98,8764045   | 0      |
| Mean Acc:98,98 |            |              | Mean F1:98,989 |    |    |    |              |        |

## CHAPTER 5

### CONCLUSION AND FUTURE WORK

#### 5.1 Conclusion

Malware detection methods have been studied since the early days of information systems. We first see a malware variant in 1971, namely Creeper System<sup>1</sup>. Creeper System caused the creation of its opponent, which is another malware, namely Reaper<sup>2</sup>, and the need for a systematical approach to detect malware instead of deploying another malware to remove existing malware. Since then, the sheer increase in usage of the Internet and personal computers has made it easier for cybercriminals to expose Internet users to widespread and damaging threats. The early detection mechanisms relied on signatures identifying malware. However, techniques developed by malware authors to bypass antimalware applications increased the need for innovative and fully automated malware detection methods. In the light of those needs, AI-based detection methods became the best candidate due to the advancements in the artificial intelligence area. The early AI-based studies employed Machine learning (ML) classification algorithms to classify the data obtained from malicious and benign software. Since ML classification algorithms require time and effort for feature selection and extraction, they do not provide fully automated methods. So, in recent studies, the focus was shifted 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 studies to classify malicious and benign software. The present research found that malware detection by using assembly instructions with a Stacked BiLSTM based DL Language Model is feasible. We found that incorporating techniques from natural language processing (NLP), specifically, document-level analysis with word embedding and bidirectional LSTMs (BiLSTM), dramatically improves the performance. We also discovered that we could obtain even better performance by including a Variational RNN technique in our model. The DLAM model was able to detect files with an average accuracy of over 98%. We conjecture that the context obtained from assembly instructions in DLAM is the key to getting this strong performance. On the other hand, the SLAM, our sentence-level analysis model performed poorly, due to the short text nature of assembly instructions.

---

<sup>1</sup> Creeper: <http://virus.wikidot.com/creeper> (retrieved on: 15 Jun 2021)

<sup>2</sup> Reaper: <http://virus.wikidot.com/nematode> (retrieved on: 15 Jun 2021)

## 5.2 Limitations and Future Work

The dataset consisted of the malware and benign files used in the present study is limited to a few hundred files. Future research should address using improvements of the data processing pipeline, developing a service of API for extracting assembly instructions to collect them automatically for a given PE file. On the other hand, the feature selection method used in the present study depends on the number of unique words in assembly instructions. The number may be increased with a different sized dataset to represent assembly instructions in a more general form. Moreover, word embedding methods are limited to TF-IDF and n-grams provided by the TextVectorization module from Keras. Since deep-learning architectures are highly dependent on computational power and the environment we experimented on is relatively weak, we also did not increase the embedding dimensions. Due to the short text form at most nine assembly instructions used in SLAM, the applied feature selection and extraction mechanisms did not fully represent the necessary features. For future work, since the DLAM model could detect files with an average accuracy above 98%; more can be done to investigate why applying NLP techniques is effective in detecting malware with regard to document-level analysis. The usage of an automatic hyperparameter tuning improved our model's overall accuracy, and the use of Variational RNN as a regularization technique prevented early overfitting on DLAM. Apart from the parameters, other methods can be considered—for example, experiments involving different word embedding algorithms (e.g., GloVe<sup>3</sup>, BERT<sup>4</sup>) and using extensive malware databases such as EMBER to create statically ready and more general embedding representations, like the ones<sup>5</sup> used in NLP tasks. Additionally, Positional Embedding methods used in Generative Pre-Train Transformers (GPT-2) would be worthwhile to research to increase the accuracy of the SLAM. Since attention-based mechanisms (Vaswani et al., 2017) deployed in GPT-2 (Radford & Narasimhan, 2018; Radford et al., 2019) architecture is claimed better in dealing with short texts. Further research into the possible benefits of stacked BiLSTM and applying different embedding methods in this problem domain would be of great interest. Finally, we believe that it is essential to determine the steps and comparison methods of deep learning-based studies as a scientific study methodology in the academic environment. To conclude, it is evident that publishing the benchmark datasets to be used on a domain basis and ensuring that all parties implement the steps specified in the article (Lones, 2021) will significantly contribute to the quality and re-creation of the future studies.

---

<sup>3</sup> GloVe: <https://nlp.stanford.edu/projects/glove/> (retrieved on: 21 Jan 2021)

<sup>4</sup> BERT: <https://github.com/google-research/bert> (retrieved on: 10 Feb 2021 )

<sup>5</sup> Embedding Repository: <http://vectors.nlpl.eu/repository/> (retrieved on: 27 July 2021)

## REFERENCES

- Acarturk, C., Sirlanci, M., Balikcioglu, P. G., Demirci, D., Sahin, N., & Kucuk, O. A. (2021). Malicious code detection: Run trace output analysis by lstm. *IEEE Access*, 9, 9625–9635. <https://doi.org/10.1109/ACCESS.2021.3049200>
- Anderson, H., & Roth, P. (2018). Ember: An open dataset for training static pe malware machine learning models.
- Bengio, Y., Ducharme, R., Vincent, P., & Janvin, C. (2000). A neural probabilistic language model. *J. Mach. Learn. Res.*
- Bilar, D. (2007). Opcodes as predictor for malware. *Int. J. Electron. Secur. Digit. Forensics*, 1, 156–168.
- Christodorescu, M., Jha, S., Seshia, S., Song, D., & Bryant, R. (2005). *Semantics-aware malware detection*. <https://doi.org/10.1109/SP.2005.20>
- Collobert, R., & Weston, J. (2008). A unified architecture for natural language processing: Deep neural networks with multitask learning. *Proceedings of the 25th International Conference on Machine Learning*, 160–167. <https://doi.org/10.1145/1390156.1390177>
- Cornegruta, S., Bakewell, R., Withey, S., & Montana, G. (2016). Modelling radiological language with bidirectional long short-term memory networks, 17–27. <https://doi.org/10.18653/v1/W16-6103>
- Donahue, J., Hendricks, L., Guadarrama, S., Rohrbach, M., Venugopalan, S., Saenko, K., & Darrell, T. (2014). Long-term recurrent convolutional networks for visual recognition and description. *Arxiv, PP*. <https://doi.org/10.1109/TPAMI.2016.2599174>
- Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12, 2121–2159.
- Eisenstein, J. (2019). *Introduction to natural language processing*. The Mit Press.
- El Merabet, H., & Hajraoui, A. (2019). A survey of malware detection techniques based on machine learning. *International Journal of Advanced Computer Science and Applications*, 10. <https://doi.org/10.14569/IJACSA.2019.0100148>
- Elisan, C. C. (2012). Malware, rootkits and botnets a beginner’s guide. *McGraw-Hill Education*, 1(2), 9625–9635.
- Gal, Y., & Ghahramani, Z. (2016). A theoretically grounded application of dropout in recurrent neural networks.

- Gibert, D., Mateu, C., & Planes, J. (2020). The rise of machine learning for detection and classification of malware: Research developments, trends and challenges. *Journal of Network and Computer Applications*, 153, 102526. <https://doi.org/10.1016/j.jnca.2019.102526>
- Graves, A., & Jaitly, N. (2014). Towards end-to-end speech recognition with recurrent neural networks, II–1764II–1772.
- Griffin, K., Schneider, S., Hu, X., & Chiueh, T.-c. (2009). Automatic generation of string signatures for malware detection. In E. Kirda, S. Jha, & D. Balzarotti (Eds.), *Recent advances in intrusion detection* (pp. 101–120). Springer Berlin Heidelberg.
- Harang, R., & Rudd, E. M. (2020). Sorel-20m: A large scale benchmark dataset for malicious pe detection.
- He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep residual learning for image recognition. *CoRR*, *abs/1512.03385*. <http://arxiv.org/abs/1512.03385>
- He, X., Cai, D., & Niyogi, P. (2005). Laplacian score for feature selection. *NIPS*.
- Hinton, N., G. and Srivastava, & Swersky, K. (2012). Lecture 6d - a separate, adaptive learning rate for each connection. slides of lecture neural networks for machine learn. <https://www.cs.toronto.edu/~hinton/coursera/lecture6/lec6.pdf>
- Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8), 1735–1780. <https://doi.org/10.1162/neco.1997.9.8.1735>
- Jahromi, A. N., Hashemi, S., Dehghantanha, A., Parizi, R. M., & Choo, K.-K. R. (2020). 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*, 4(5), 630–640. <https://doi.org/10.1109/tetci.2019.2910243>
- John, V. (2017). A survey of neural network techniques for feature extraction from text. *CoRR*, *abs/1704.08531*. <http://arxiv.org/abs/1704.08531>
- Kao, C.-C., Sun, M., Wang, W., & Wang, C. (2020). A comparison of pooling methods on lstm models for rare acoustic event classification. *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 316–320. <https://doi.org/10.1109/ICASSP40776.2020.9053150>
- Karpathy, A. (2019). *A recipe for training neural networks*. <http://karpathy.github.io/2019/04/25/recipe/> (accessed: Feb 2020)
- Khan, R. U., Zhang, X., & Kumar, R. (2019). Analysis of resnet and googlenet models for malware detection. *Journal of Computer Virology and Hacking Techniques*, 15(1), 29–37. <https://doi.org/10.1007/s11416-018-0324-z>
- Khraisat, A., Gondal, I., Vamplew, P., & Kamruzzaman, J. (2019). Survey of intrusion detection systems: Techniques, datasets and challenges. *Cybersecurity*, 2(1), 20. <https://doi.org/10.1186/s42400-019-0038-7>

- Kingma, D. P., & Ba, J. (2017). Adam: A method for stochastic optimization.
- Krcál, M., Švec, O., Bálek, M., & Jasek, O. (2018). Deep convolutional malware classifiers can learn from raw executables and labels only. *ICLR*.
- Kumar, R., Xiaosong, Z., Khan, R. U., Ahad, I., & Kumar, J. (2018). Malicious code detection based on image processing using deep learning, 81–85. <https://doi.org/10.1145/3194452.3194459>
- Le, Q. V., & Mikolov, T. (2014). Distributed representations of sentences and documents. *CoRR*, *abs/1405.4053*. <http://arxiv.org/abs/1405.4053>
- Lee, Y., Kwon, H., Choi, S.-H., Lim, S.-H., Baek, S. H., & Park, K.-W. (2019). Instruction2vec: Efficient preprocessor of assembly code to detect software weakness with cnn. *Applied Sciences*, 9(19). <https://doi.org/10.3390/app9194086>
- Lewis, D. (2000). Feature selection and feature extraction for text categorization. <https://doi.org/10.3115/1075527.1075574>
- Li, Z., Yang, Y., Liu, J., Zhou, X., & Lu, H. (2012). Unsupervised feature selection using nonnegative spectral analysis. *Proceedings of the National Conference on Artificial Intelligence*, 2, 1026–1032.
- Li, Z., Liu, J., Yang, Y., Zhou, X., & Lu, H. (2014). Clustering-guided sparse structural learning for unsupervised feature selection. *IEEE Transactions on Knowledge and Data Engineering*, 26(9), 2138–2150. <https://doi.org/10.1109/TKDE.2013.65>
- Lin, Y., Lai, Y.-C., Lu, C.-N., Hsu, P.-K., & Lee, C.-Y. (2014). Three-phase behavior-based detection and classification of known and unknown malware. *Security and Communication Networks*, 8. <https://doi.org/10.1002/sec.1148>
- Lones, M. A. (2021). How to avoid machine learning pitfalls: A guide for academic researchers. *CoRR*, *abs/2108.02497*. <https://arxiv.org/abs/2108.02497>
- Lu, R. (2019). Malware detection with LSTM using opcode language. *CoRR*, *abs/1906.04593*. <http://arxiv.org/abs/1906.04593>
- Mikolov, T., Sutskever, I., Chen, K., Corrado, G., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. *Advances in Neural Information Processing Systems*, 26.
- Moon, D., Im, H., Lee, J., & Park, J. (2014). Mlds: Multi-layer defense system for preventing advanced persistent threats. *Symmetry*, 6(4)(1), 997–1010.
- Moskovitch, R., Feher, C., Tzachar, N., Berger, E., Gitelman, M., Dolev, S., & Elovici, Y. (2008). Unknown malcode detection using opcode representation (D. Ortiz-Arroyo, H. L. Larsen, D. D. Zeng, D. Hicks, & G. Wagner, Eds.), 204–215.
- Mosli, R., Li, R., Yuan, B., & Pan, Y. (2017). A behavior-based approach for malware detection. In G. Peterson & S. Sheno (Eds.), *Advances in digital forensics xiii* (pp. 187–201). Springer International Publishing.

- Pascanu, R., Mikolov, T., & Bengio, Y. (2012). Understanding the exploding gradient problem. *CoRR*, *abs/1211.5063*. <http://arxiv.org/abs/1211.5063>
- Radford, A., & Narasimhan, K. (2018). Improving language understanding by generative pre-training.
- Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). Language models are unsupervised multitask learners.
- Raff, E., Zak, R., Cox, R., Sylvester, J., Yacci, P., Ward, R., Tracy, A., Mclean, M., & Nicholas, C. (2018). An investigation of byte n-gram features for malware classification. *Journal of Computer Virology and Hacking Techniques*, *14*. <https://doi.org/10.1007/s11416-016-0283-1>
- Rathore, H., Agarwal, S., Sahay, S. K., & Sewak, M. (2019). Malware detection using machine learning and deep learning. *CoRR*, *abs/1904.02441*. <http://arxiv.org/abs/1904.02441>
- Richardson, R., & North, M. M. (2017). Ransomware: Evolution, mitigation and prevention". *Faculty Publications.*, (7). <https://digitalcommons.kennesaw.edu/facpubs/4276>
- Rong, X. (2014). Word2vec parameter learning explained. *CoRR*, *abs/1411.2738*. <http://arxiv.org/abs/1411.2738>
- Ruder, S. (2017). An overview of gradient descent optimization algorithms.
- Ruizendaal, R. (2017). *Deep learning #3: More on cnns & handling overfitting what are convolutions, max pooling and dropout?* <https://towardsdatascience.com/deep-learning-3-more-on-cnns-handling-overfitting-2bd5d99abe5d> (accessed: Feb 2020)
- Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533–536. <https://doi.org/10.1038/323533a0>
- Salloum, S., Alshurideh, D. M., Elnagar, A., & Shaalan, K. (2020). Machine learning and deep learning techniques for cybersecurity: A review, 50–57. [https://doi.org/10.1007/978-3-030-44289-7\\_5](https://doi.org/10.1007/978-3-030-44289-7_5)
- Salton, G., Wong, A., & Yang, C. S. (1975). A vector space model for automatic indexing. *Commun. ACM*, *18*(11), 613–620. <https://doi.org/10.1145/361219.361220>
- Sanh, V., Debut, L., Chaumond, J., & Wolf, T. (2019). Distilbert, a distilled version of BERT: smaller, faster, cheaper and lighter. *CoRR*, *abs/1910.01108*. <http://arxiv.org/abs/1910.01108>
- Santos, I., Brezo, F., Nieves, J., Peña, Y. K., Sanz, B., Laorden, C., & Bringas, P. G. (2010). Idea: Opcode-sequence-based malware detection. In F. Massacci, D. Wallach, & N. Zannone (Eds.), *Engineering secure software and systems* (pp. 35–43). Springer Berlin Heidelberg.

- Santos, I., Sanz, B., Laorden, C., Brezo, F., & Bringas, P. G. (2011). Opcode-sequence-based semi-supervised unknown malware detection (Á. Herrero & E. Corchado, Eds.), 50–57.
- Scarfone, K., & Souppaya, M. (2013). Guide to malware incident prevention and handling for desktops and laptops. *National Institute of Standards and Technology Special Publication 800-83 Revision 1*, 7(4), 1. <http://dx.doi.org/10.6028/NIST.SP.800-83r1>
- Shabtai, A., Moskovitch, R., Feher, C., Dolev, S., & Elovici, Y. (2012). Detecting unknown malicious code by applying classification techniques on opcode patterns. *Security Informatics*, 1(1), 1. <https://doi.org/10.1186/2190-8532-1-1>
- Shahmirzadi, O., Lugowski, A., & Younge, K. (2018). Text similarity in vector space models: A comparative study. *CoRR*, *abs/1810.00664*. <http://arxiv.org/abs/1810.00664>
- Sharma, R., Gupta, M., Agarwal, A., & Bhattacharyya, P. (2015). Adjective intensity and sentiment analysis, 2520–2526. <https://doi.org/10.18653/v1/D15-1300>
- Sherstinsky, A. (2018a). Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network. *CoRR*, *abs/1808.03314*.
- Sherstinsky, A. (2018b). Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network. *CoRR*, *abs/1808.03314*. <http://arxiv.org/abs/1808.03314>
- Siarni-Namini, S., Tavakoli, N., & Namin, A. S. (2019). The performance of lstm and bilstm in forecasting time series. *2019 IEEE International Conference on Big Data (Big Data)*, 3285–3292. <https://doi.org/10.1109/BigData47090.2019.9005997>
- Souri, A., & Hosseini, R. (2018). A state-of-the-art survey of malware detection approaches using data mining techniques. *Human-centric Computing and Information Sciences*, 8(1), 3. <https://doi.org/10.1186/s13673-018-0125-x>
- Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15, 1929–1958.
- Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., & Rabinovich, A. (2015). Going deeper with convolutions. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.
- Torfi, A., Shirvani, R. A., Keneshloo, Y., Tavaf, N., & Fox, E. A. (2020). Natural language processing advancements by deep learning: A survey. *CoRR*, *abs/2003.01200*. <https://arxiv.org/abs/2003.01200>
- Vasan, D., Alazab, M., Wassan, S., Safaei, B., & Zheng, Q. (2020). Image-based malware classification using ensemble of cnn architectures (imcec). *Computers & Security*, 92, 101748. <https://doi.org/10.1016/j.cose.2020.101748>

- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention is all you need. *CoRR*, *abs/1706.03762*. <http://arxiv.org/abs/1706.03762>
- Vinayakumar, R., Alazab, M., Soman, K. P., Poornachandran, P., & Venkatraman, S. (2021). Robust intelligent malware detection using deep learning," in *iee access*. *IEEE Access*, 7(5), 46717–46738.
- Yan, G., Brown, N., & Kong, D. (2013). Exploring discriminatory features for automated malware classification, 41–61. [https://doi.org/10.1007/978-3-642-39235-1\\_3](https://doi.org/10.1007/978-3-642-39235-1_3)
- Yoder, J. (2018). Determining optimum drop-out rate for neural networks, *The Bridge, The Magazine of IEEE-Eta Kappa Nu*, 115, 10–17. [http://micsymposium.org/mics2018/proceedings/MICS\\_2018\\_paper\\_27.pdf](http://micsymposium.org/mics2018/proceedings/MICS_2018_paper_27.pdf)
- Zeiler, M. D. (2012). ADADELTA: an adaptive learning rate method. *CoRR*, *abs/1212.5701*. <http://arxiv.org/abs/1212.5701>
- Zhang, H., Xiao, X., Mercaldo, F., Ni, S., Martinelli, F., & Sangaiah, A. K. (2019). Classification of ransomware families with machine learning based onn-gram of opcodes. *Future Generation Computer Systems*, 90, 211–221. <https://doi.org/https://doi.org/10.1016/j.future.2018.07.052>
- Zirn, C., Niepert, M., Stuckenschmidt, H., & Strube, M. (2011). Fine-grained sentiment analysis with structural features. *Proceedings of IJCNLP*.

## Appendix A

### RESERVED SECTIONS IN PE

Table A.1: Reserved Sections in PE Header

| Section Name | Content                          |
|--------------|----------------------------------|
| .bss         | Uninitialized data (free format) |
| .data        | Initialized data (free format)   |
| .edata       | Export tables                    |
| .idata       | Import tables                    |
| .pdata       | Exception information            |
| .rdata       | Read-only initialized data       |
| .reloc       | Image relocations                |
| .rsrc        | Resource directory               |
| .text        | Executable code (free format)    |

*Note.* Additionally debug(debug symbols, types), cormeta(managed code), tls(Threadlocal storage), idlsym (registerd seh).



## Appendix B

### TEXT VECTORIZATION

#### B.1 Sample Text Vector

The output is created using 400 for the `sequence_length` parameter, and 3-grams in `TextVectorization` class.

```
Vectorized opcode sequence (<tf.Tensor: shape=(1, 400), dtype=int64, numpy= array([[ 23, 13, 20, 16, 71, 8, 6, 3, 15, 10, 8, 6, 3, 4, 10, 8, 6, 3, 4, 4, 39, 10, 8, 6, 3, 4, 10, 17, 21, 5, 3, 4, 8, 6, 3, 4, 2, 33, 8, 6, 3, 4, 10, 8, 6, 3, 4, 2, 10, 8, 6, 3, 4, 10, 151, 5, 3, 4, 8, 6, 3, 4, 10, 14, 62, 20, 18, 7, 16, 2, 129, 2, 7, 4, 2, 14, 19, 14, 2, 56, 2, 107, 56, 2, 53, 5, 3, 22, 11, 25, 28, 6, 3, 15, 2, 29, 48, 25, 6, 3, 18, 9, 48, 25, 6, 3, 18, 9, 48, 25, 5, 3, 18, 9, 34, 2, 86, 19, 59, 3, 13, 2, 56, 2, 48, 25, 6, 3, 18, 9, 28, 6, 3, 12, 2, 40, 28, 6, 3, 11, 12, 58, 2, 10, 28, 6, 3, 13, 2, 40, 60, 50, 62, 31, 4, 2, 8, 6, 3, 4, 10, 8, 6, 3, 4, 10, 8, 6, 3, 4, 10, 37, 13, 4, 24, 2, 2, 118, 15, 39, 84, 2, 118, 15, 39, 84, 2, 45, 30, 29, 28, 11, 5, 3, 9, 15, 77, 2, 119, 28, 5, 3, 11, 16, 77, 2, 11, 27, 4, 2, 84, 2, 121, 2, 22, 7, 19, 2, 14, 16, 119, 28, 11, 5, 3, 12, 15, 77, 2, 27, 4, 2, 84, 2, 14, 16, 52, 19, 5, 3, 15, 2, 2, 14, 4, 21, 13, 8, 6, 3, 4, 10, 8, 6, 3, 4, 10, 8, 4, 5, 3, 4, 8, 5, 3, 11, 4, 68, 2, 9, 8, 6, 3, 4, 10, 8, 6, 3, 4, 10, 8, 10, 29, 8, 6, 3, 11, 32, 8, 5, 3, 15, 12, 8, 5, 3, 18, 2, 4, 8, 6, 3, 4, 40, 8, 5, 3, 4, 2, 8, 6, 3, 4, 10, 8, 6, 3, 4, 10, 8, 6, 3, 4, 41, 8, 6, 3, 4, 10, 8, 6, 3, 4, 41, 8, 6, 3, 4, 10, 8, 6, 3, 4, 2, 10, 21, 4, 8, 6, 3, 4, 10, 45, 6, 3, 4, 10, 8, 6, 3, 4, 10, 8, 10, 6, 3, 4, 8, 6, 3, 4, 4, 39, 10, 8, 6, 3, 4, 10, 8, 6, 3, 4, 10, 8, 6, 3, 4, 10, 8, 10, 2, 8, 6, 3, 4, 10, 8, 6, 3]])>, <tf.Tensor: shape=(), dtype=int32, numpy=1>)
```

TEZ İZİN FORMU / THESIS PERMISSION FORM

ENSTİTÜ / INSTITUTE

Fen Bilimleri Enstitüsü / Graduate School of Natural and Applied Sciences

☐

Sosyal Bilimler Enstitüsü / Graduate School of Social Sciences

☐

Uygulamalı Matematik Enstitüsü / Graduate School of Applied Mathematics

☐

Enformatik Enstitüsü / Graduate School of Informatics

☒

Deniz Bilimleri Enstitüsü / Graduate School of Marine Sciences

☐

YAZARIN / AUTHOR

Soyadı / Surname : Demirci

Adı / Name : Deniz

Bölümü / Department : Cyber Security

TEZİN ADI / TITLE OF THE THESIS (İngilizce / English) : Static Malware Detection using Stacked Bi-Directional LSTM

TEZİN TÜRÜ / DEGREE: Yüksek Lisans / Master ☒

Doktora / PhD ☐

1. Tezin tamamı dünya çapında erişime açılacaktır. / Release the entire work immediately for access worldwide. ☐
2. Tez iki yıl süreyle erişime kapalı olacaktır. / Secure the entire work for patent and/or proprietary purposes for a period of two year. \* ☐
3. Tez altı ay süreyle erişime kapalı olacaktır. / Secure the entire work for period of six months. \* ☐

\* Enstitü Yönetim Kurulu Kararının basılı kopyası tezle birlikte kütüphaneye teslim edilecektir.  
A copy of the Decision of the Institute Administrative Committee will be delivered to the library together with the printed thesis.

Yazarın imzası / Signature



Tarih / Date ..02.08.2021.....