Vulnerability Detection in Assembly Code Using Deep Learning

by

Karthiga Thangavelu

B.E., Sri Shakthi Institute of Engineering and Technology, India, 2014

A Project submitted in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF ENGINEERING

in the Department of Electrical and Computer Engineering

© Karthiga Thangavelu, 2022

University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopy or other means, without the permission of the author.

Vulnerability Detection in Assembly Code Using Deep Learning

by

Karthiga Thangavelu

B.E., Sri Shakthi Institute of Engineering and Technology, India, 2014

Supervisory Committee

Dr. Mihai SIMA, Supervisor

Department of Electrical and Computer Engineering

Dr. Chris Papadopoulos, Departmental Member

Department of Electrical and Computer Engineering

Abstract

Language modelling for source code is a state-of-the-art method which is developing significantly in recent years. Its applications are found in code completion, translating programming languages from one to another, translating text documents to code, finding vulnerabilities in source code, etc. Unlike other source code modelling such as C, C++ or Python, modelling assembly language is a tedious process. Most of the approaches involved in feature engineering are manual in assembly code. In this project, the pattern of assembly code is recognized, and malicious code is classified from non-malicious code. The strings of jumps are introduced into the assembly code to make it non-malicious. The pattern recognition and classification process consist of 3 main tasks. Firstly, the strings of jumps are introduced to the assembly code and tokenize the assembly code. Secondly, converting instructions to vectors using assembly language model for instruction embedding based on BERT language transformer, which minimizes the manual process of dataset pre-processing. The final task is a downstream task where the instruction embeddings are fed into the LSTM network for classifying malicious code from non-malicious code using an assembly code dataset. The performance of the model is evaluated using various evaluation metrics such as accuracy, confusion matrix, recall, precision, and F1 score.

Table of Contents

List of Tables

List of Figures

Acronym

NLP – Natural Language Processing

JMP – Jump Instruction

GCC – GNU Compiler Collection

BERT – Bidirectional Encoder Représentations from Transformer

PALMTREE – Pre-trained Assembly Language Model for Instruction Embedding

LSTM – Long Short-Term Memory

TF-IDF – Term Frequency – Inverse Document Frequency

CBOW – Continuous Bag of Words

PCA – Principal Component Analysis

OOV – Out-of-Vocabulary

MLM – Masked Language Model

CWP – Context Window Prediction

DUP – Def-Use Prediction

NSP – Next Sentence Prediction

RELU – Rectified Linear Unit

RNN – Recurrent Neural Network

SGD – Stochastic Gradient Descent

GPT – Generative Pre-trained Transformer

IDE – Integrated Development Environment

GPU – Graphics Processing Unit

RAM – Random Access Memory

CUDA – Compute Unified Device Architecture

Acknowledgement

I am grateful to God for helping me in my difficult times. I would like to express my gratitude to my supervisor Dr. Mihai SIMA for his patience, advice, encouragement, and continuous support throughout my studies and for believing in me, and for giving me an opportunity to do a wonderful project. My completion of the project would not be possible without his support. I also express my sincere gratitude to other members of my supervisory committee Dr. Chris Papadopoulos for his time being a part of the supervisory committee. I would like to offer special thanks to my parents and my husband for their motivation and support. Finally, I would like to express my thanks to the faculties at UVic for their continued support in co-op and courses throughout my university journey.

Dedication

I dedicate this work to my parents, teachers, sister, my husband, and in-laws for their prayers, affection, and support which motivated me to accomplish my dream.

Chapter 1

Introduction

Machine learning is a subfield of Artificial Intelligence that has the capability to imitate human behavior. It has wide applications from smart homes to autonomous vehicles. Machine learning model can decide, recommend, predict, and perform some actions based on observation of environments. It is used in various fields such as medical, surveillance, security, social media, agriculture, and manufacturing [1].

In recent years, the importance of security has increased in every organization. Firmware security is given equal importance to physical security and information security. Vulnerability to security risks has grown rapidly, and so have the countermeasures for the exploits as well. Lack of firmware security possess risks such as tampering with data, gaining access to computer, and interrupting the service by disabling the device. This could damage the organization's assets and cause revenue loss for the organization. Any vulnerability in firmware should be continuously monitored [2].

Machine learning in security is one of the state-of-the-art technologies that has been growing rapidly. Monitoring and alerting are some of the significant advancements in security using machine learning. Finding a vulnerability in low-level languages remains a challenging task. Program analysis is the traditional method for finding vulnerabilities in programs. For example, type checking, taint checking, monitoring, and control flow analysis are some programming analysis traditional features to check for common vulnerabilities. However, these traditional features lead to a high false positive rate and low accuracy. Application of machine learning in security, detects vulnerabilities earlier and terminates the attack. Not only does it terminate the attack but also lower the false positive rate compared to the traditional method. It also detects unknown vulnerabilities by analyzing the pattern and behavior of the code [2].

Many firmware security breaches exploit faults in the code. The faulty code is a backdoor to bypass the security in the firmware and gain control of the system. The bugs and flaws in low-level code are sometimes considered as malicious which can compromise the system. Any malicious behavior of the code is a pattern deviation from the normal code. The malicious code may also contain vulnerabilities such as buffer and stack overflow. A few vulnerabilities can be

generated inadvertently by a compiler. A compiler like GCC compiles programs written in a high-level language, such as C, C++ into assembly code. In this process, the compiler may inadvertently generate vulnerability in the resulting code [3].

The malicious code is differentiated from non-malicious code by the control flow and data flow of the code. If there is any code snippet that has a malicious behavior in its pattern, such as moving or copying value beyond the allocated memory (referred to as stack overflow) or calling a routine that is not part of the code snippet, then this is considered malicious code which possesses possible vulnerabilities. Analyzing the source code and finding behavior is an NLP (Natural Language Processing) task.

NLP is a field in machine learning, in which human language is converted into a machine-understandable format. Deep learning and machine learning techniques are used in NLP tasks such as language modeling, machine translation, sentiment analysis, chatbot, language translation, etc. In recent years, there is a significant development in language modeling for source code. Completing incomplete code, structuring code based on documentation, translating programming language from one to another, finding errors in the program, and auto-filling the code are some examples of programming language source code modeling. These features are integrated into different IDE for software or application development [4].

1.1 Overview

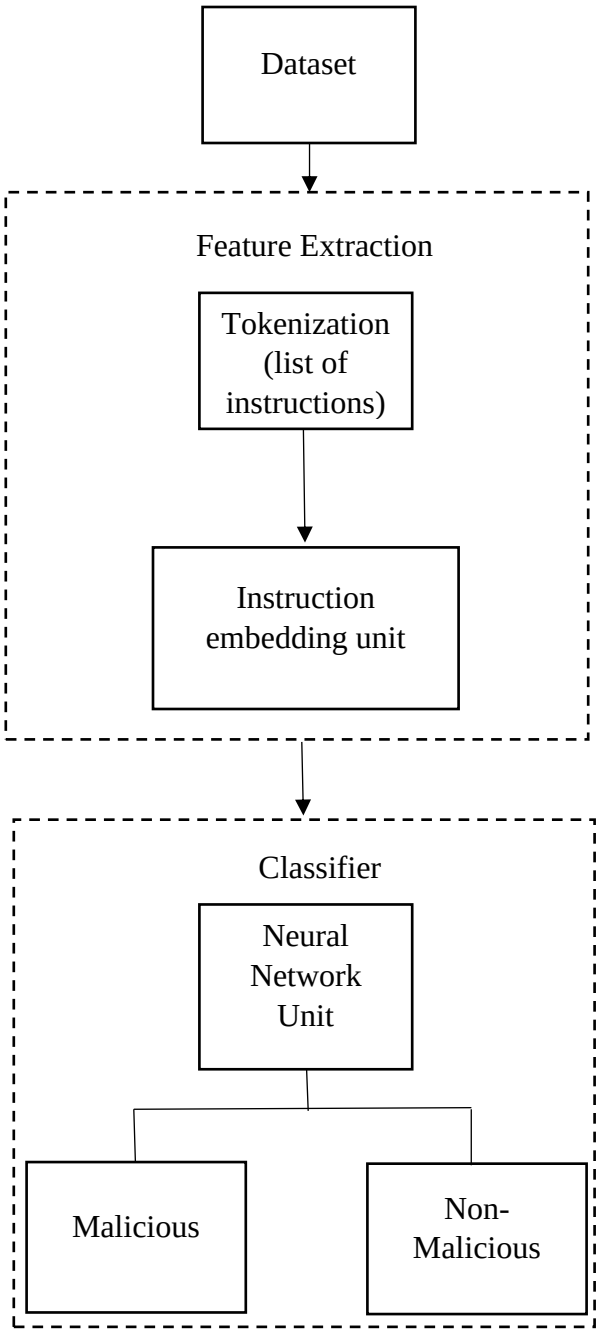

Figure 1.1. General Block Diagram of Vulnerability Detection classifier

```
_search:
LFB10:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        subl    $16, %esp
        movl    $0, -4(%ebp)
        jmp     L2
        jmp     L5
L5:
        movl    -4(%ebp), %eax
        leal    0(,%eax,4), %edx
        movl    12(%ebp), %eax
        addl    %edx, %eax
        movl    (%eax), %eax
        cmpl    8(%ebp), %eax
        jne     L3
        movl    $1, %eax
        jmp     L4
        jmp     L3
L3:
        addl    $1, -4(%ebp)
        jmp     L2
L2:
        movl    -4(%ebp), %eax
        cmpl    16(%ebp), %eax
```

Figure 1.2. Strings of jumps

['LC0:\n', '\t.ascii "Enter the 3 digit number :\\0"\n', 'LC1:\n', '\t.ascii "%d\\0"\n', 'LC2:\n', '\t.ascii "The number is Armstrong \\0"\n', 'LC3:\n', '\t.ascii "The number is Not Armstrong\\0"\n', '\t.text\n', '\t.globl\t_main\n', '\t.def\t_main;\t.scl\t2;\t.type\t32;\t.endef\n', '_main:\n', 'LFB10:\n', '\t.cfi_startproc\n', '\tpushl\t%ebp\n', '\t.cfi_def_cfa_offset 8\n', '\t.cfi_offset 5, -8\n', '\tmovl\t%esp, %ebp\n', '\t.cfi_def_cfa_register 5\n', '\tandl\t$-16, %esp\n', '\tsubl\t$32, %esp\n', '\tcall\t___main\n', '\tmovl\t$0, 28(%esp)\n', '\tmovl\t$LC0, (%esp)\n', '\tcall\t_printf\n', '\tleal\t16(%esp), %eax\n', '\tmovl\t%eax, 4(%esp)\n', '\tmovl\t$LC1, (%esp)\n', '\tcall\t_scanf\n', '\tmovl\t16(%esp), %eax\n', '\tmovl\t%eax, 24(%esp)\n', '\tmovl\t16(%esp), %ecx\n', '\tmovl\t$1717986919, %edx\n', '\tmovl\t%ecx, %eax\n', '\timull\t%edx\n', '\tsarl\t$2, %edx\n', '\tmovl\t%ecx, %eax\n', '\tsarl\t$31, %eax\n', '\tsubl\t%eax, %edx\n', '\tmovl\t%edx, %eax\n', '\tmovl\t%eax, 20(%esp)\n', '\tmovl\t20(%esp), %edx\n', '\tmovl\t%edx, %eax\n', '\tsall\t$2, %eax\n', '\taddl\t%edx, %eax\n', '\taddl\t%eax, %eax\n', '\tsubl\t%eax, %ecx\n', '\tmovl\t%ecx, %eax\n', '\tmovl\t%eax, 20(%esp)\n', '\tmovl\t20(%esp), %eax\n', '\timull\t20(%esp), %eax\n', '\timull\t20(%esp), %eax\n', '\taddl\t%eax, 28(%esp)\n', '\tmovl\t16(%esp), %ecx\n', '\tmovl\t$1717986919, %edx\n', '\tmovl\t%ecx, %eax\n', '\timull\t%edx\n', '\tsarl\t$2, %edx\n', '\tmovl\t%ecx, %eax\n', '\tsarl\t$31, %eax\n', '\tsubl\t%eax, %edx\n', '\tmovl\t%edx, %eax\n', '\tmovl\t%eax, 16(%esp)\n', '\tmovl\t16(%esp), %ecx\n', '\tmovl\t$1717986919, %edx\n', '\tmovl\t%ecx, %eax\n', '\timull\t%edx\n', '\tsarl\t$2, %edx\n', '\tmovl\t%ecx, %eax\n', '\tsarl\t$31, %eax\n', '\tsubl\t%eax, %edx\n', '\tmovl\t%edx, %eax\n',

Figure 1.3. Tokenization of assembly code

```
[-0.0109625   0.00470664  0.10213382  0.18005367 -0.1857076  -0.03184296
  0.0066714  -0.08295536 -0.15423976 -0.02961071  0.          0.
  0.          0.          0.          0.01138997  0.14885063 -0.0159678
 -0.05300715 -0.17498079  0.          0.          0.          0.
  0.          0.04931039 -0.01779061  0.11052511 -0.05487208  0.04529563]
0
[-0.0109625   0.00470664  0.10213382  0.18005367 -0.1857076  -0.03184296
  0.0066714  -0.08295536 -0.15423976 -0.02961071  0.          0.
  0.          0.          0.          0.01138997  0.14885063 -0.0159678
 -0.05300715 -0.17498079  0.          0.          0.          0.
```

Figure 1.4. Numerical representation of the assembly listing

Figure 1.1 shows the block diagram of vulnerability detection classifier. The block diagram has three units: Dataset, Feature Extraction, and Classifier. In this work, the dataset unit consists of assembly routines, which are separated into two classes: malicious and non-malicious. Strings of jumps are added to the assembly routines to make the assembly routines malicious as shown in Figure 1.2 in feature extraction unit. In this step, the malicious and non-malicious assembly routines are converted to assembly listing. The assembly listing is the tokenized list of codes as shown in Figure 1.3. The important features such as opcode, operand, and labels are extracted from the code. The assembly listing is then converted into vectors in the instruction embedding unit. The vectors are the numerical representation of the assembly list as shown in Figure 1.4. These vectors are fed into the classifier unit. The classifier unit consists of a neural network. The neural network classifies the malicious code from non-malicious code.

1.2 Problem Statement

In assembly routines, the unconditional jump is performed by JMP (jump) instruction. Jump instruction transfers the flow of execution to a different set of instructions or routines. In non-malicious code, the jump instruction transfers the control flow only to a routine that needs to be executed as the original code intended to do. If it transfers the control flow to a routine that is not part of the code and has any abnormal strings of jumps that executes the code in a loop, the code is considered to have malicious behavior. This execution of a malicious jump instruction can be exploited to execute some other routine that performs an action of tampering with data, monitoring the device activity, or denying the service by running the loop of instruction an infinite amount of times. The jump instruction doesn't directly perform harmful actions instead it hides the virus and when exploited it executes a code snippet that performs harmful actions or scatter the virus.

In this project, unusual strings of jumps in assembly source code are considered malicious. Deep learning models are used to recognize this pattern of jumps to classify malicious code from non-malicious code. The deep learning model learns the pattern of control flow and data flow in the source code. Similar to these unusual strings of jumps, there can be other malicious snippets in the assembly code.

The deep learning model learns the flow of operand and opcode in the assembly routines. The pattern recognition task having assembly code as input using deep learning has three parts. Firstly, the assembly source code for the dataset is collected. The assembly source code dataset is then divided into two datasets: malicious and non-malicious. The dataset has 515 assembly code files converted using GCC (GNU Compiler Collections) from C into assembly and a few Intel x86 and Intel 386 files taken from GitHub. The assembly code from Intel x86 and intel 386 is used to train the classifier to learn the structure of assembly code used in Intel processors. The non-malicious dataset is altered by introducing strings of jumps. The dataset is pre-processed to eliminate unwanted details such as comments, file name, and compiler details at the end of the code, and collect the important features such as operand and opcode in the code. Since most deep learning models accept only numerical data, the words are converted to vectors, which are a numerical representation of the assembly code. Chapter 2 and Chapter 3 explain the feature

extraction steps in detail, consisting of data pre-processing and instruction embedding (converting strings to its vector representation) based on BERT (Bidirectional Encoder Representations from Transformers). In this project, PALMTREE: Learning an Assembly Language Model for Instruction Embedding [5] is used for instruction embedding and henceforth this is referred as “Assembly language model for instruction embedding based on BERT” in this report. The BERT is a transformer-based pre-trained model for natural language processing developed by Google[6]. Secondly, in a downstream task, vectorized data are fed into the LSTM (Long Short-Term Memory) classifier for training the model, which is explained in detail in Chapter 4. Other deep learning models for the NLP task are also used in comparison for choosing the model that provides higher accuracy. Finally, the classifier is tested with a test dataset to classify malicious code. Using evaluation metrics like accuracy, confusion matrix, F1 score, precision, and recall the performance of the model is determined. Evaluation metrics, performance, and other existing approaches are discussed in Chapter 5 and Chapter 6. Chapter 7 gives an overview of the software and libraries used to accomplish the task. The conclusion and future work are explained in Chapter 8.

Chapter 2

Overview of Dataset

2.1. Assembly Code Structure

Assembly code is a low-level language that communicates directly with hardware computer devices. It is the bridge between hardware and high-level languages. The structure of assembly routines has a set of instructions. The instruction has one opcode and one or two operands. An opcode is an action to be performed such as moving an element from one register to another - (MOV), adding two values in registers (ADD), comparing two operands (CMP), jumping to execute a new set of routines (JMP), etc., Operands are registers and/or memory locations where the instruction is executed. Operand name varies from processor to processor. Labels are names to subroutines which can be used with a jump instruction to execute the subroutine [7].

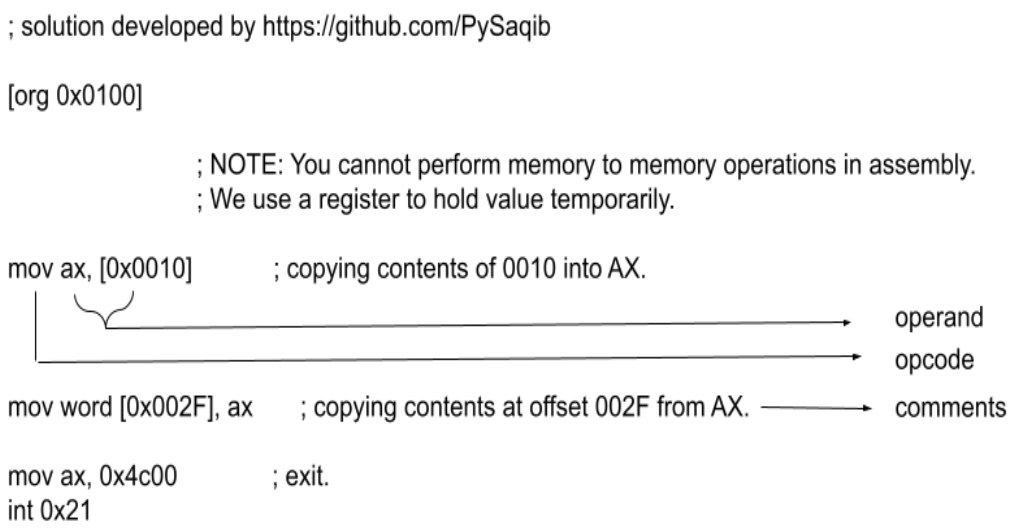

Figure 2.1. Pictorial representation of Intel x86 assembly code

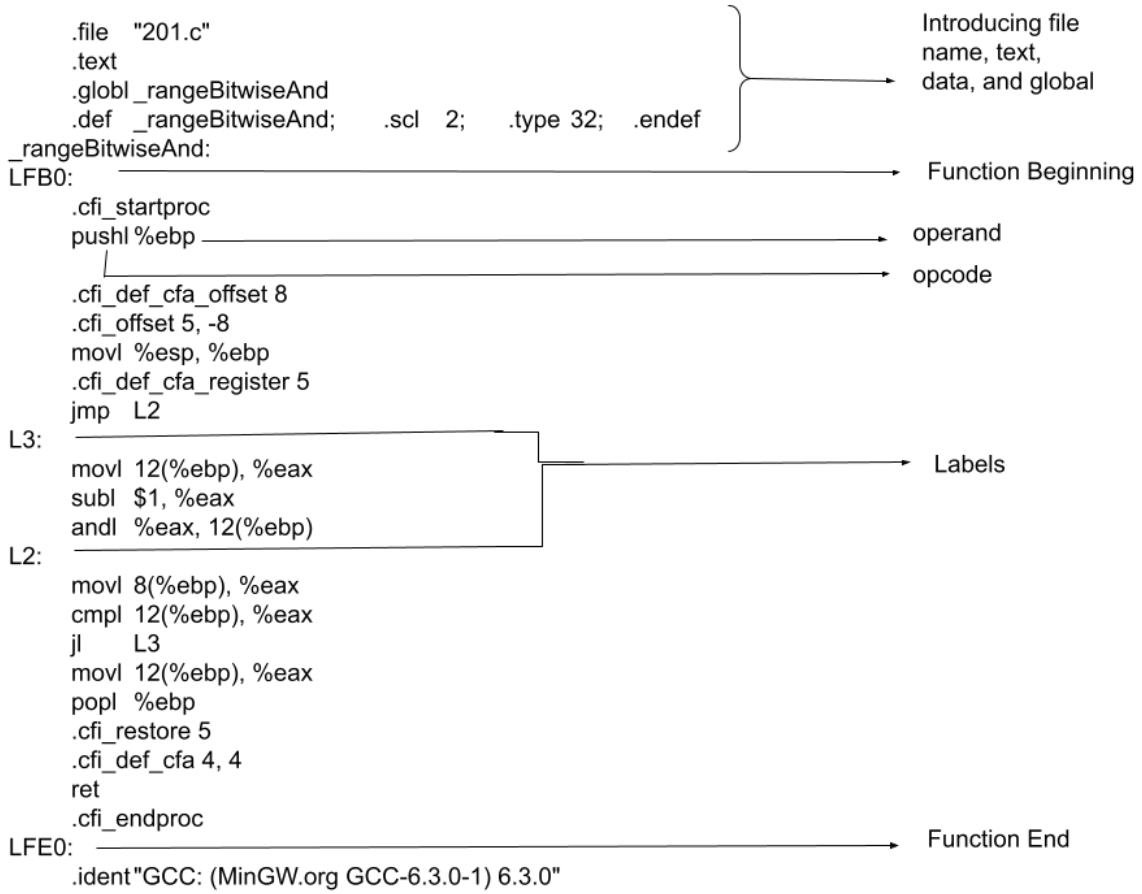

Figure 2.2. Pictorial representation of converted assembly code from C using GCC compiler

The above Figure 2.1 and Figure 2.2 shows the pictorial representation of intel x86 assembly code and converted assembly code from C using GCC compiler. The assembly code are converted using GCC in local system.

2.2 Assembly Code Dataset

Collection of x86 and x86 series i386 (Intel 80386) assembly code are used as datasets. In addition to this, various C source codes from GitHub are converted into assembly using the GCC compiler. Since the pre-trained assembly language model for instruction embedding [5] is trained with x86 assembly code, the downstream task is also limited to the x86 assembly code. The collection of the assembly code is divided into malicious and non-malicious datasets. The size of the dataset is 1.8 MB which is a mixture of x86, i386, and assembly code converted using GCC. In total, 515 files have been used. In this, 265 files are considered malicious, and 250 files are

considered non-malicious files. At end of the cleaning process, 80,686 lines of assembly code are fed to the network for training and testing.

Chapter 3

Feature Extraction

3.1 Pre-Processing Dataset

Malicious and non-malicious x86 series assembly code file and assembly code that are converted using GCC compiler from C code are processed separately. The structure of x86 and i386 is different from GCC converted assembly code. C code, x86, and i386 files are taken directly from GitHub. The structure of x86 code is shown in figure 2.1 and C code that are converted to assembly code are shown in figure 2.2. The C code that are converted to assembly code using GCC compiler are shown in Table 3.1 in non-malicious assembly code column. The comma between each operand is removed and separated by space. As a proof of concept, the assembly code is made malicious by injecting strings of jumps which is shown in Table 3.1 in malicious column. Similarly, there are various other ways to make an assembly code malicious. The jump instruction is added at end of each subroutine which is represented as jump opcode ‘jmp’ followed by the label of the next subroutines in the assembly code. For which the labels in the code are collected and joined to jump opcode ‘jmp’ which branch out to that subroutine which is shown in Figure 3.1. Both malicious and non-malicious assembly code is tokenized into a list of instructions which is shown in Figure 3.3. In non-malicious, the assembly code is tokenized into a list of instructions without adding strings of jumps.

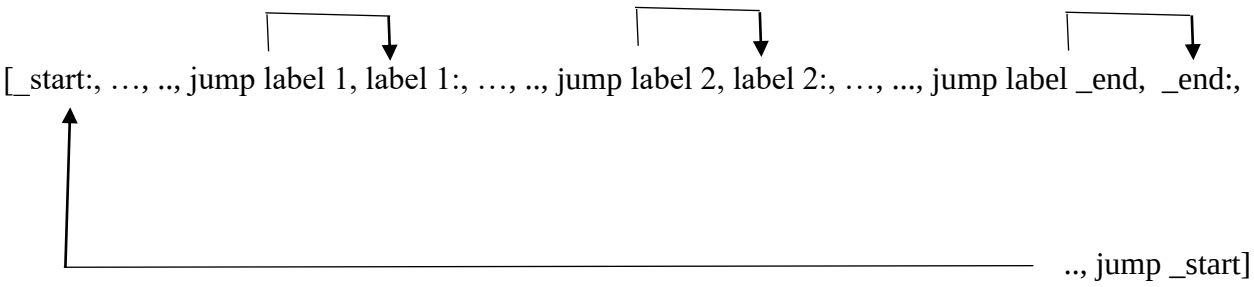

Figure 3.1. Malicious list of instructions

Figure 3.1 shows that the jump is added along with label from the next routine. Now, the previous jumps point to the following sub-routine of the assembly code, thus forming a loop.

The pictorial representation of pre-processing steps is given below.

Figure 3.2 shows the C code for recursion in a C code dataset. These C codes are converted to assembly code.

```c
// Sum of natural numbers using recursion

#include <stdio.h>
int sum(int n);

int main() {
  int number, result;

  printf("Enter a positive integer: ");
  scanf("%d", & number);

  result = sum(number);

  printf("sum = %d", result);
  return 0;
}

int sum(int num) {
  if (num != 0)
    return num + sum(num - 1); // sum() function calls itself
  else
    return num;
}
```

Figure 3.2. C code for recursion

Table 3.1 shows malicious and non-malicious recursion assembly code that are converted from C code using GCC compiler. The non-malicious code is changed to malicious code by adding string of jumps to non-malicious assembly code. Then, the code is tokenized by adding commas between each line of the code.

Non-Malicious Assembly Code	Malicious Assembly Code
.file “Recursion.c” .def ___main; .scl 2; .type 32; .endef .section .rdata,”dr” LC0: .ascii “Enter a positive integer: \0” LC1: .ascii “%d\0” LC2: .ascii “sum = %d\0” .text .globl _main .def _main; .scl 2; .type 32; .endef _main: LFB10:	.file "Recursion.c" .def ___main; .scl 2; .type 32; .endef .section .rdata,"dr" LC0: .ascii "Enter a positive integer: \0" LC1: .ascii "%d\0" LC2: .ascii "sum = %d\0" .text .globl _main .def _main; .scl 2; .type 32; .endef _main: LFB10:

```
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        andl    $-16, %esp
        subl    $32, %esp
        call    ___main
        movl    $LC0, (%esp)
        call    _printf
        leal    24(%esp), %eax
        movl    %eax, 4(%esp)
        movl    $LC1, (%esp)
        call    _scanf
        movl    24(%esp), %eax
        movl    %eax, (%esp)
        call    _sum
        movl    %eax, 28(%esp)
        movl    28(%esp), %eax
        movl    %eax, 4(%esp)
        movl    $LC2, (%esp)
        call    _printf
        movl    $0, %eax
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
LFE10:
        .globl _sum
        .def _sum; .scl 2; .type 32; .endef
_sum:
LFB11:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        subl    $24, %esp
        cmpl    $0, 8(%ebp)
        je      L4
        movl    8(%ebp), %eax
        subl    $1, %eax
        movl    %eax, (%esp)
        call    _sum
        movl    %eax, %edx
        movl    8(%ebp), %eax
        addl    %edx, %eax
        jmp     L5
L4:
        movl    8(%ebp), %eax
L5:
        leave
```

```
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        andl    $-16, %esp
        subl    $32, %esp
        call    ___main
        movl    $LC0, (%esp)
        call    _printf
        leal    24(%esp), %eax
        movl    %eax, 4(%esp)
        movl    $LC1, (%esp)
        call    _scanf
        movl    24(%esp), %eax
        movl    %eax, (%esp)
        call    _sum
        movl    %eax, 28(%esp)
        movl    28(%esp), %eax
        movl    %eax, 4(%esp)
        movl    $LC2, (%esp)
        call    _printf
        movl    $0, %eax
        jmp     _sum    ———   added strings of jump
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc

LFE10:
        .globl _sum
        .def _sum; .scl 2; .type 32; .endef
_sum:
LFB11:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        subl    $24, %esp
        cmpl    $0, 8(%ebp)
        je      L4
        movl    8(%ebp), %eax
        subl    $1, %eax
        movl    %eax, (%esp)
        call    _sum
        movl    %eax, %edx
        movl    8(%ebp), %eax
        addl    %edx, %eax
        jmp     L5
        jmp     L4    ———    added string of jump
```

.cfi_restore 5 .cfi_def_cfa 4, 4 ret .cfi_endproc LFE11: .ident “GCC: (MinGW.org GCC-6.3.0-1) 6.3.0” .def _printf; .scl 2; .type 32; .endef .def _scanf; .scl 2; .type 32; .endef	L4: movl 8(%ebp), %eax jmp L5 ——— added strings of jump L5: jmp _main ——— added strings of jump leave .cfi_restore 5 .cfi_def_cfa 4, 4 ret .cfi_endproc LFE11: .ident "GCC: (MinGW.org GCC-6.3.0-1) 6.3.0" .def _printf; .scl 2; .type 32; .endef .def _scanf; .scl 2; .type 32; .endef

Table 3.1 Malicious and non-malicious assembly code

Below Figure 3.3 shows the non-malicious recursion assembly code that are tokenized into list of instructions. In tokenization step, the non-malicious code is converted into list of instructions. Similarly, malicious code is also tokenized into list of instructions which is shown below.

[['LC0:\n', '\t.ascii "Enter a positive integer: \\0"\n', 'LC1:\n', '\t.ascii "%d\\0"\n', 'LC2:\n', '\t.ascii "sum = %d\\0"\n', '\t.text\n', '\t.globl\t_main\n', '\t.def\t_main;\t.scl\t2;\t.type\t32;\t.endef\n', '_main:\n', 'LFB10:\n', '\t.cfi_startproc\n', '\tpushl\t%ebp\n', '\t.cfi_def_cfa_offset 8\n', '\t.cfi_offset 5, -8\n', '\tmovl\t%esp, %ebp\n', '\t.cfi_def_cfa_register 5\n', '\tandl\t$-16, %esp\n', '\tsubl\t$32, %esp\n', '\tcall\t___main\n', '\tmovl\t$LC0, (%esp)\n', '\tcall\t_printf\n', '\tleal\t24(%esp), %eax\n', '\tmovl\t%eax, 4(%esp)\n', '\tmovl\t$LC1, (%esp)\n', '\tcall\t_scanf\n', '\tmovl\t24(%esp), %eax\n', '\tmovl\t%eax, (%esp)\n', '\tcall\t_sum\n', '\tmovl\t%eax, 28(%esp)\n', '\tmovl\t28(%esp), %eax\n', '\tmovl\t%eax, 4(%esp)\n', '\tmovl\t$LC2, (%esp)\n', '\tcall\t_printf\n', '\tmovl\t$0, %eax\n', '\tleave\n', '\t.cfi_restore 5\n', '\t.cfi_def_cfa 4, 4\n', '\tret\n', '\t.cfi_endproc\n', '_sum\n', 'LFB11:\n', '\t.cfi_startproc\n', '\tpushl\t%ebp\n', '\t.cfi_def_cfa_offset 8\n', '\t.cfi_offset 5, -8\n', '\tmovl\t%esp, %ebp\n', '\t.cfi_def_cfa_register 5\n', '\tsubl\t$24, %esp\n', '\tcmpl\t$0, 8(%ebp)\n', '\tje\tL4\n', '\tmovl\t8(%ebp), %eax\n', '\tsubl\t$1, %eax\n', '\tmovl\t%eax, (%esp)\n', '\tcall\t_sum\n', '\tmovl\t%eax, %edx\n', '\tmovl\t8(%ebp), %eax\n', '\taddl\t%edx, %eax\n', '\tjmp\tL5\n', 'L4:\n', '\tmovl\t8(%ebp), %eax\n', 'L5:\n', '\tleave\n', '\t.cfi_restore 5\n', '\t.cfi_def_cfa 4, 4\n', '\tret\n', '\t.cfi_endproc\n']]

Figure 3.3. Non-malicious recursion assembly code tokenized into list of instructions

Below Figure 3.4 shows the vector representation of tokenized recursion assebmly code. These vectors are used as an input for classifier.

```
[[ 2.1986115  -0.57662386  0.809586   ...  3.3328645   2.0614183
  -2.3996646 ]
 [ 2.6054008  -0.30241543  0.95139503 ...  3.9425962   0.9281082
  -2.2762554 ]
 [ 1.765189    1.8456701   1.6342763  ...  4.628826    0.7051365
  -0.97854215]
 ...
 [ 1.5926443  -0.5184817   0.8298852  ...  3.0147703   0.59990114
  -1.3438734 ]
 [ 2.7522426  -0.9092636   1.2035706  ...  4.2560267   1.1782715
  -2.9976425 ]
 [ 1.7180493  -0.8862034   0.02915816 ...  2.1300073   1.1584195
  -1.7494713 ]]
```

Figure 3.4. Vector representation of recursion assembly code

In data pre-processing or cleaning, the comment for each instruction is removed. Also, in assembly code converted using GCC, the information before label ’LC0’ such as the text section and end of the file after label ‘LFE’ are removed. In the x86 processor code, lines before the main code starts, are deleted. The jump is added in the middle of the subroutine and at the end of the subroutine alternatively in x86. The list of tokenized assembly code from x86 and GCC converted assembly code, are merged and converted to a pickle file to use in training the network.

3.2 Word Embedding

The word embedding technique represents the word in real-values vectors. Below Figure 3.5 shows that the assembly code is represented in vector values using word embedding technique. Each word is represented in high-dimension vectors [8]. One-hot coding, TF-IDF, and Word2Vec are some popular word embedding techniques. The word embedding is then used as an initialization for classification in the downstream task which is mentioned in Figure 1.1. TF-IDF, Word2Vec, and assembly language model for instruction embedding based on BERT are discussed in this section along with the disadvantages of Word2Vec word embedding. In general, word embedding is used to represent words in vectors which are mainly used in tasks such as sentiment analysis, sentence completion and text classification but instruction embedding are particularly used for instruction level representation.

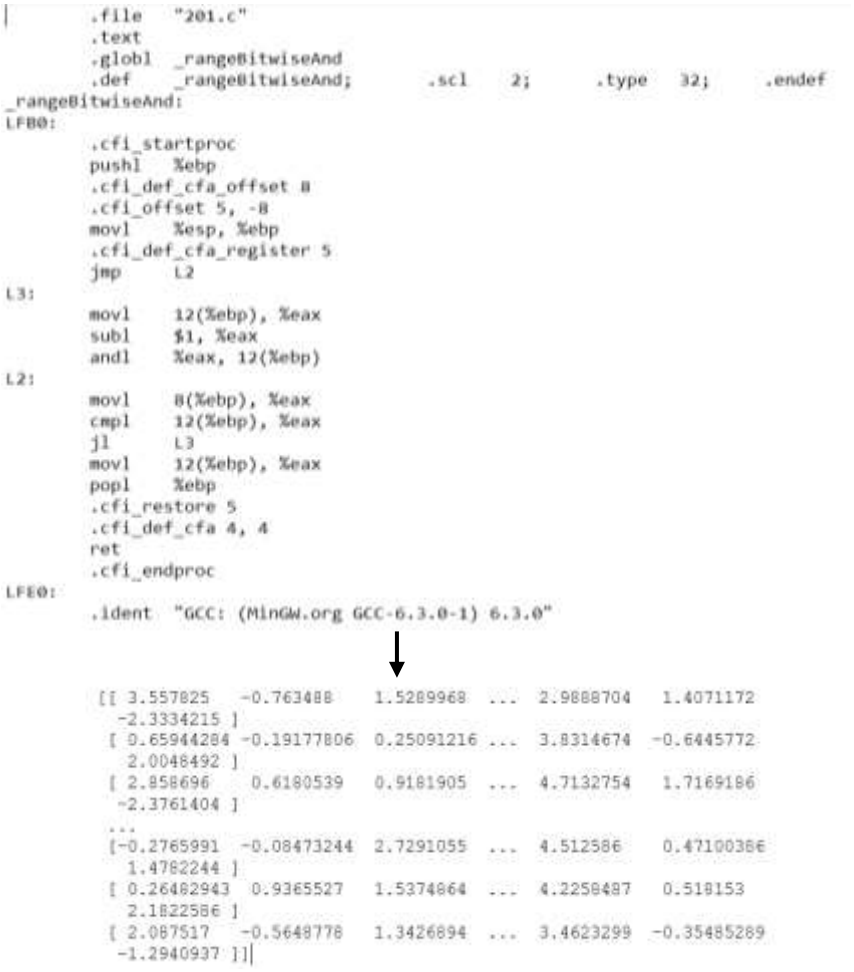

Figure 3.5. Converting assembly code to vectors

3.2.1. Term Frequency-Inverse Document Frequency

Term Frequency-Inverse Document Frequency or TF-IDF is a traditional algorithm for transforming text into vectors based on the frequency of the words. TF-IDF is combination of two different terms TF and IDF:

3.2.1.1. Term Frequency

Term frequency or TF refers to the number of occurrences of a specific word in a document. It is calculated as the ratio between the number of times a word occurred in the document and the total number of words in a document.

TF (w, d) = No of occurrence of a word (w) / total number of words in document (d)

Where w is the specific word in document d.

3.2.1.2. Inverse Document Frequency

Inverse document frequency or IDF which is the measure of the frequency of a word in the total number of documents. It is calculated as the ratio between the number of documents containing a specific word (w) which is represented as n to the number of documents (D), and it is normalized using a log. It is formulated as,

IDF(w) = log (number of document with word (n) / total number of documents (D))

where D is the total number of documents.

The TF-IDF score for each word in a document is the product of term frequency and inverse document frequency. It is formulated as follows,

TF-IDF(w,d) = TF (w,d) * IDF (w)

TF-IDF score increases with an increase in the number of words in a document and decreases with the number of documents containing that specific word increases in corpus [9].

3.2.2 Word2Vec

Word2Vec is another popular word embedding technique that uses two different learning models CBOW (Continuous Bag of Words) and Skip-Gram models. These models capture the semantics of the word sequence. Word2Vec model is an unsupervised learning process with unlabeled data which generates the word vectors. In this technique, the vector size is selected based on the downstream task and corpus size. The corpus trained on the Word2Vec model captures the closest value to each word in the corpus in terms of cosine similarity [8].

Cosine similarity is a metric to determine the similarity of two documents irrespective of their size. It measures the cosine angle between two vectors projected in multi-dimensional space. Smaller the angle between two vectors higher the cosine similarity. Let x and y be the two vectors, then the cosine similarity is given by,

$$cos\theta = \frac{x.y}{||x||\,||y||}$$

The Figure 3.6 is an example to illustrate cosine similarities in word2vec using countries and capitals. It is the two-dimensional PCA projection of the 1000-dimensional vectors of countries and capitals. Principal Component Analysis (PCA) is a technique to reduce the dimensionality of dataset while minimizing the information loss.

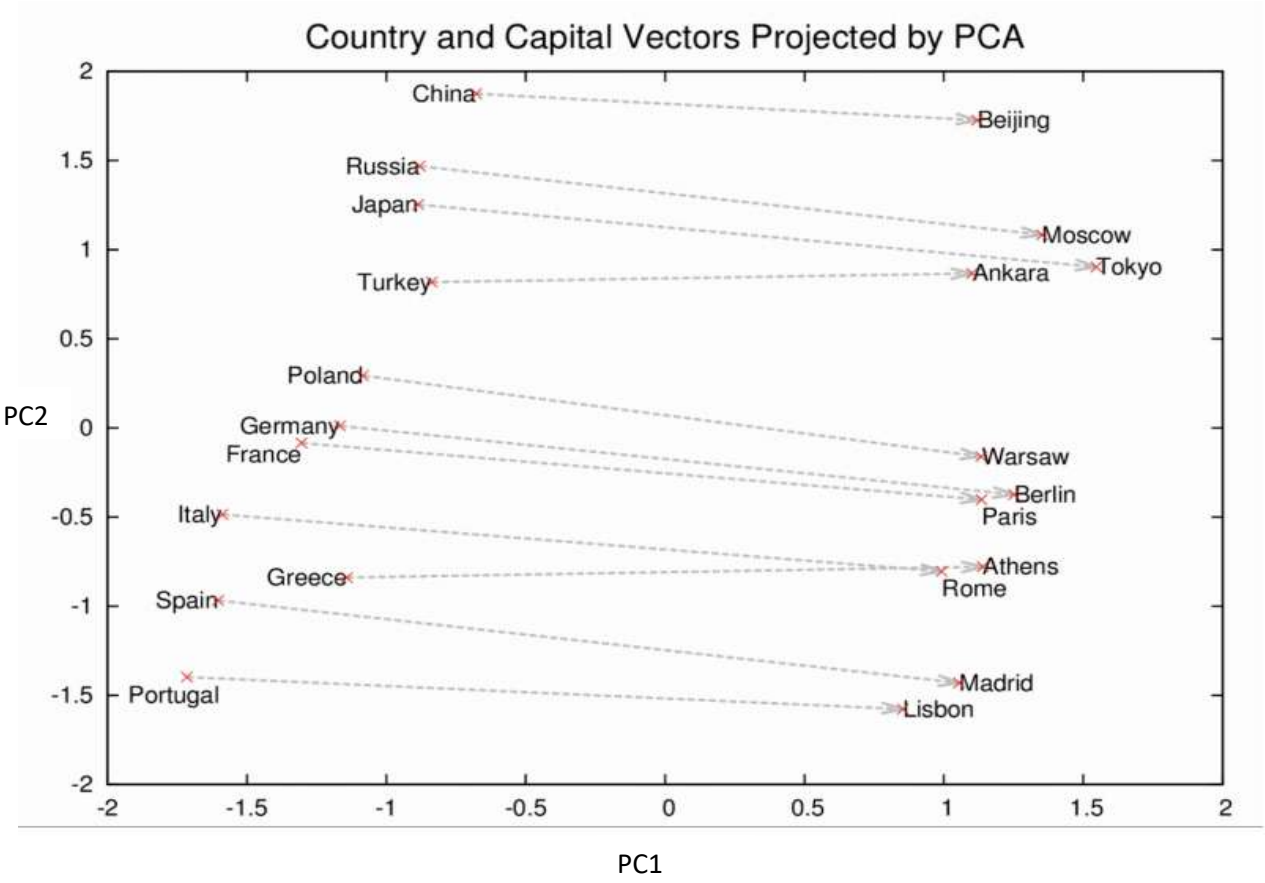

Figure 3.6. Illustrate the two-dimensional PCA projection of country and its capital [10].

From Figure 3.6, the vector distance between China and Beijing is similar to the vector distance between France and Paris in vector space, and likewise for other countries and capital. The equation for learning countries and their capital reference from other known countries and capital is given as Paris – France + Italy = Rome, this explains that subtracting the vector of Paris (capital) and France (country) and then adding Italy (country) is equal to the vector of Rome (capital). Therefore, learning that France's capital is Paris, the model will know that it has to find the capital of countries in a cluster of cities [10].

3.2.2.1. CBOW

Continuous Bag of Word predicts the targeted word by understanding the context of words surrounding the targeted word. Context window size decides how many surrounding words are to be considered for predicting the targeted word. If the context window size is 2 then it takes two surrounding words along with the target word. The objective of CBOW is to maximize the probability of:

$$1/n \ (\log p \ (w_t | \ w_{t+j}))$$

3.2.2.1.1. The Model Architecture

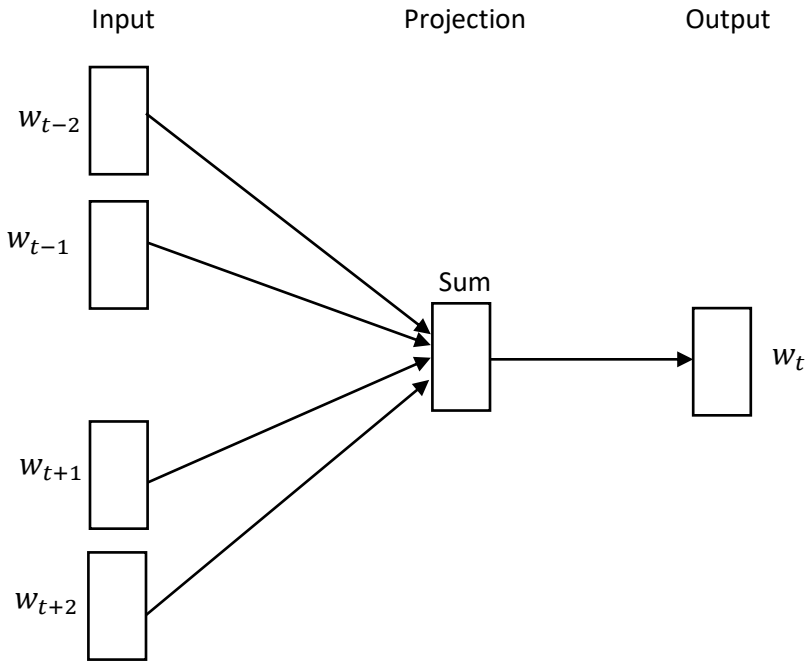

Figure 3.7. CBOW architecture - takes sequences of words and predict targeted output [11].

3.2.2.2. Skip-Gram

Skip-gram is a reverse model of the CBOW model. In this, each targeted word is used as input, and words are predicted within a certain range before and after the targeted word. The targeted word is mapped to the hidden layer and the output vector from the hidden layer is fed to the output layer to find the context of the targeted word. The objective of the Skip-gram model is to maximize the log probability of:

$$1/n \ (\log p(w_{t+j}|w_t))$$

3.2.2.2.1. The Model Architecture

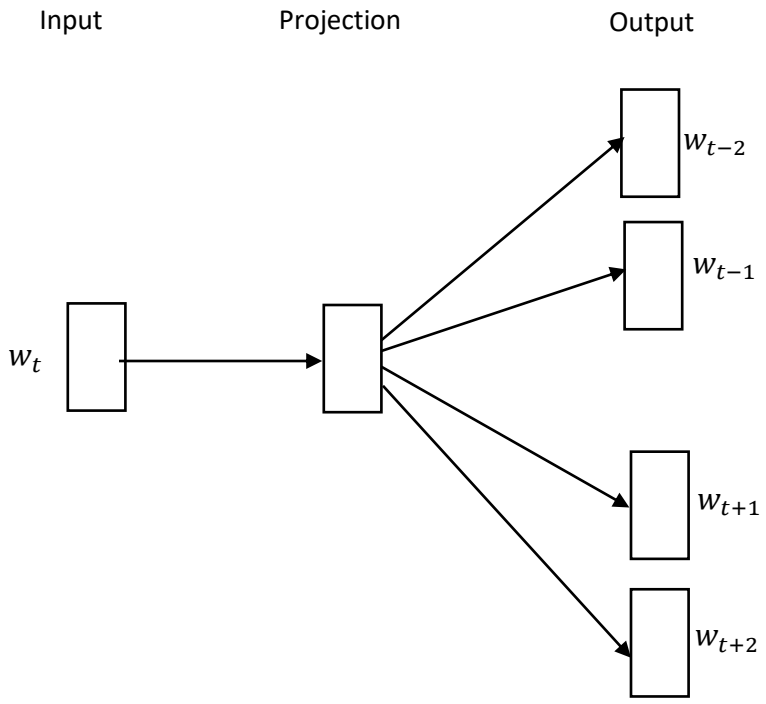

Figure 3.8. Skipgram architecture – that takes targeted word as input and predicts the context related to targeted word [11].

3.2.3. Disadvantages of TF-IDF and word2vec

The drawback of TF-IDF is if a particular word exists in all the documents, then the TF-IDF value will be zero. This would consider a word not uniquely important even though the term is uniquely essential, it would be considered as not uniquely important [12]. The disadvantage of word2vec is its inability to handle unknown or out-of-vocabulary words. It also requires a large corpus to train the model. The word2vec model does not share the parameters, meaning new languages cannot share the models trained with some other language. Comparing CBOW and skip-gram, CBOW is faster than skip-gram since CBOW is trained to predict targeted words from a fixed window size of context words. The disadvantage of CBOW and skip-gram, they fail to capture the combined word phrases. For example, ‘New York is a single word but CBOW and skip-gram treat them as two different words ‘New’ and ‘York’.

3.2.4. Assembly Language Model for Instruction Embedding based on BERT

Learning-based encoding is challenging for assembly code. To encode instruction, the model should consider control flow and implicit internal structure of instruction. Word2Vec, TF-IDF works only on control flow and ignores important internal structure. For example, the arithmetic operation changes the EFLAGS implicitly where these implicit changes are not tracked by word embedding technique. In this project, a pre-trained model for instruction embedding based on PALMTREE: Learning an Assembly Language Model for Instruction Embedding [5] imported from GitHub has been used. The pre-trained assembly language model generates instruction embedding which can be used in downstream tasks. The model is trained on a large-scale unlabeled assembly code corpus. In this approach, each instruction is considered a sentence. The pre-trained assembly model consists of three main tasks to understand the internal structure of instruction based on BERT. Those are Masked Language Model (MLM), Context Window Prediction (CWP), and Def-Use Prediction (DUP). These tasks are used in training the BERT model.

3.2.4.1. Tokenization

Each instruction result from pre-processing step is considered a sentence. The start of the sequence is identified using the special token [CLS] and the pair of instructions is separated by another special token [SEP]. Tokenization is a fine process that tokenizes each word in the sentence such as ‘mov’, ‘ax’, ‘dx’. In addition to this OOV (Out-of-Vocabulary) caused by string and constant is normalized. The strings are replaced with a special token [str] and larger length constant value are replaced with a special token [addr]. The smaller length constants are encoded as a one-hot vector. They are considered as significant information that is accessed [5].

3.2.4.2. BERT

Assembly language model for instruction embedding is based on the BERT (Bidirectional Encoder Representations from Transformer) model. BERT is a Bidirectional Encoder Representation from Transformers, a deep learning model based on transformers. Transformer architecture uses an attention mechanism to transform a given sequence of sentences into another sequence. The attention-mechanism technique has an encoder and decoder where the encoder

takes the input as a sequence of sentences and the decoder gives the output sequence which can be language, symbols, etc., In this project, the output is vectors of a sequence of words. The BERT model uses only the encoder part of the transformer and the input to the encoder is sequentially bidirectional which means the encoder reads the entire sequence of words input sequentially from left to right or right to left at once. The BERT model uses two strategies to train the model. One is MLM and the other is CWP [14, 15].

3.2.4.2.1. MLM

In MLM, 15% of the input instruction is randomly chosen to replace. Of the chosen tokens, 80% are masked by masked out tokens [MASK], 10% are corrupted tokens that are replaced with vocabulary, and the remaining 10% of the tokens are unchanged. The transformer predicts the masked-out and corrupted tokens. The probability of predicting tokens that are masked is given by:

$$P(\widehat{t_i}|I) = \frac{\exp\,(w_i\theta(I)_i)}{\sum_{k=1}^{K}\exp\,(w_i\theta(I)_i)}$$

The masked token is represented as t_i and prediction is represented as $\widehat{t_i}$. I is input where I = $t_1, t_2, t_3, \ldots\ t_n$. Where the above equation gives the probability of prediction from given input I. w is the weight of labels i and K are possible labels of the token t_i.

The model is trained with cross-entropy model which is given as:

$$L_{MLM} = \sum_{t_i \in m(I)} \log p(\hat{t}|I)$$

The masked out token and the corrupted token are considered in calculating the loss function [5].

Figure 3.9. Masked Language Model [5]

Figure 3.9 illustrates that there are masked tokens and corrupted tokens. ‘ebx’ and ‘rbx’ are the correct predictions of those masked and corrupted tokens.

3.2.4.2.2 CWP

In the BERT training process, NSP (Next Sentence Prediction) is the second task. The model receives a pair of sentences (instructions) as an input and predicts if the second sentence is subsequent to the first sentence in pair of sentences. In this process, 50% of the second sentences are chosen from a random sentence from the corpus [13].

In training assembly language model for instruction embedding based on BERT, the NSP process is replaced by CWP (Context Window Prediction). CWP process captures control flow information and understands contextual relations of the instruction sequence. The prediction occurs within a context window which is easier than predicting in a whole sentence. Therefore, NSP may not be suitable for capturing contextual information of control flow which is replaced with Context Window Prediction [5].

In Context Windows Prediction, the window size is considered as w, if w = 2 then, each instruction before 2 steps and after 2 steps of the targeted instructions are considered as contextually related. The probability of finding whether target instruction is contextually related or not is given by:

$$p(\hat{y}|I, I_C) = \frac{1}{1 + \exp\left(\theta(I||I_C)_{cls}\right)}$$

$\hat{y}$ denotes the prediction, I is the instruction, and I_C is the candidate instruction within instruction I. The prediction is the probability that candidate instruction I_C present in I

The loss function is given as:

$$L_{CWP} = -\sum_{I \in D} \log p(\hat{y}|I, I_C)$$

Where D is the dataset.

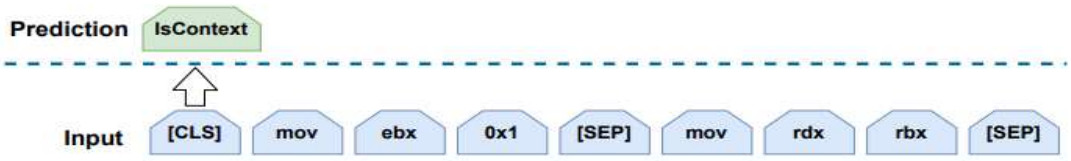

Figure 3.10. Context Window Prediction [5]

Figure 3.10 illustrate whether the token belongs to the context instructions or not [5].

3.2.4.3.2 DUP

DUP (Def-Use Prediction) which is originally not a part of the BERT training process but used in training the assembly language model for instruction embedding to improve the quality of instruction embedding. DUP is based on SOP (Sentence Order Prediction) which understands the data relation. Considering two pairs of instructions I1 and I2 and two instructions are swapped which are denoted as positive or otherwise negative. The probability the instruction pairs are swapped or not is given by:

$$p(\hat{y}|I_1, I_2) = \frac{1}{1 + \exp\left(\theta(I_1||I_2)_{cls}\right)}$$

$\hat{y}$ is the prediction. $I_1\ and\ I_2$ are instruction pair. $I_1||I_2$ is a positive sample.

The Cross Entropy loss function is given as:

$$L_{DUP} = -\sum_{I \in D} p(\hat{y}|I_1, I_2)$$

Where D is the dataset.

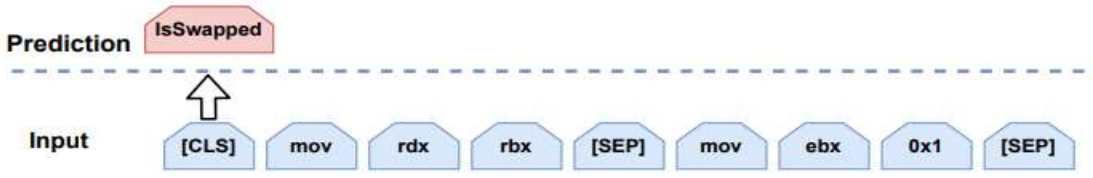

Figure 3.11. Def-Use Prediction [5]

Figure 3.11 illustrates that the instruction pair are swapped. The original flow of instruction is "CLS mov ebx 0x1 [SEP] mov rdx rbx [SEP]". The swapped instruction is fed into the network whose prediction will be swapped for the swapped pair of instructions [5].

The total loss of all three tasks is given by:

$$L = L_{MLM} + L_{CWP} + L_{DUP}$$

Following these 3 main tasks, the pre-trained assembly language model for instruction embedding is trained with different embedding size of 64, 128, 256, and 512. However, for the pre-trained model embedding size 128 is set as default.

Chapter 4

Downstream Task

4.1 Neural network

Neural Network or Artificial Neural Network is a subset of machine learning and is a set of algorithms that resemble the working of the human brain. The pattern neural networks recognize is numerical in vectors. They can also extract features in data automatically. Neural networks are used to solve both classification and clustering tasks. In the NLP task, extracting features in the language is crucial. Neural networks can extract the features automatically improving the performance of the model. Especially in recognizing the patterns in assembly, the neural network can minimize classification errors.

The basic architecture of a neural network consists of 3 layers of nodes. An input layer, one or more hidden layers, and an output layer. Each node in one layer is connected to another layer and carries weights and thresholds. The output is computed as the sum of weighted inputs with units. The output state is either active or inactive. If the output computed exceeds the threshold, the neuron is said to be active otherwise it is inactive [14, 15].

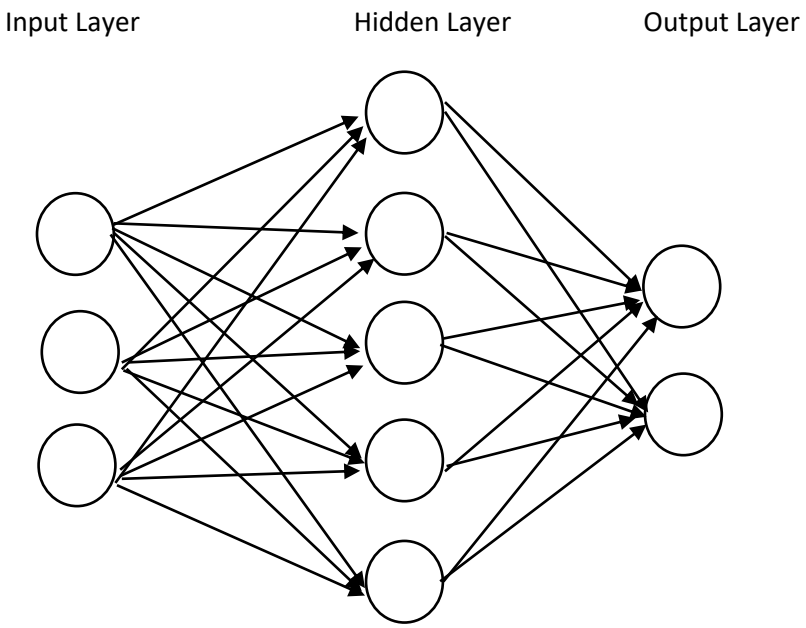

Figure 4.1. Basic Architecture of Neural Network [15]

Figure 4.1 illustrates one input layer, one hidden layer, and one output layer. The input layer has 3 cells or nodes, the output layer has 2 nodes, and the hidden layer has 5 nodes. More than one hidden layer is considered deep learning. Deep learning networks are capable of processing large datasets with billions of parameters. Neural networks can find hidden structures in the unlabeled and unstructured dataset.

4.1.1 Backward propagation

Backward propagation is a significant process in the neural network. The backward propagation process moves from right to left in the neural network. It fine-tunes the weights by minimizing the loss function on each iteration which in turn improves the performance and generalization of the model [16].

4.1.2 Activation

An activation function is also called a transfer function. The activation function determines the output from the input layer. The range from the activation layer is between 0 and 1 or -1 and 1 depending on the activation layer.

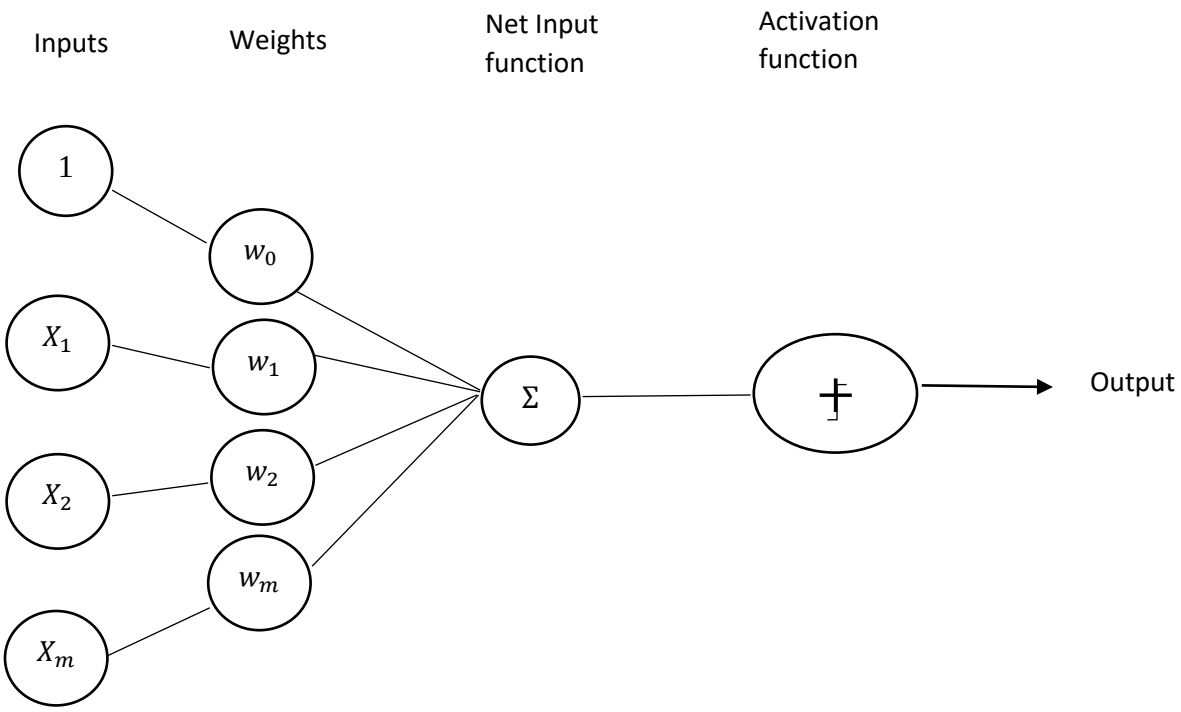

Figure 4.2. One node with activation function [17].

The non-linear activation function commonly used are:

4.1.2.1 Sigmoid function

In the Sigmoid function, the prediction probability is between 0 and 1. The Sigmoid activation function is used for binary classification. The equation for the sigmoid function is given by:

$$f(z) = \frac{1}{1 + e^{-z}}$$

The derivative of sigmoid function is given as:

$$f`(z) = f(z)(1 - f(z))$$

4.1.2.2 Tanh

Tanh is a hyperbolic tangent activation function that is better than the sigmoid function. The range of Tanh is between -1 and 1. Similar to the sigmoid function, Tanh is used for two classes of classification. The equation for the Tanh function is given by:

$$\tanh(x) = \frac{2}{1 + e^{-2x}} - 1$$

The derivative of Tanh function is given as:

$$f`(x) = 1 - f(x)^2$$

4.1.2.3 ReLU

ReLU is a Rectified Linear Unit activation function. The range of ReLU is between 0 and ∞. The equation for ReLU is given by:

$$f(x) = \max(0, x)$$

If x is less than 0, f(x) is 0. If x is greater than or equal to 0 then f(x) is x. Therefore, the range is between 0 and infinity for the ReLU function.

4.1.2.4 Leaky ReLU

Leaky ReLU is based on ReLU but it has a small negative slope. In this f(x) is ax if x is less than 0. The value of a is very small at 0.01. Therefore, the range of Leaky ReLU is between – infinity to infinity.

4.2 RNN

A recurrent neural network is a type of Artificial Neural network which works on sequential data or time series data. RNN is commonly used for natural language processing tasks, language translation, speech recognition, and image captioning. They have a concept of memory which store the information of previous input to generate the next output of the sequence. In traditional neural networks, the input and output are independent of each other while in RNN output depends on the previous state [18].

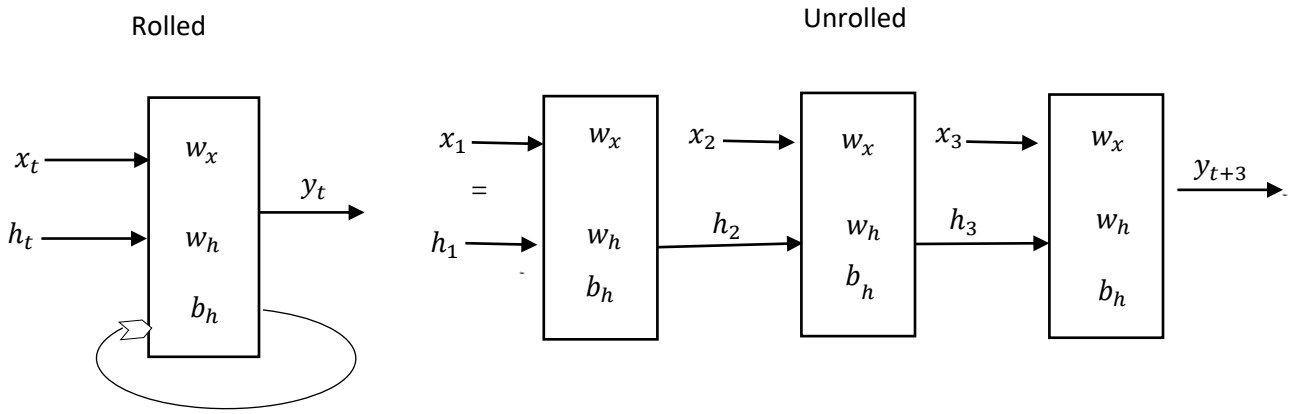

Figure 4.3. Recurrent Neural Network Architecture [19]

The rolled figure in Figure 4.3 shows the feedback loop. The unrolled figure shows 3-time steps to produce the output y_{t+3}. h_t stores the value of hidden units as a time. The unrolled network is similar to a feedforward network. Where h_2 = f(x_1, h_1, w_h, w_x, b_h) = $f(w_x x_1 + w_h h_1 + b_h)$ [19].

4.2.1 Activation layer in RNN

The commonly used activation function in RNN is the Sigmoid function, Tanh function, and ReLU function.

4.2.1.1 Architecture with activation layer

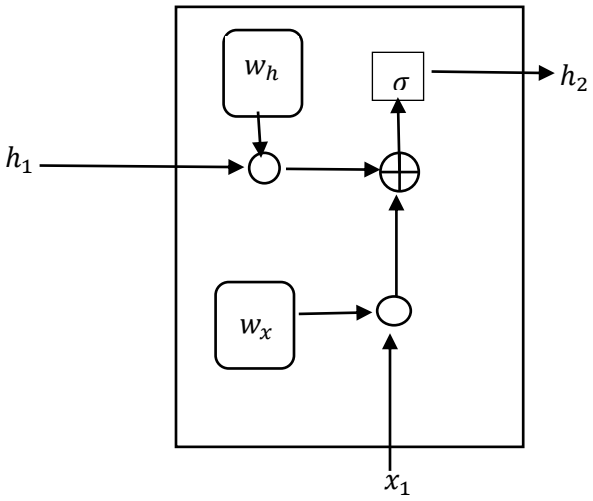

Figure 4.4. RNN hidden layer internal architecture with activation function [20]

Figure 4.4 illustrates the internal architecture of hidden layer. Let us take the layer 1 where x_1 is the input. With sigmoid activation function it is given as:

$$h_2 = \sigma(w_h h_1 + x_1 w_x)$$

Where elementwise addition is denoted as, $\oplus$ and dot product is denoted as $\bigcirc$

In RNN, the same weight parameter is shared across each layer of the network. RNN uses backpropagation through a time algorithm to calculate the gradient. BPTT is different from traditional backpropagation algorithms where BPTT sums errors at each time step [18].

4.2.2 Disadvantages of RNN

RNN has two major issues which are known as exploding gradients and vanishing gradients during backpropagation through time.

4.2.2.1 Vanish gradients

During backpropagation, the gradient which is derivative of the cost function is calculated with respect to weights to update the weight in each layer to minimize error. If the gradient is smaller, the model learns very slowly, and training time takes longer due to which the gradients vanish. This occurs due to sigmoid and tanh activation functions [21].

4.2.2.2 Exploding gradient

On other hand, when the gradient is larger, the slope is steeper which causes the gradient to explode. This occurs due to assigning importance to weights [22].

RNN is not suitable for larger sequences since it takes a long time to train, also, their memory is short-term to process long sequences due to gradient vanishing and exploding.

4.3 LSTM

LSTM (Long Short-Term Memory) network is designed to overcome long-term dependency issues in RNN due to gradient vanishing and exploding. LSTM has a special gate called forget gate which helps the network to remember the state for the long term.

4.3.1 Architecture of LSTM

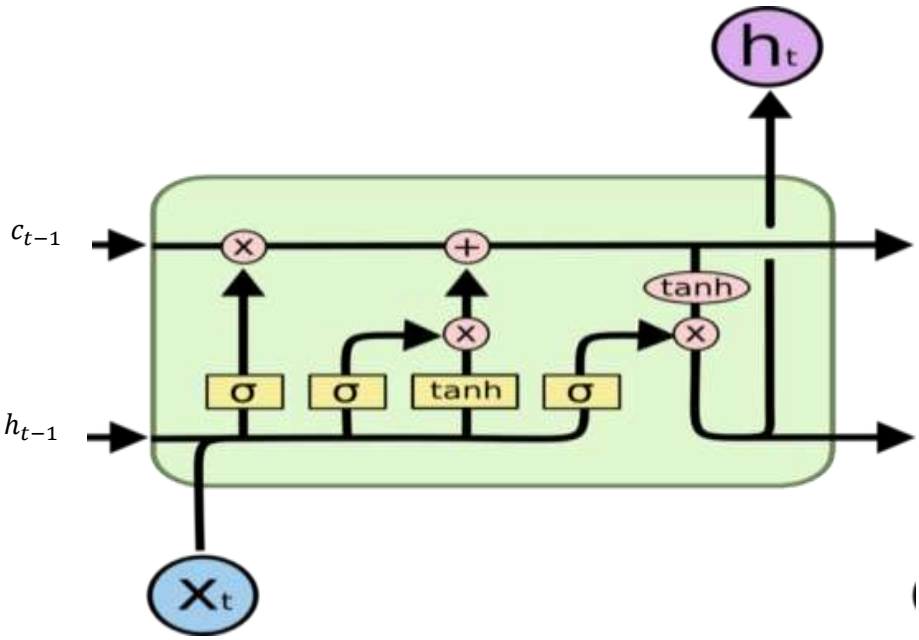

Figure 4.5. LSTM cell representation [23].

Figure 4.5 illustrate the working of LSTM cell in the hidden layer. The cell has 3 gate functions, forget gate which is specific to the LSTM network, the input gate, and the output gate.

4.3.2 Forget gate

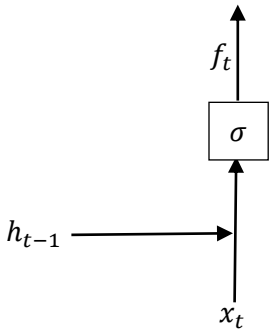

Figure 4.6. Forget Gate

Forget gate decides whether the information in cell state should be omitted or not. h_{t-1} is the previous state in the network. x_t is the input and f_t is the output. The output from the sigmoid function is given as,

$$f_t = \sigma(W_f.[\,h_{t-1}, x_t] + b_f)$$

4.3.3 Input gate

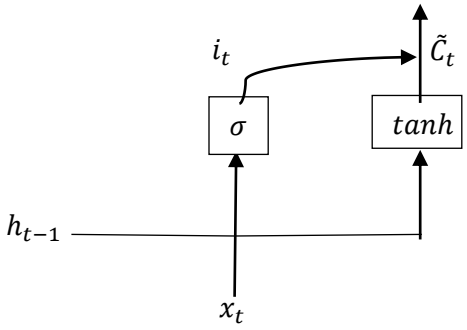

Figure 4.7. Input gate

Input gates decide what new information to store in the cell state. i_t is the output with ranges 0 and 1 and decide which new candidates are relevant for the current time step. Whereas $\tilde{C}_t$ ranges between -1 and 1. The new candidate is stored in the cell state.

$$i_t = \sigma(W_i.[\,h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = tanh(W_C.[\,h_{t-1}, x_t] + b_C)$$

4.3.4 Updating cell state

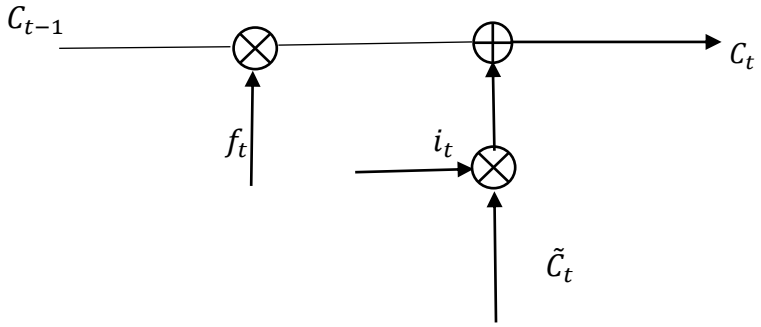

Figure 4.8. Updating cell state based on output from input gate and forget gate

After processing forget and input operation cell state gets updated with the new value. The value from forget gate is multiplied with the cell state C_{t-1} and the value is omitted. Now, the value from i_t and $\tilde{C}_t$ are multiplied and based on the importance of the value they are updated as a new cell state.

4.3.5 Output gate

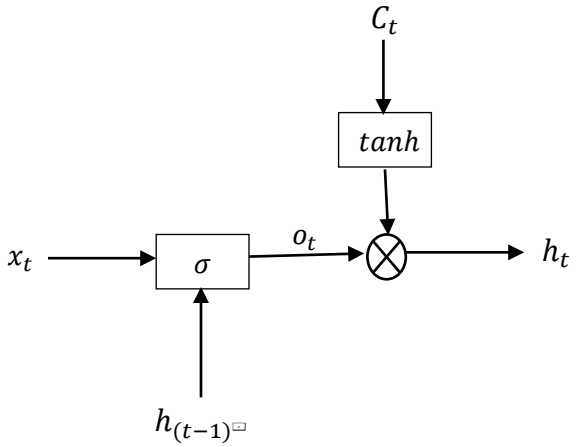

Figure 4.9. Output gate

In the output gate, the cell state passes through the tanh function which gives the output range between -1 and 1. The output from the tanh and o_t are multiplied and a relevant new candidate is sent to output to h_t [22].

$$o_t = \sigma(W_o . [\, h_{t-1}, x_t] + b_o)$$

$$h_{t=}\, o_t * \tanh(C_t)$$

4.4 Assembly Language Model for Instruction Embedding based on BERT with LSTM

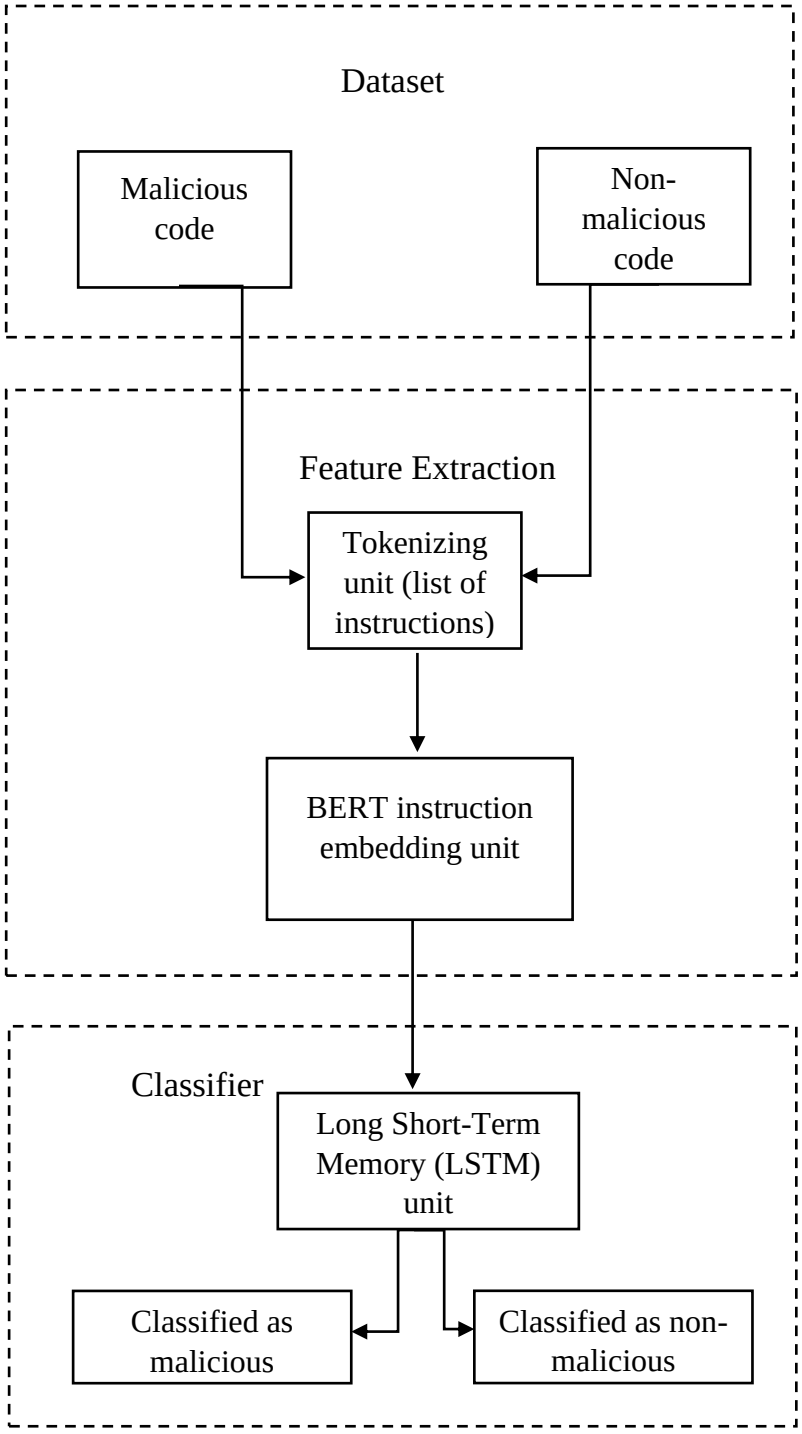

Figure 4.10. Block diagram of assembly language model for instruction embedding with LSTM

In feature extraction, the malicious dataset and non-malicious dataset are cleaned in the pre-processing unit. From pre-processing unit, the dataset is passed to a pre-trained instruction embedding transformer model to determine the vectors of the dataset. The malicious vectors from instruction embedding are labeled as ‘0’ and the non-malicious vectors are labeled as ‘1’. Both vectors are merged to feed into the classifier which is the LSTM network for classification. Each instruction embedding vector from the transformer is (n x 128) where 128 is the vector size and ‘n’ is the number of lines in each file. For instance, if there are two files with code A and B, A is of length n=5, and B is of length n=6, then the vectors of each instruction are 5x128 and 6x128, respectively. The unequal length of input vectors cannot be fed into the LSTM network which requires equal row vector size. The input vectors are padded to make the rows equal. The dataset is split into train and test and then fed into the classifier for classification. The classifier classifies the malicious code in which strings of jumps are added from the non-malicious assembly code. In this project, bi-directional LSTM is used.

4.4.1 Padding

The maximum row length from the list of the instructions is taken as a reference for padding other instructions row lengths. The row and column are padded with zeros to make the rows of all instructions equal.

4.4.2 Train and test split

For training and testing, the dataset is split into 80% and 20%. In general, the dataset is split into 80% for training and 20% for testing or 67% for training and 33% for testing for better performance of the model.

4.4.3 Class weight

Class weights are used when there are discrepancies in the dataset. Especially, in classification, if positive samples are not equal to negative samples, then using class weight gives equal importance to all classes.

4.4.4 Bidirectional LSTM

In this project, bidirectional LSTM is used. The bidirectional LSTM improves the performance of the model on sequential classifications. In bidirectional LSTM, the input flows in two directions one is from forward to backward and another is from backward to forward. Both LSTM network process the input independently and then combine the output from both forward and backward LSTM [23].

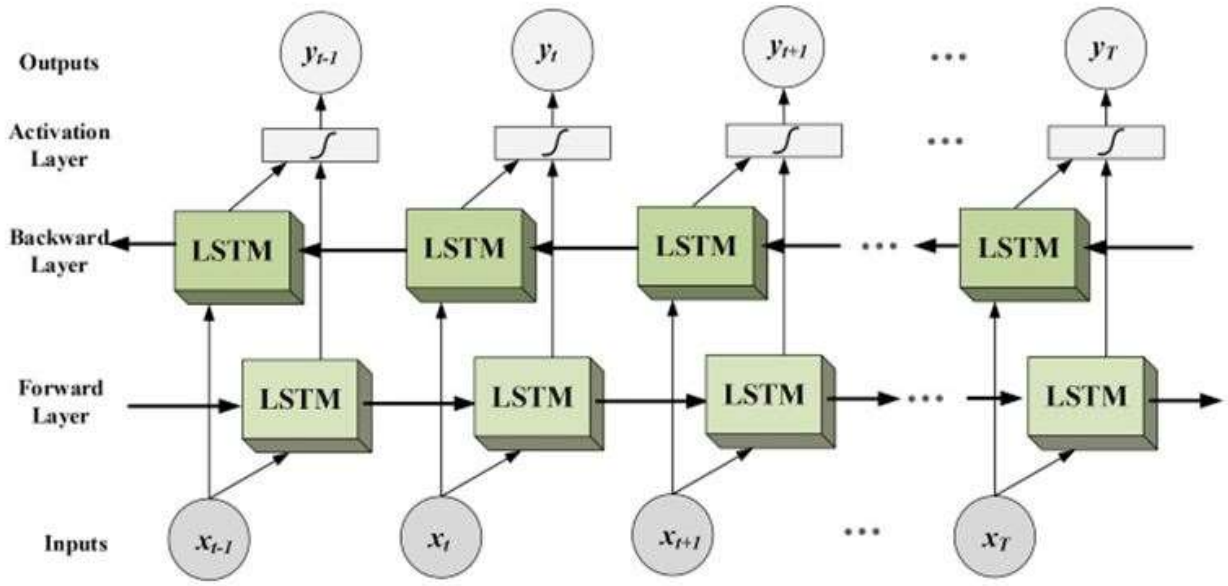

Figure 4.11. Bidirectional LSTM network [23]

4.4.5 Dense layer

The dense layer receives input from the bidirectional LSTM layer. The dense layer performs the matrix multiplication and is used to change the dimension of the vector. They are also called fully connected layers. Here the unit of dense layer is 2.

4.4.6 Activation layer

Softmax and sigmoid activation layers have been used. The performance and test accuracy of the model deteriorate using other activation layers such as tanh and Relu in classifying malicious

from non-malicious code. The softmax and sigmoid outperformed the tanh and ReLU activation layers.

4.4.7 Adam optimizer

The optimizer is a function used to minimize the loss function or error function. They help to choose suitable weights and learning rates to reduce the overall loss and improve accuracy. In this project, an Adam optimizer is employed to update the suitable weights. The advantage of using the Adam optimizer over other optimizers such as SGD (Stochastic Decent Gradient), Adagrad, etc., is that they have faster computation time, low memory requirements, and requires only fewer parameter to tune.

Adam optimizer is a first-order gradient-based optimization of stochastic objective functions. It is based on adaptive estimation of lower-order moments. The Adam optimizer combine advantages of both AdaGrad and RMSProp methods. First (m_t) and second (v_t) moments of gradients are estimated to compute individual adaptive learning rates. The steps in the algorithm involve finding the gradient of the stochastic objective function by taking derivation with respect to the stochastic objective parameter (θ) at timestep t which is given as $g_t$and expressed mathematically as [25],

$$g_t = \nabla_\theta f_t(\theta_{t-1})$$

With g_t, m_t and v_t is calculated as,

$$m_t = \beta_1 . m_{t-1} + (1-\beta_1) . g_t$$

$$v_t = \beta_2 . v_{t-1} + (1-\beta_2) . g_t^2$$

β_1 and β_2 are the decay rate of the average of the gradient.

Bias corrected first and second moment are calculated as,

Finally, parameter theta is updated as,

$$\theta_t = \theta_{t-1} - \alpha . \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$$

4.4.8 Train

Training a model is a significant step in a neural network. The argument that takes for the fitting model is instruction embedding vectors of assembly code, label, epoch, batch size, call back, and class weight.

The call back function monitors the loss and stops training the model early based on the monitored parameter and patience.

4.4.9 Test

Before testing, the model is evaluated using instruction embedding vectors and labels. The predict function takes only the instruction embedding vectors and the batch size. Based on the input the predict function predicts either 1 or 0 to classify if the given sample is non-malicious or malicious.

4.4.10 Parameter

4.4.10.1 BERT

BERT base has 12 layers, 12 attention heads, 110 million parameters, and a hidden dimension of 768. However, in the assembly code, the instruction embedding model using BERT has 128 hidden dimensions, 12 layers, and 8 attention heads for better efficiency.

4.4.10.2 LSTM

The LSTM network has 183,200 layers in the bidirectional LSTM layer and 402 layers in the dense layer with a total trainable parameter of 183,602.

Chapter 5

Evaluation Metrics

The performance of the model is determined by evaluation metrics. Evaluation metrics are significant in determining the efficacy of the model. The evaluation metrics used to determine here are accuracy, confusion matrix, recall, precision, and F1 score.

5.1 Accuracy

The accuracy of the model is in percentage which is calculated by dividing correct predictions by the total number of predictions.

Accuracy = correct prediction / overall prediction

5.2 Confusion matrix

The confusion matrix gives the summary of correct and incorrect predictions which are given by true positive, true negative, false positive, and false negative. The malicious is represented as negative and non-malicious is represented as positive.

		Actual Values: Positive (1)	Actual Values: Negative (0)
Predicted Values	Positive (1)	True Positive	False Positive
	Negative (0)	False Negative	True Negative

Table 5.1. Confusion Matrix

5.2.1 True Positive

True positive is when the predicted positive is same as the actual positive values

5.2.2 True Negative

True negative is when the predicted negative is same as the actual negative values

5.2.3 False Positive

False positive is where the actual values are negative and predicted values are positive.

5.2.4 False Negative

False negative is where the actual values are positive and predicted values are negative.

5.3 Precision

Precision is the calculation of true positives in the given sample. It is the ratio of predicted true positives to the total number of positives predicted in samples.

Precision = True Positives/ (True Positives + False Positives)

5.4 Recall

The recall is a fraction between true positive and, classification and misclassification of positive samples.

Recall = True Positives/ (Ture Positives + False Negative)

5.5 F1 score

F1 score is defined as the harmonic mean of precision and recall.

F1 Score = (2*Precision*Recall)/ (Precision + Recall)

5.6 Evaluation

The performance of the model is determined by softmax and sigmoid activation layers with epochs of 30 and batch sizes of 4 and 8.

5.6.1 Sigmoid activation layer with epoch = 30, batch size = 4 and patience = 5

5.6.1.1 Accuracy vs Epoch and Loss vs Epoch:

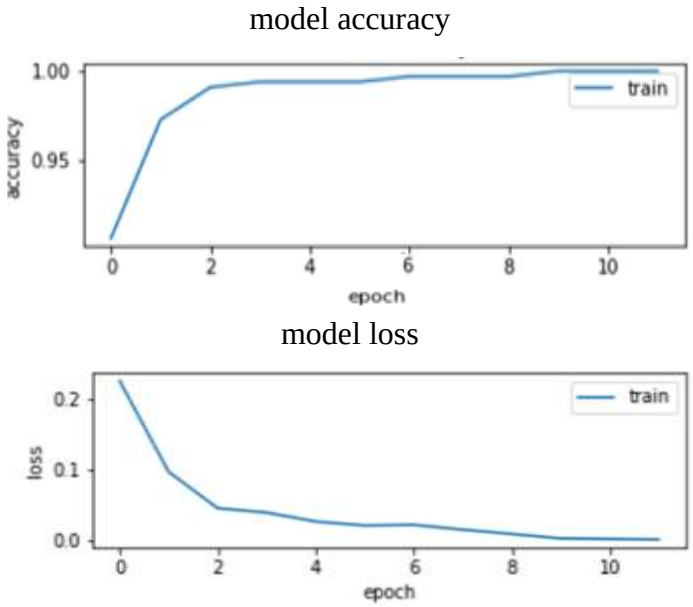

Figure 5.1. Accuracy vs Epoch and Loss vs Epoch

Figure 5.1 shows the graph between accuracy and epochs and loss and epochs. For batch size 4 the training stops at the 12th epoch.

5.6.1.2. Confusion matrix and Accuracy

		Actual Values: 1	Actual Values: 0
Predicted Values	1	TP 48	FP 0
	0	FN 2	TN 53

Table 5.2. Confusion matrix for sigmoid activation layer and batch size = 4

Precision and recall for the sigmoid activation layer are 1.0 and 0.96, respectively. F1 score is 0.979 with a test accuracy of 98% and validation accuracy of 100%.

5.6.2. Sigmoid activation layer with epoch = 30, batch size = 8 and patience = 5

5.6.2.1. Accuracy vs Epoch and Loss vs Epoch

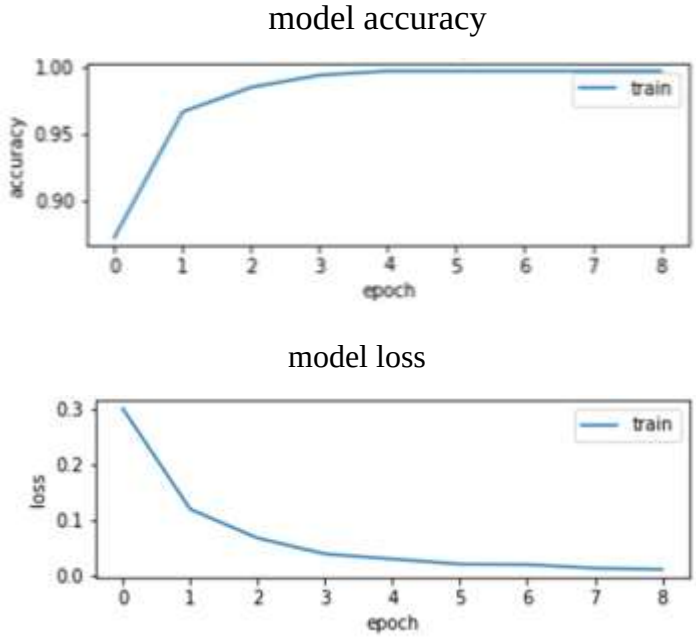

Figure 5.2. Accuracy vs Epoch and Loss vs Epoch

Figure 5.2 shows the graph between accuracy and epochs and loss and epochs. For batch size 8 the training stops at the 9th epoch

5.6.2.2. Confusion Matrix and Accuracy

		Actual Values: 1	Actual Values: 0
Predicted Values	1	TP 49	FP 0
	0	FN 1	TN 53

Table 5.3. Confusion matrix for sigmoid activation layer and batch size = 8

Precision and recall for the sigmoid activation layer are 1.0 and 0.98, respectively. F1 score is also 0.98 with a test accuracy of 99% and validation accuracy of 96%.

5.6.3. Softmax activation layer with epoch = 30 and batch size = 4 and patience = 5

5.6.3.1. Accuracy vs Epoch and Loss vs Epoch

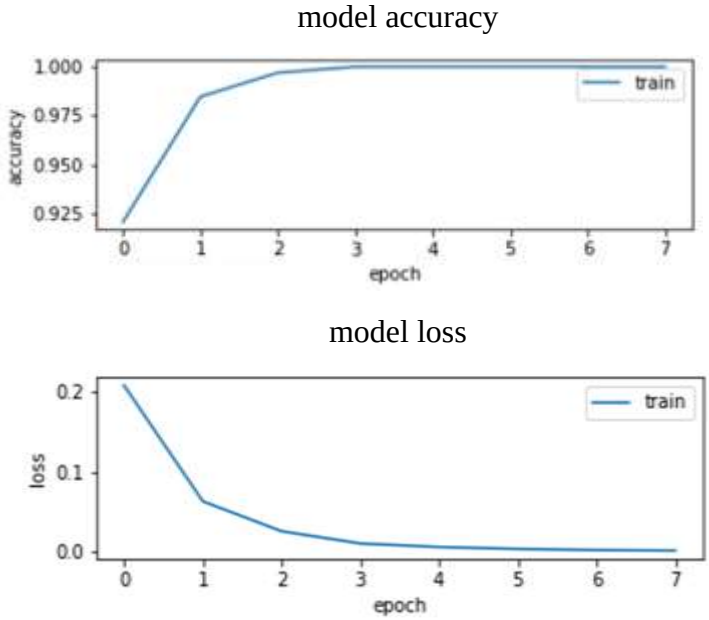

Figure 5.3. Accuracy vs Epoch and Loss vs Epoch

Figure 5.3 shows the graph between accuracy and epochs and loss and epochs for batch size 4. The training stops at 8th epoch.

5.6.3.2. Confusion matrix and accuracy

		Actual Values: 1	Actual Values: 0
Predicted Values	1	TP 50	FP 3
	0	FN 0	TN 50

Table 5.4. Confusion matrix for softmax activation layer and batch size = 4

Precision and recall for the softmax activation layer are 0.943 and 1.0, respectively. F1 score is 0.9708 with a test accuracy of 97.08% and validation accuracy of 97.59%.

5.6.4 Softmax activation layer with epoch = 30 and batch size = 8 and patience = 5

5.6.4.1 Accuracy vs Epoch and Loss vs Epoch

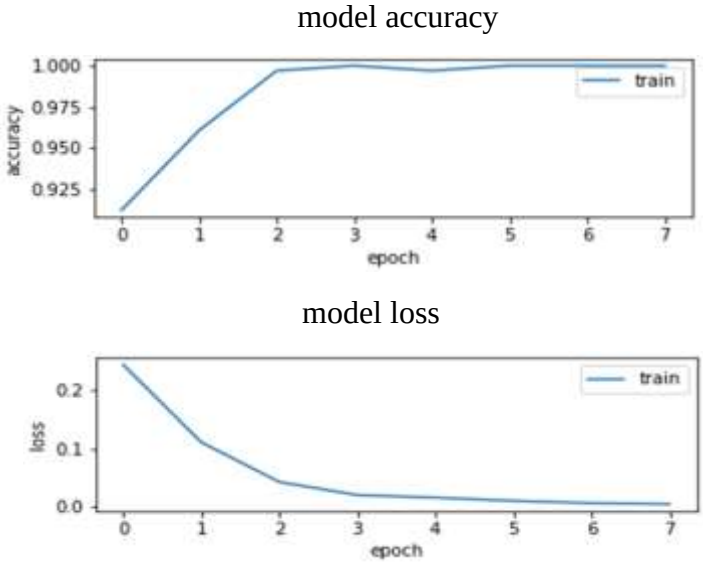

Figure 5.4. Accuracy vs Epoch and Loss vs Epoch

Figure 5.4 shows that graph between accuracy and epochs and loss and epochs for batch size 8. The training stops at the 8th epoch

5.6.4.2 Confusion matrix and accuracy

		Actual Values: 1	Actual Values: 0
Predicted Values	1	TP 49	FP 2
	0	FN 1	TN 51

Table 5.5. Confusion matrix for softmax activation layer and batch size = 8

Precision and recall for the softmax activation layer are 0.9607 and 0.98, respectively. F1 score is also 0.970 with a test accuracy of 97% and validation accuracy of 100%.

Chapter 6

Discussion and Existing Approaches

The performance of the model as a result of using assembly language model for instruction embedding, as well as using two different activation functions in the output layer, is discussed in this chapter. Furthermore, the disadvantages of using GPT-2 for this project are discussed, as are comparisons between other embeddings and assembly language model for instruction embedding based on BERT transformer.

6.1 Discussion

As described in Chapter 5, various evaluation metrics such as accuracy, precision, recall, confusion matrix, and F1 score are determined for 85 validation samples and 103 test samples. Two different types of activation layers, the sigmoid activation layer and the softmax activation layer, are used in the output layer to train the model. With a batch size of 8 and a sigmoid activation layer, the model achieves 99% test accuracy. Similarly, training the model with batch size of 4 and the sigmoid activation function results in 98.05% test accuracy. Training model with softmax activation function and batch size of 4 results in 97.08% test accuracy. Similarly, with the softmax activation function and batch size of 8, the model achieves 97% test accuracy. The precision value for sigmoid activation is 1.0 for batch sizes of 4 and 8. For batch sizes of 4 and 8, the recall values are 0.96 and 0.98, respectively. Here, the value of precision is higher than recall due to a high false positive rate. Using the softmax activation function, the precision value is 0.943 and 0.9607 for batch size of 4 and batch size of 8. The recall for batch size of 4 and batch size of 8 are 1.0 and 0.98, respectively. Here, the recall value is higher than the precision due to the high false negative rate.

The pre-processing time for malicious and non-malicious datasets is 5 seconds. The training time for the sigmoid activation function and batch size of 4 is 206 seconds, whereas batch size of 8 is 86.87 seconds. Similarly, the training time for the softmax activation function and batch size of 4 is 120.94 seconds, whereas the training time for batch size of 8 is 66.94 seconds. The smaller the batch size, the longer it takes to train the model.

The memory consumption for the sigmoid activation function for batch size of 4 and batch size of 8 is 6976 MB and 6978.72 MB, respectively. Similarly, the memory consumption for the softmax function for batch size of 4 and batch size of 8 is 6980.37 MB and 6971 MB, respectively.

However, the batch size of 8 in sigmoid, and the softmax activation function in the output layer take less time compared to the batch size of 4. Again, comparing the softmax activation function and sigmoid activation function with batch size of 8, it is observed that the softmax activation function time is less than the sigmoid activation function. Comparing the performance of the sigmoid and softmax activation layers, the sigmoid activation layer with batch size of 4 and batch size of 8 has a lower false positive rate than the false negative rate. Softmax activation layer for batch sizes of 4 and batch size of 8 has a high false positive (misclassified malicious as non-malicious). If the size of the test dataset increases, then the probability of misclassification as non-malicious instead of malicious may increase using the softmax activation function. This will eventually increase the false positive rate. On the other hand, misclassification as malicious instead of non-malicious increases the false negative rate using the sigmoid activation function, which is more tolerable than a high false positive rate. Also, the test accuracy in both cases may vary depending on the test samples. In real world, if test sample has unbalanced or biased test samples it can lead to poor performance of the model. By increasing the size of the test samples the performance of the model can be improved.

6.2. Existing Approaches

Initially, GPT-2 (Generative Pre-trained) was planned to be used for malicious and non-malicious classification instead of an assembly language model for instruction embedding model based on BERT transformer with LSTM. GPT-2 is a transformer-based language model. They have 1.5 million parameters (weights and biases of the layer) trained on an 8 million web page dataset. Since GPT-2 is pre-trained with web pages, it must be fine-tuned to train the model with assembly code. Also, the objective of the GPT model is to generate text based on the learned text dataset. However, the main objective of this project is to classify malicious code from non-malicious code, which makes GPT-2 not suitable for this task.

The GPT-2 model requires large assembly code datasets to fine-tune and classify malicious code from non-malicious code. Also, fine-tuning the GPT-2 model with assembly code is a time-consuming process for this project. Assembly language model for instruction embedding based on BERT transformer is already fine-tuned with a large assembly dataset of 2.25 billion, which would be more suitable for the downstream task of classification using LSTM.

The other existing approach to converting instructions to vectors is the Instruction2vec model. Instruction2vec model performance is poor compared to embedding instructions. It is because the feature selection in Instruction2vec is manual, which affects the performance of the downstream task [26, 27].

The Asm2vec model is another existing approach that addresses problems like manual feature selection. But they failed to consider the relationship between features when identifying a unique pattern that is related to assembly code. They learn embedding for opcode and operand separately. Asm2vec performs slightly better than the Instruction2vec model. However, the embedding instruction model outperforms the asm2vec model [28].

The PALMTREE model [5] outperforms other existing models such as Istruction2Vec, Asm2Vec, and Word2Vec. In PALMTREE [5] evaluation, the set of instructions is randomly created with an outlier in it. The presence of an outlier in a set of instructions indicates that the instruction is distinct from the other instructions. The outlier is detected by calculating the cosine distance between two vector representations of instructions and selecting the vector that is most

distant from the rest of the vectors. Cosine distance is a measure of similarity between two sequences of numbers. The performance is evaluated by creating 50,000 sets of instructions based on the operand and another 50,000 sets of instructions based on the opcode.

Model	Opcode Outlier Average Accuracy	Operand Outlier Average Accuracy
Instruction2Vec	0.863	0.860
Word2vec	0.269	0.256
Asm2Vec	0.865	0.542
PALMTREE	**0.871**	**0.944**

Table 6.1 Accuracy for Operand and Opcode outlier detection of different model [5]

Table 6.1 shows the average accuracy of different models for Opcode and Operand outlier detection. From the above table, it can be inferred that the average accuracy of PALMTREE [5] model is better than other models.

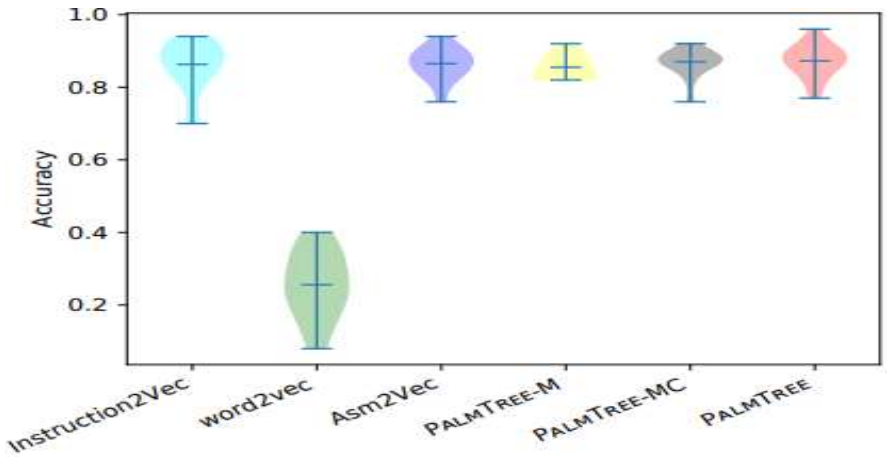

Figure 6.1 Accuracy of Opcode Outlier Detection [5].

Figure 6.1 shows the range of accuracy between different embedding models for opcode outliers.

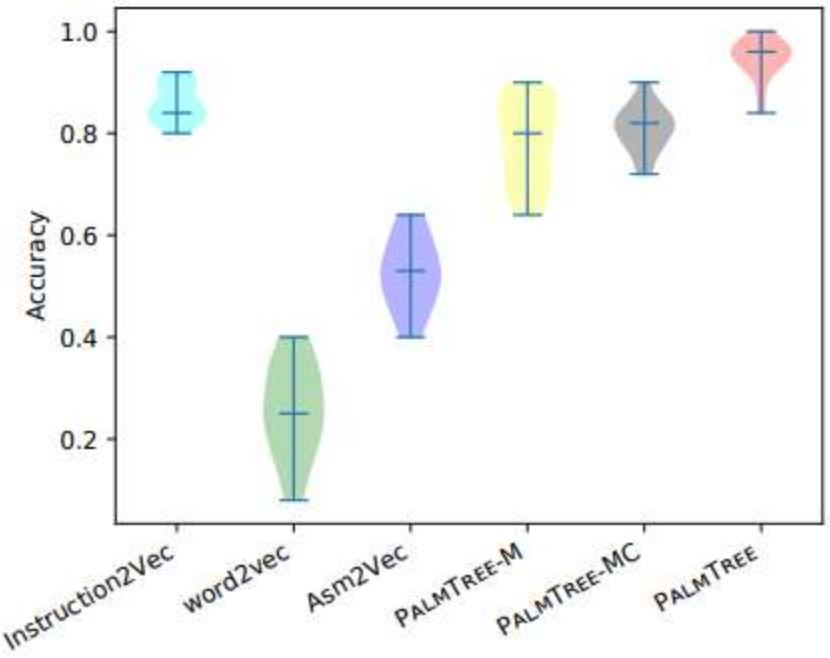

Figure 6.2 Accuracy of Operands Outlier Detection [5].

Figure 6.2 shows the accuracy between different embedding models for operands outlier.

From the above, it is inferred that PALMTREE [5] result in high accuracy compared to other embedding models. The accuracy in classifying malicious from non-malicious assembly code is very high when the embedding (vector representation) from the assembly language model for instruction embedding is fed to the LSTM classifier. This also improves the training speed of the network [5].

Chapter 7

Software and Tools

The classification task is accomplished by using various software and tools. The python language has been used for pre-processing assembly code, training, and testing the dataset since they provide a large number of libraries for developing machine learning tasks. In this section, various IDEs, frameworks, and libraries used for performing classification are discussed.

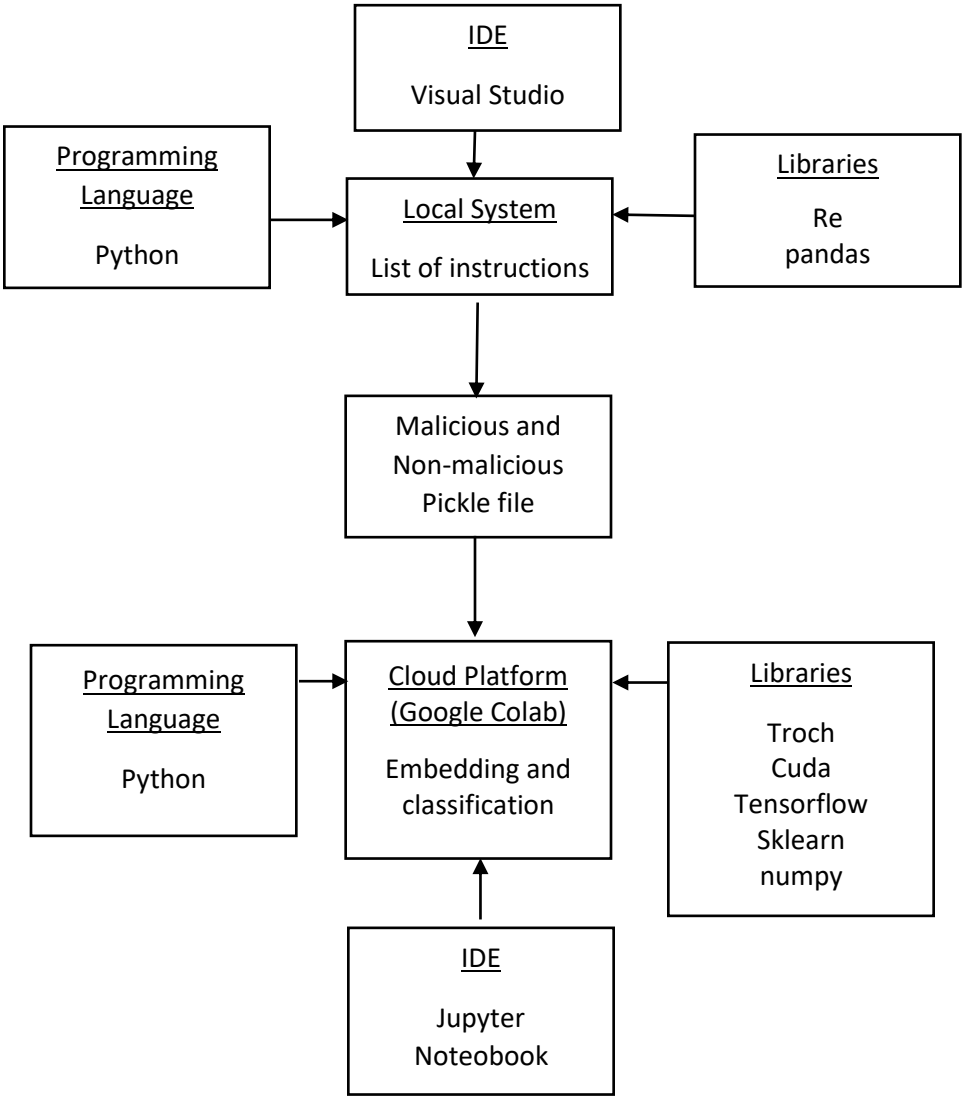

Figure 7.1 Development using local system and cloud platform block diagram

Figure 7.1 explains the development using both the local systems and the cloud platform. The block diagram shows the libraries, IDE, and programming languages used to develop classifiers both in the local system and on the cloud platform. In the local system, Visual Studio is used as an IDE, Python is used as the programming language, and libraries such as Re, Pandas, and Pickle are used. These libraries are used to clean the dataset. The list of instructions of malicious and non-malicious code is converted to a pickle file. In the cloud platform, Python is used as a programming language, and Jupyter Notebook is used as an IDE. Jupyter notebook is used since it is user-friendly for machine learning tasks and data can be visualised and shared. Libraries such as CUDA for GPU access, Torch, TensorFlow, NumPy, and Sklearn are used since these libraries were developed to provide machine learning models and computation to solve machine learning tasks. The pickle is uploaded to Google Colab, and datasets are accessed for instruction embedding and classification.

7.1 Google Colabatory

Google Colabotary or Google Colab is a free Jupyter notebook environment managed and run in the cloud. They have GPU, System RAM, and disk. The total memory allocation for RAM is 12.68GB and for the disk is 166.83GB. The downstream task of training, testing, and obtaining instruction embedding vectors from the pre-trained transformer models are done using google collaboratory. They also have inbuilt in python and other python libraries such as pandas, NumPy, sklearn, Keras, etc., [29]

7.2 IDE (Integrated Development Environment)

Integrated Development Environment is a software application to develop software code. Other than text editing, IDE has capabilities to build and test the code, debug, and automatic error corrections. There are local IDEs and cloud IDEs. Local IDEs are directly installed on the local machine and cloud IDEs allow the developer to compile and run the code directly in the browser.

7.2.1. Jupyter Notebook

Jupyter notebook is an open-source web application-based interactive IDE (Integrated Development Environment) used to create and share documents. It supports programming

languages such as python, R, Julia, and Scala. Jupyter notebook has other features such as running the code, displaying output, adding explanations to the code, and charts making the code understandable and shareable. In this project, Jupyter notebook is used as an IDE where pre-trained transformers are uploaded to convert a list of instructions into vectors. It is also used for the training and testing of LSTM networks for classification. Converting a list of instructions to vectors uses a pre-trained BERT transformer model which requires GPU and a large amount of memory. Jupyter Notebook supports machine-learning libraries in python.

7.2.2. Visual Studio Code

Visual studio code is a source code editor for developing an application. It includes development supports such as debugging, intelligent code completion, code refactoring, and embedded Git. The pre-processing step and adding strings of jumps to malicious code are done in visual studio code. The malicious and non-malicious tokenized (list of instructions) assembly codes are stored in pickle format to upload in Jupyter Notebook for word embedding and classification process.

7.3. Python

Python is a high-level language used for multiple tasks such as building websites, developing software, machine learning, automation, Image Processing, etc., It supports structured, object-oriented, and functional programming. In this project, python is used as a programming language for pre-processing, testing, and training. For pre-processing tasks such as adding malicious code to the dataset and cleaning the assembly code Python 3.9 version is used. In google collaboratory, a built-in Python 3.7.13 version is used. Python is chosen as a programming language because it provides a large number of libraries for processing data, simplicity, and readability, and has a lesser learning curve. Python libraries such as OpenCV, scikit-learn, NumPy, and pandas are used for machine learning tasks. All these libraries are written for the python programming language.

7.4. Python Libraries

Python libraries are reusable code that contains a collection of modules and packages. Python has many libraries that supports machine-learning task. In this project python libraries and

frameworks used are Pytorch, Re, TensorFlow and Keras, Pandas, and Sklearn. These libraries and frameworks are explained below.

7.4.1. TensorFlow and Keras

TensorFlow is an open-source deep learning framework for deep learning tasks. Keras is an API for developing deep learning life cycles such as fit, evaluate, and train. Keras API is integrated with TensorFlow 2 framework. Model, layers, and optimizer are classes in Keras API used to develop the LSTM network [30].

7.4.2. Pandas

Pandas is a library in python used to manipulate data in DataFrames. Pandas support different file formats such as CSV, JSON, MS Excel, and SQL database queries. These types of files can be loaded into DataFrame for manipulation.

7.4.3. Re

Re is a Regular Expression Operations library in python. They are used to match or search the string in the given pattern using Re functions such as match and search. They also have other functions to split the given string and find all the string that matches the pattern.

7.4.4. Sklearn

Scikit learn is a machine learning library in python which has classes and functions for both supervised learning and unsupervised learning. It has classes and functions for predictive analytics such as Linear models, metrics, loaders, feature selection, model selection, pipeline, neural network, etc., Importing the library in python, the classes for models, metrics, and feature selection can be used.

7.4.5 Torch

PyTorch is an open-source deep learning framework used for computer vision and natural language processing tasks. BERT library is used from Pytorch, also, it supports CUDA for utilizing GPU for computation.

7.5 GCC compiler

GCC is a GNU Compiler Collection which supports various programming languages, hardware architecture, and operating systems. GCC compiles the code and converts assembly code. In this project, GCC compiler is used to convert C code to assembly code.

7.6 GPU

GPU is Graphical User Interface; it was originally designed to accelerate graphics rendering. They are used in deep learning to compute large mathematical data in a shorter period. NVIDIA Tesla K80 is the GPU used in google colab. In this project, GPU is required only for embedding the pre-processed assembly code into vectors in google colab.

7.7 CPU

The CPU used for pre-processing is Intel Core i7-6600U. The RAM memory of 16.0 GB and the drive memory of 241.58 GB is used in the pre-processing dataset. Adding strings of jumps, cleaning the assembly code, and pickle conversion are done in the remote system.

Chapter 8

Conclusion and Future Work

The main goal of this project is to develop a classifier to classify malicious from non-malicious assembly code. A collection of x86, x86-series i386 (Intel 80386), and various C source codes from GitHub converted into assembly code using the GCC compiler are used as datasets. 515 assembly code files have been used, of which 265 are considered malicious and 250 are considered non-malicious files. The assembly code has unconditional jump (JMP) instructions that transfer the flow to another routine. In malicious assembly code, these jump instructions are added at the end of each subroutine, forming strings of jumps. These strings of jumps are considered malicious and jump to another routine that might execute a malicious instruction that performs the harmful task.

As discussed earlier in Chapter 1, the classifier is trained with malicious and non-malicious assembly code to recognize the pattern. The classifier learns the pattern of the malicious code with strings of jumps and classifies the code that has strings of jump instructions from the non-malicious code. The process of classification has three tasks.

The first task is to add strings of jumps to convert non-malicious assembly code to malicious assembly code, clean the dataset using the regex library, and tokenize the code into a list of instructions. The second task involves converting a list of instructions to a vector representation of the instructions. For the vector representation (instruction embedding), assembly language model for instruction embedding based on BERT transformer [5] is used. The vector representation of assembly code is labelled "0" for malicious code and "1" for non-malicious code. In the third task, these labelled instruction embeddings are fed into the LSTM classifier for classification. The LSTM classifier network classifies malicious code from non-malicious code.

The LSTM network is more suitable for NLP tasks. The sigmoid activation layer and softmax activation layer are used in the output layer. The performance of the LSTM model is evaluated using these two activation functions. For each activation function, the confusion matrix, accuracy, recall, precision, and F1-score are calculated. Based on these values, the activation

layer that results in high accuracy is considered a better classifier for classifying malicious code from non-malicious code.

8.1 Conclusion

In this project Vulnerability Detection in Assembly Code using Deep Learning, challenges and solution for finding the pattern in assembly code have been summarized. A pre-trained model based on BERT transformers for converting instructions to vectors has been introduced. Along with the pre-trained model, the LSTM model is used as a classifier in downstream tasks to classify malicious from non-malicious code. Accuracy, recall, precision, and F1 score are significant performance metrics for choosing a model. The model achieves the highest accuracy of 99%. Compared to other instruction embeddings such as the asm2vec model [27] and instruction2vec model [26], the assembly language model for instruction embedding based on BERT transformer (PALMTREE) performs efficiently with the LSTM model. Using activation layer sigmoid and softmax with batch sizes of 8 and batch size of 4 outperforms other embedding models with high accuracy. The time taken for training the softmax activation function is 120.94 seconds and 66.94 seconds for batch size of 4 and batch size of 8 using GPU, which takes less training time compared to the sigmoid activation function.

Therefore, it is learned that assembly language model for instruction embedding based on BERT transformer improves the training speed and accuracy of the classifier. Existing word embedding models such as word2vec fail to capture the internal and unique characteristics of the assembly code. Using this simple word embedding model for assembly code may not be suitable. The instruction embedding model improves the quality of the vector representation, which in turn improves the accuracy of the classification and reduces the training speed. Along with this, using the sigmoid activation function in the output layer also supports the model's goal of reducing the false positive rate of misclassifying malicious as non-malicious. A high false positive rate (misclassified malicious as non-malicious) is considered more harmful than a high false negative rate (misclassified non-malicious as malicious). Considering "0" as malicious and "1" as non-malicious, achieving high precision is more significant than achieving high recall. Even though the speed of the sigmoid activation function with batch size of 8 in the output layer is less compared to the softmax activation function, the sigmoid activation function with batch size of 8 results in high test accuracy with a low false positive rate. So, the sigmoid activation function

with batch size of 8 can be considered for training the classifier. This classifier is trained with strings of jumps. Similarly, other vulnerabilities in assembly code can also be trained using the combination of assembly language model for instruction embedding based on BERT transformer with an LSTM classifier for classifying malicious code from non-malicious code.

8.2 Future Work

Converting assembly code to vectors is an automatic process that is done using the pre-trained model. However, the process of cleaning and tokenizing assembly code is tedious. Also, the current process does not include automating the control flow graph or the data flow graph. The future work of this project involves automating the cleaning process and the control and data flow of the assembly code. In addition, the size of the dataset used to train and test the model will be increased, and other vulnerabilities in assembly code will be included. The embedding size is currently set to 128 by default. In the future, a larger embedding size of 256 or 512 will be used for converting to vectors to implement along with downstream tasks. Among these, the predominant enhancement is to replace BERT with GPT-2. Unlike BERT, GPT-2 does not require a massive dataset to train the model, which will make fine-tuning the GPT-2 model less difficult.

Chapter 9

References

[1] Introduction, “Machine learning, explained” [*Online*]. Available: https://mitsloan.mit.edu/ideas-made-to-matter/machine-learning-explained. [Accessed 5 July 2022].

[2] Li, X.; Wang, L.; Xin, Y.; Yang, Y.; Tang, Q.; Chen, Y. 2021. Automated Software Vulnerability Detection Based on Hybrid Neural Network. *Appl. Sci.* 2021, 11, 3201.

[3] Michael J. Hohnka, Jodi A. Miller, Kenrick M. Dacumos, Timothy J. Fritton, Julia D. Erdley, and Lyle N. Long. 2019. Evaluation of Compiler-Induced Vulnerabilities. *Journal of Aerospace Information Systems.* 16. 1-18. 10.2514/1.I010699.

[4] Triet H. M. Le, Hao Chen, M. Ali Babar. 2020. Deep Learning for Source Code Modeling and Generation: Models, Applications and Challenges. *arXiv:2002.05442v1 [cs.SE]* 13 Feb 2020.

[5] Xuezixiang Li, Yu Qu, Heng Yin. 2021. PALMTREE: Learning an Assembly Language Model for Instruction Embedding. *arXiv:2103.03809v3 [cs.LG]* (2021).

[6] Problem Statement, “BERT Explained: State of the art language model for NLP”, [*Online*]. Available: https://towardsdatascience.com/bert-explained-state-of-the-art-language-model-for-nlp-f8b21a9b6270. [Accessed 26 October 2022].

[7] Assembly code structure, “Assembly Language” [*Online*]. Available: https://www.tutorialspoint.com/assembly_programming/assembly_introduction.htm. [Accessed 7 July 2022].

[8] Word Embedding, “What Are Word Embedding for Text”, [*Online*]. Available: https://machinelearningmastery.com/what-are-word-embeddings. [Accessed 19 September 2022].

[9] Word Embeddings, ‘Understanding tf-idf (Term Frequency-Inverse Document Frequency) [*Online*]. Available: https://www.geeksforgeeks.org/understanding-tf-idf-term-frequency-inverse-document-frequency/. [Accessed 11 July 2022].

[10] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. *arXiv:1310.4546v1 [cs.CL]* (2013).

[11] Tomas Mikolov, Kai Chen, Greg Corrado, Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. *arXiv:1310.3781v3 [cs.CL]* (2013).

[12] Disadvantages of TF-IDF and word2vec, "What's in a word?", [*Online*]. Available: https://towardsdatascience.com/whats-in-a-word-da7373a8ccb. [Accessed 25 September 2022].

[13] BERT, "BERT Explained: State of the art language model for NLP". [*Online*]. Available: https://towardsdatascience.com/bert-explained-state-of-the-art-language-model-for-nlp-f8b21a9b6270. [Accessed 15 July 2022].

[14] BERNHARD MEHLIG. 2021. Machine learning with neural networks. *arXiv:1901.05639v4 [cs.LG]* (2021).

[15] Neural Network, "What is neural networks". [*Online*]. Available: https://www.ibm.com/cloud/learn/neural-networks. [Accessed 19 July 2022].

[16] Neural Network, "Back Propagation in Neural Network: Machine Learning Algorithm". [*Online*]. Available: https://www.guru99.com/backpropogation-neural-network.html. [Accessed 26 August 2022].

[17] Neural Network, "Neural Network Elements", [*Online*]. Available: http://wiki.pathmind.com/neural-network. [Accessed 26 August 2022].

[18] Recurrent Neural Networks, "What are recurrent neural networks". [*Online*]. Available: https://www.ibm.com/cloud/learn/recurrent-neural-networks. [Accessed 20 July 2022].

[19] Recurrent Neural Network, "An Introduction to Recurrent Neural Networks And The Math That Powers Them". [*Online*]. Available: https://machinelearningmastery.com/an-introduction-to-recurrent-neural-networks-and-the-math-that-powers-them/. [Accessed 20 July 2022].

[20] AvraamTsantekidis, NikolaosPassalis, AnastasiosTefas. Recurrent Neural Networks. *Deep Learning for Robot Perception and Cognition*. Academic Press: Cambridge, MA, USA, 2002; pp. 101-115.

[21] Disadvantages of RNN. "Vanishing and Exploding Gradients in Neural Networks". [*Online*]. Available: https://www.numpyninja.com/post/vanishing-and-exploding-gradients-in-neural-networks. [Accessed 21 July 2022].

[22] Exploding Gradient, “Disadvantages of RNN”, [*Online*]. Available: https://iq.opengenus.org/disadvantages-of-rnn/. [Accessed 21 July 2022].

[23] LSTM, “Recurrent Neural Networks and LSTM explained”. [*Online*] https://purnasaigudikandula.medium.com/recurrent-neural-networks-and-lstm-explained-7f51c7f6bbb9. [Accessed 22 July 2022].

[24] LSTM, “Complete Guide To Bidirectional LSTM”. [*Online*] Available: https://analyticsindiamag.com/complete-guide-to-bidirectional-lstm-with-python-codes/. [Accessed 24 July 2022].

[25] Diederik P. Kingma, Jimmy Lei Ba. 2017. ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION. *arXiv:1412.6980v9 [cs.LG]* (2017).

[26] Lee, Yongjun, Hyun Kwon, Sang-Hoon Choi, Seung-Ho Lim, Sung Hoon Baek, and Ki-Woong Park. 2019. Instruction2vec: Efficient Preprocessor of Assembly Code to Detect Software Weakness with CNN. *Applied Sciences* 9, no. 19: 4086. https://doi.org/10.3390/app9194086.

[27] Young Jun Lee, Sang-Hoon Choi, Chulwoo Kim, Seung-Ho Lim, Ki-Woong Park. 2017. Learning Binary Code with Deep Learning to Detect Software Weakness. *KSII The 9th International Conference on Internet (ICONI) 2017 Symposium.*

[28] Steven H. H. Ding, Benjamin C. M. Fung, Philippe Charland. 2019. Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 472-489.

[29] Software and Tools, “Google Colab – What is Google Colab?” [*Online*]. Available: https://www.tutorialspoint.com/google_colab/what_is_google_colab.htm. [Accessed 29 July 2022].

[30] Software and Tools, “TensorFlow 2 Tutorial: Get Started in Deep Learning with tf.keras”. [*Online*]. Available: https://machinelearningmastery.com/tensorflow-tutorial-deep-learning-with-tf-keras/. [Accessed 29 July 2022].