Asm2Seq: Explainable Assembly Code Functional Summary Generation

by

Scarlett Taviss

A thesis submitted to the School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen's University
Kingston, Ontario, Canada
(May 2021)

Abstract

Technology is at the forefront of nearly every aspect of the modern world. Humans write code for technology to function as we desire, but often times little is understood about the code needed to control the computer without significant human analysis. This research aims to bridge this gap by producing human-readable summarizations of the functionality of the assembly code needed for computer execution. Vulnerability datasets are used as starting datasets on the model because finding and understanding vulnerabilities in a program are important for software maintenance, software analysis, and software development. Source code files exhibiting various vulnerabilities are compiled to produce their assembly code counterparts and used as input to the model. Descriptions of how the vulnerabilities function are extracted from the source code files and used as the desired output for the model.

Various neural network architectures make up the encoder-decoder experiments to determine the best model. Each experiment undergoes significant training in order to produce accurate predictions. Attention was added in order to understand what aspects of the assembly code had the biggest effect on generating the summary. The models produced high rates of accuracy and Bilingual Evaluation Understudy (BLEU) score, which are both indicative of a well performing network. Comparisons between model predictions and the true descriptions showcase the favorable results.

Acknowledgements

I would very much like to thank my supervisor Steven Ding for all of his support and help through this process; I certainly would not have completed this without you. I would also like to thank my other supervisor Mohammad Zulkernine for giving me the chance to pursue this degree and for giving me the freedom to research what I want. I wish you both all the best!

Contents

Glossary

Bi-GRU Bidirectional GRU.

Bi-LSTM Bidirectional LSTM.

BLEU Bilingual Evaluation Understudy.

CNN Convolution Neural Network.

EOS End-Of-Sequence.

FFNN Feed-Forward Neural Network.

GCC GNU Compiler Collection.

GRU Gated Recurrent Unit.

LSTM Long Short-Term Memory.

NN Neural Network.

RMSprop Root Mean Squared prop.

RNN Recurrent Neural Network.

SCA Sparse Categorical Accuracy.

SOS Start-Of-Sequence.

List of Figures

List of Tables

Chapter 1

Introduction

Humans produce source code in natural language for any digitized technology to be able to function. Computers are only able to read binary code, meaning 1's and 0's. This code is often translated to assembly language because it allows humans to understand what the computer does with those 1's and 0's. A compiler is a special program that does the translation from source code into assembly code for the computer to be able to execute it. Without the help from human analysis, translating assembly code back into source code is not easily done. The goal of this research is to facilitate this analysis by generating a human-friendly natural-text summary of a piece of assembly code that describes its functionality in order to facilitate the manually intensive process of reverse engineering. This research focuses on the functionality of vulnerabilities in the code with the hopes of generalizing the translation to include more reverse engineering problems in future iterations of the model.

Code summarization, machine learning, and binary analysis are all very broad fields that have been studied and combined extensively in preceding years. Previous literature on summarization is primarily focused on summarizing natural language texts [2] [3] [9] [16] [22] [33] [41], and summarizing of source code [14] [23] [24] [28] [42].

Binary analysis research has generally been focused on relating the assembly language to its source code counterparts [10] [26] [29] [37] and analyzing the assembly code to further understand how a specific program functions [6] [43]. Machine learning techniques have been widely used to achieve results in all of these cases. This research combines these three fields and uses machine learning techniques, specifically neural networks, to produce natural short language descriptions that describe the functionality of assembly code.

Figure 1.1 shows the flow of information for manual reverse engineering and the Asm2Seq model. Adding the assembly-to-sequence step gives the reverse engineers and security analysts a head start to decoding the binary program.

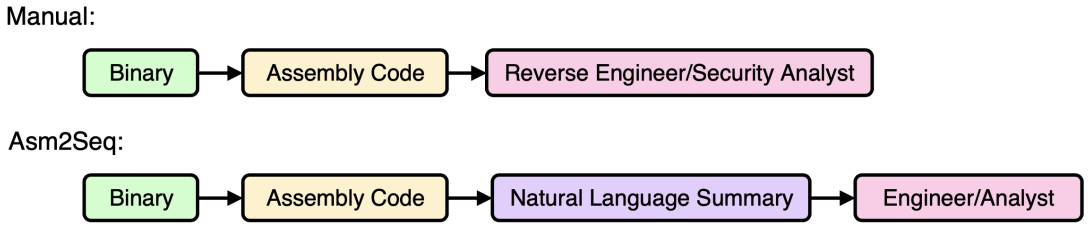

Figure 1.1: Simple flow of information from binary to engineer/analyst for two different models, manual reverse engineering and Asm2Seq.

The data used for the research are vulnerability datasets, meaning each file of source code contains a vulnerability that will also be present in the assembly code. Vulnerabilities are one of the most sought after discoveries in programs because they lead to better software maintenance, software analysis, and software development. Vulnerabilities are also the most sought after discovery of cyber attackers and malware authors in order to penetrate a system for generally ill-intentioned reasons. Discovering vulnerabilities requires extensive background knowledge about the concepts used in a program and a tremendous amount of time and effort. The datasets

applied to the model are used to convert assembly code into the descriptions of how the source codes are or are not vulnerable.

After significant training, the model demonstrates a very good ability to predict natural language descriptions from assembly code. The results show how the model was able to learn not only the dependencies between the input and the output, but also some of the meaning behind the words. This was observed when comparing the predictions from the model to their true descriptions, with one sample output predicting "non-negative" in the place of "larger than zero".

1.1 Motivation

A solution to the presented problem will be useful for a multitude of computing related topics that require reverse engineering. In particular, reverse engineering related to security aspects will benefit profusely in being able to understand how a program fulfills its purpose and what a program is capable of doing. Malware analysis requires a significant amount of reverse engineering to understand what the malicious software was designed to do. Producing human-readable summarizations of the malicious code will not only guide malware analysts, but allows the everyday human to better understand and grasp the concepts of how malicious software functions on their personal devices. Another security related aspect to benefit from this research is security adaptations for existing hardware and software. This is becoming a significant issue in the modern world with security measures often being built in after the fact and therefore requiring updates to ensure users are protected. Ease of understanding the binary code of technology with vulnerabilities will allow reverse engineers and program developers to produce results faster and potentially produce better solutions. These

improvements and modifications will be largely due to an increased understanding of what the computer does with a given piece of source code. Creating a model that will output how a piece of code is vulnerable is the first step in being able to tackle these problems on a larger scale.

1.2 Problem Overview

Assembly language is often confusing and difficult to interpret for those that are not, and even those that are, in the field of computing. The purpose of this research is to facilitate the analysis process of assembly code containing vulnerabilities. The work looks at creating assembly to text pairs that, when inputted to the model, are able to find patterns between the two and will output natural language descriptions of the codes functionality. The research will not focus on summarizing the source code or the binary code but will instead focus on the relationship between the two languages by summarizing how they are vulnerable.

1.3 Thesis Contributions

There are four primary contributions from this thesis.

- We define the assembly code summary generation problem, which is vital in the domain of reverse engineering for vulnerability analysis and malware studies. In the context of machine learning, we define this problem as a sequence-to-sequence learning problem that can be solved with data-driven solutions. We are among the first to study this problem.
- One of our key contributions is to create the first labeled dataset for binary

code summarization. Specifically, we compile the Juliet Test Suite and the NDSS18 vulnerable source code dataset and link them with their corresponding vulnerability descriptions. In total, we generated 97,492 unique pairs. The creation of the assembly-description pairs will be very beneficial to future work on developing an AI model that can truly understand the semantics of assembly code.

- With this research being at the forefront for this problem, this thesis presents one of the first feasible solutions in the area. We evaluated the proposed solution with two publicly available vulnerability datasets and show that it is able to adequately predict English sentence outputs for sections of assembly code.
- The explainable results adds another element to the list of contributions. With the attention mechanism, the model also highlights aspects of the inputs that are directly related to producing a correct prediction. This additional output helps the reverse engineers and security analysts to understand why a certain functional summary is generated.

1.4 Thesis Outline

The remainder of this thesis is organized as follows. In **Chapter 2** this report delves into the concepts of machine learning necessary for understanding how the model works and was created to solve the problem presented. **Chapter 3** covers related work in the field, the accomplishments achieved, and the gaps that exist surrounding the topic. The methodology followed is presented in **Chapter 4** and goes more in-depth into the concepts discussed in the Background section. **Chapter 5** discusses the individual experiments used to test the model and presents the results achieved

for the various experiments. Lastly, concluding remarks and research highlights are given in **Chapter 6**.

Chapter 2

Background

This chapter discusses the background knowledge necessary to fully understand the concepts used for this research. An understanding of the components that make up assembly code are important for grasping how the input is created. Natural language processing is discussed to explain the difficulties presented when computers attempt to produce natural language outputs. Different types of neural networks and their functionality are explored and are very important to understand as they make up the model created for solving the problem at hand.

2.1 Assembly Code

The two main components of this research revolve around computer language, referred to as assembly code, and natural language, in this case, English. Your ability to read this report means you are familiar with the essential concepts and meanings of the English language, but the same can not necessarily be said about assembly code. Assembly code is created by compiling source code written in a programming language. Every programming language, i.e. java, C, python, etc., has it's own compiler; a special program that processes the statements in programming languages

and turns them into machine language that a computer's processor can read.

This assembly code consists of instructions that tell the machine what to do and in what order based on the desired functionality of the written program. Different processor require different assembly architectures and therefore different assembly language instructions. The x86 assembly language architecture is used for this research. Instructions are made up of mnemonics and operands that are generally very simple tasks for the machine to complete. A few examples of simple instructions can be seen in Figure 2.1. The simplicity of individual instructions results in the entirety of the assembly code for a program to be quite vast and often very complex to analyze. This complexity is due to the assembly code being a direct representation of a possibly very complex program that has been broken down into thousands of individual steps.

Mnemonics are symbolic names for a single executable machine language instruction. For example, when adding two values, the mnemonic used for addition is simple ADD; when comparing two values, the mnemonic for comparison is CMP. The left-hand side of Figure 2.1 showcases different mnemonics. Operands follow the mnemonic and are the arguments or parameters that the executable instruction is applied to. With respect to assembly code, operands are generally registers or values. Registers are memory locations that store values in the computer processor. Values can be literal constants, a direct memory address in the computer, or a label representing a data item like a function or string. The right-hand side of Figure 2.1 are examples of various operands.

These concepts are important to understand because this research extracts the mnemonics and operands from the assembly code of compiled programs to use as

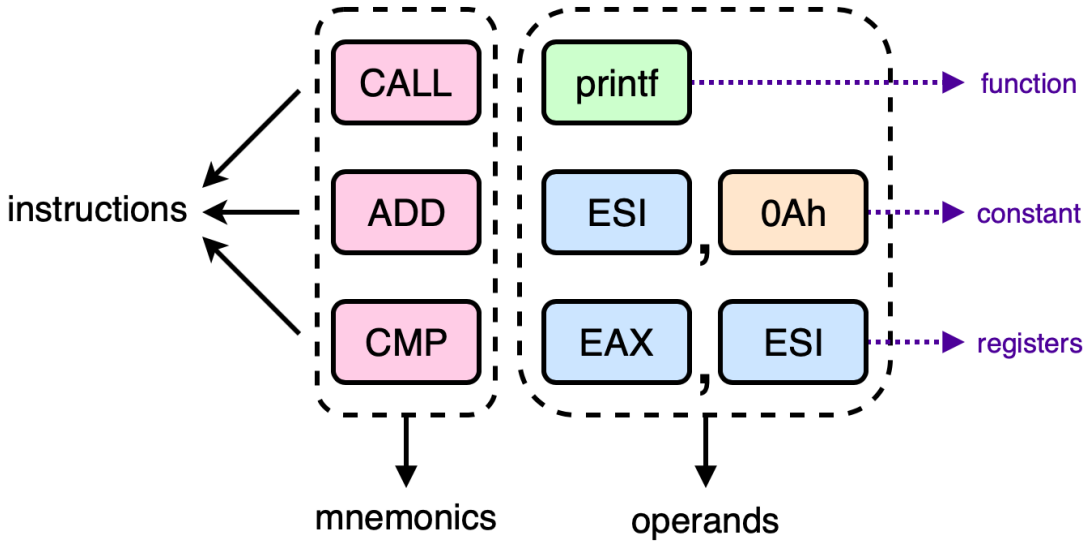

Figure 2.1: Three arbitrary assembly code instructions. The first is calling the function $printf$; the second is adding the hexadecimal value $0A$ to the value stored in register ESI; the third is comparing the values stored in registers EAX and ESI.

input in machine learning algorithms. The overall goal is to convert these machine instructions into human readable natural language descriptions, in this case, English. To produce a natural language output, it is important to understand how to best process languages, often considered a difficult task.

2.2 Natural Language Processing

Humans are great at using language, but not so great at understanding and describing the rules that govern language. Natural languages are discrete, compositional, and sparse, creating a challenge for computational approaches [15].

Languages are made up of words that are made up of characters. These characters are combined in different ways to represent objects, concepts, events, actions, or ideas. Different words are distinct from one another as the relationship between

words cannot be inferred simply by the characters that make them up. Characters or letters form words, and words form phrases and sentences, thus making language compositional. The meaning of a phrase or sentence is often larger than the individual words that comprise it, making it necessary to look at longer sequences of words such as paragraphs or entire texts [15]. The discrete and compositional nature of language leads to data sparseness. The possible combinations of words to create meaning in language is essentially infinite.

These challenges with natural languages makes it difficult to program computers to understand and produce language. Supervised machine learning is the best known method for dealing with language data [15]. Supervised learning attempts to learn the mapping between the given inputs and outputs through patterns and regularities in the data [15].

2.3 Sequence-to-Sequence Learning

Sequence-to-sequence learning is about training models to convert sequences from one domain into sequence in another domain. For example, sentences in English to sentences in French. This research aims to convert sections of assembly language into sentences in English.

The simplest form of sequence-to-sequence learning occurs when the input and output sequence have the same length as the input can be directly related to an output. In the general sequence-to-sequence case, the input sequences and output sequences have different lengths. When the lengths differ, the entire input sequence is needed to start predicting the target output. To combat the different lengths in the input and output sequences, an encoder and decoder are used to help find relations

within the data.

Encoder An encoder is a neural network that processes the variable-length input sequences to a fixed-length vector [9] [4]. The encoder generates encodings containing information about which parts of the input are relevant to each other through hidden states in the neural network. Each token in the input is read into the encoder one at a time in a sequential manner, updating the hidden state after each token [9]. The intermediate hidden states are discarded and a final output vector is produced. The resultant vector is often called the context vector and is supposed to be a good summary of the entire input sequence. For smaller sequences this summary will suffice, but as the length of the sequence increases, it becomes very difficult to summarize it in a single vector and performance of the model declines [4] [8]. The left-hand side of Figure 2.2 depicts the encoder model. Inputs are passed one at a time into the network with hidden states produced at each node. The final output of the encoder, the context vector, represents the connections and patterns found within the input data and is passed to the decoder as it's initial state.

Decoder A decoder does the opposite of an encoder, it maps the fixed-length vectors back into variable-length sequences [9]. The decoder is trained to generate an output sequence by predicting the next character in the target sequence given the hidden state of the encodings [9] [4]. There are two decoder arrays, the decoder input data and the decoder target data. The target data is offset by one time-step from the decoder input data because the model is trained to predict the next character of the target sequence given previous characters of the target sequence. The right-hand side of Figure 2.2 depicts the decoder model. The context vector is passed as the initial

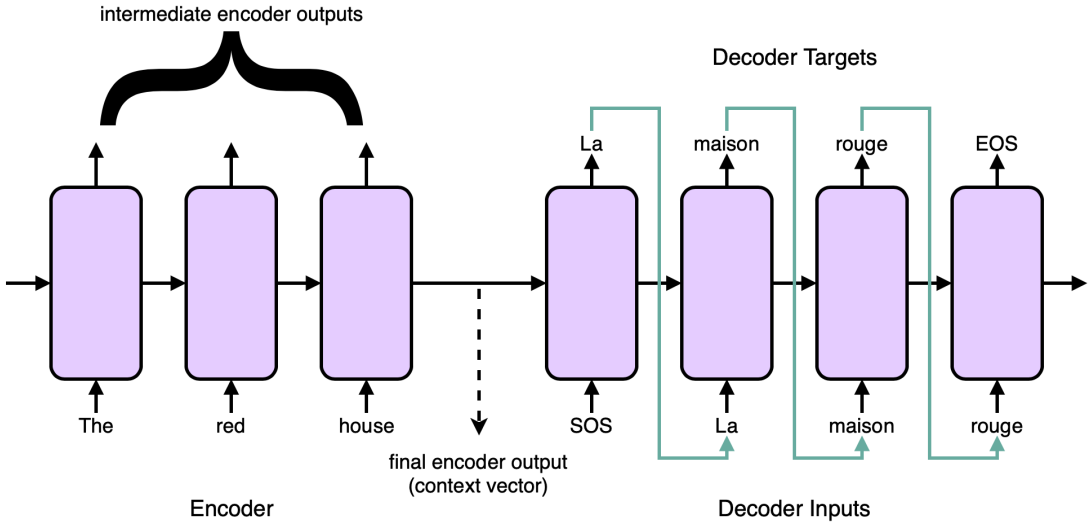

Figure 2.2: Encoder-Decoder model. Depending on the type of neural network used, the number of states passed through each node and the model overall may differ.

state of the decoder before it starts predicting outputs. Beginning with a Start-Of-Sequence token, the target data is offset by one-time step and produces the desired output until an End-Of-Sequence token is predicted.

The type of neural network used for the encoder and decoder is dependent on the type of problem being solved.

2.4 Neural Networks

Neural networks are learning techniques inspired by the way computation works in the brain [15]. They were originally developed as mathematical models to represent the way biological brains are capable of processing information [18]. The basic structure of a neural network consists of a series of small processing units, often called nodes, that are joined together by weighted connections [18]. The nodes can be related to

neurons in the brain, while the weighted connections represent the strength between any two neurons [18]. The network is activated by providing an input to one or more nodes that reverberates through the network via the weighted connections [18]. In relation to the biological brain, a poke in the finger will reverberate through the neurons in the hand and arm before spreading to the spinal cord and eventually the brain where a pain signal will be sent back to the finger. An activation function is used to determine which nodes/neurons will be "turned on" or activated and which will not. Given an input, the activation function defines the output for that node. These functions are tailored to the problem to achieve faster training processes and convergence. Neural networks have diverted from relations to the biological brain and are now seen as powerful computational tools used to solve complex problems.

Deep neural networks work by attempting to correctly represent past observations in order to make accurate future predictions [15]. Deep learning approaches feed data into a network that produces sequential transformations of the input data until a final transformation is produced [15]. This final transformation is used to predict the output. The transformations are learned during the training phase for the model where input and output mappings are uncovered. The more data that is transformed, the easier it becomes to related the input with the desired output. There are two major kinds of neural network architectures, feed-forward networks (FFNN) and recurrent/recursive networks (RNN) [15]. Figure 2.3 depicts simple diagrams for both an FFNN and RNN.

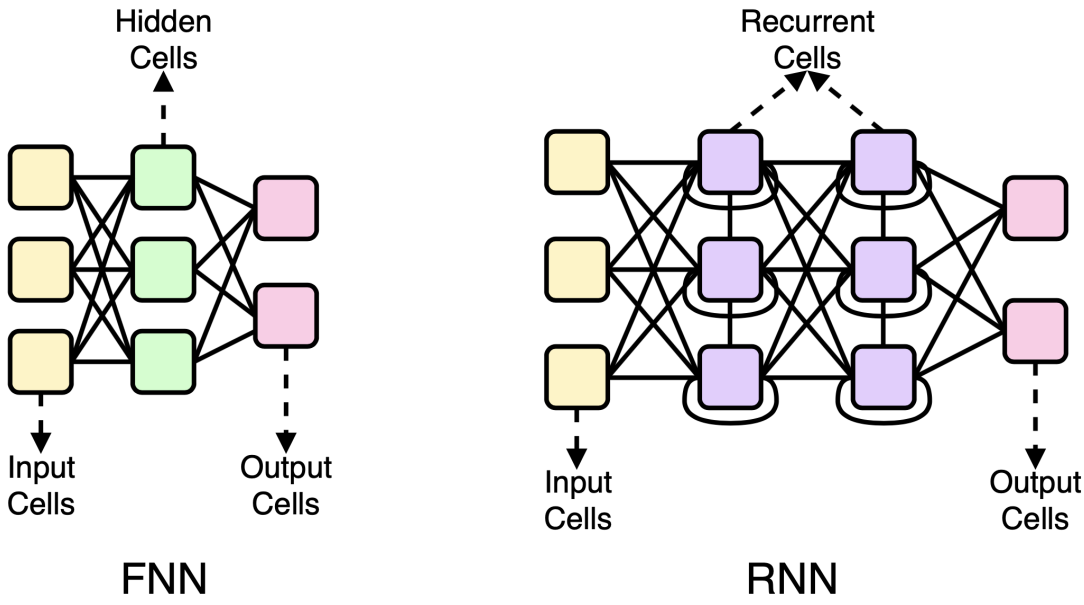

Figure 2.3: Simple diagram comparing feed-forward neural networks and recurrent neural networks. Information flow is from left to right in standard networks.

2.4.1 Feed-Forward Neural Networks (FFNN)

Feed-forward neural networks pass data forward from input layer, through any hidden layers, to the output layer [13] [18]. When using feed-forward networks in language processing, the probability used to predict the current word is based solely on the words directly before it [39]. FFNNs work with fixed sized inputs or variable lengths inputs whose order of elements can be disregarded [15]. These requirements make FFNNs useful for pattern recognition but unsuitable for many sequential data sets.

2.4.2 Recurrent Neural Networks (RNN)

Recurrent neural networks can, in general, map from the entire history of previous inputs to each output [18]. RNNs do this via feedback loops, where data that has

already been through the network can be fed back into the input for further processing [13]. This feedback loop acts as a kind of memory for the model to learn dependencies between the inputs and outputs [18]. The feedback loop is referred to as the hidden state for each node and is passed to each following node in addition to the input value. The hidden state is updated at each time-step. RNNs are specialized models used for sequential data and work by inputting sequences of data and outputting a fixed sized vector that summarises the sequence [15]. When dealing with natural language processing, sequence-to-sequence learning produces favorable results because languages are easily represented as sequences.

When dealing with sequential data, it can be beneficial to have information preserved from both past and future for predicting outputs when using sequential data. It is possible to obtain this information using bidirectional networks.

2.4.3 Bidirectional Networks

Bidirectional networks integrate both past and future inputs into its decision for predicting an output [15]. These models are composed of two separate recurrent neural networks that are connected to the same output layer [17] [12]. The forward layer output is calculated by reading the inputs in a positive sequence from time $t-n$ to $t-1$, while the backward layer output is calculated by reading the sequence in reversed order from time $t-1$ to $t-n$ [12] [4]. The hidden forward and backward states are combined in the encoding process to help with training the model to predict the target outputs. Many sequence labelling problems can benefit from having access to both future and past context of a given sequence [18]. A simple example of this would be when trying to classify a specific piece of text. Knowing the words or letters

that follow the text can be just as beneficial as knowing the words or letters that precede the text. Whether forwards or backwards, these dependencies are discovered during the training process of the model.

2.4.4 Backpropagation

Backpropagation is the network training algorithm. The algorithm adjusts the connection weights of a neural network based on the error rate/loss of the previous iteration. The combination of weights that minimize the chosen loss function is considered a solution to the problem. The backpropagation algorithm can fail with basic RNNs because as the network feeds forward the inputs from one layer to the next, the current weights can become increasingly smaller or larger from the beginning of the sequence. As the algorithm propagates backwards the loss will exponentially grow or vanish, respectively known as the exploding or vanishing gradient problem [20]. The vanishing gradient problem is more commonly seen in RNNs [11] as the weights at the start of a sequence become less and less important to the end of the sequence as it extends further. Although this is not always an accurate assumption, it is the way the network learns. To combat this problem, gated neural networks were developed.

2.4.5 Gated Neural Networks

Gated neural networks use gating mechanisms to control the flow of past information to the current state [31] by learning when to access memory and when to ignore it. The gating units modulate the flow of information inside each node and decide whether to keep the existing memory or discard it. Gated models use what are often refereed to as memory cells to remember values over time. These memory cells are fed information

via various gates within each node. Depending on the neural network used, a node can contain a combination of an input gate, output gate, forget gate, update gate, reset gate, etc. The gates regulate the information as the models is trained [18]. Not all gated networks use the same number of gates nor do the gates work in the same way for all models. Gated Recurrent Unit (GRU) neural networks and Long Short-Term Memory (LSTM) neural networks are two of the most common models that use gating mechanisms. LSTM and GRU neural networks are very powerful at capturing statistical regularities within sequential data [15], making them ideal for this problem.

Long Short-Term Memory (LSTM)

Long Short-Term Memory neural networks are composed of a cell state, a hidden state, an input gate, an output gate, and a forget gate [18]. The memory cell for an LSTM network is made up of the cell state, which is passed from node to node at each time-step [11]. The cell state is adjusted by the gates within each node as necessary as it moves through the network and makes up the long-term memory [25]. The hidden state is the output or activation for each node and is also adjusted by the gates before it is passed to the next node [11]. The hidden state acts as the short-term memory of an LSTM network [25]. All of the gates are given the previous node's hidden state and the input to the current node. These values are passed through sigmoid activation functions that output values between zero and one. This value is used as a multiplicative factor to activate certain information more than others depending on their level of importance to the solution [21]. A value closer to zero is indicates this information can be ignored, whereas a value closer to one indicates this information

should be remembered. Each gate is used to determine the importance of different information that will be passed onto the next node. A diagram of a simple LSTM can be seen in Figure 2.4.

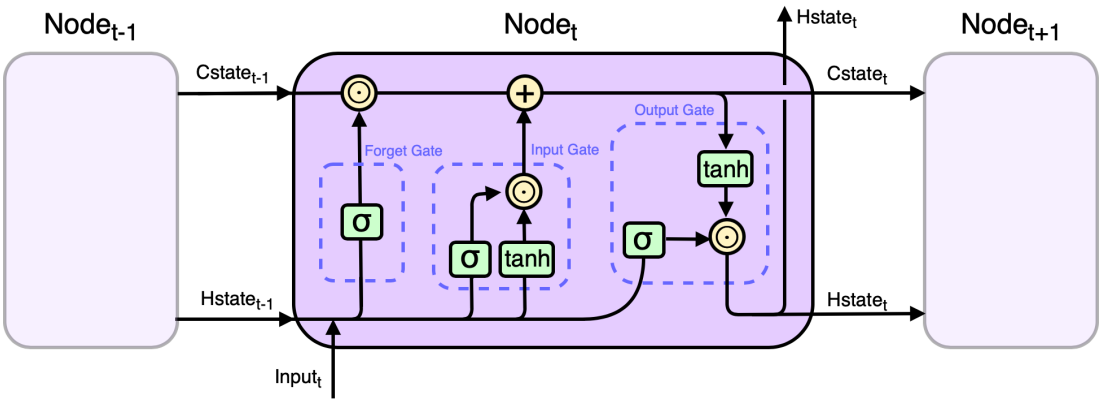

Figure 2.4: LSTM diagram of a single node. Circular operations '+' and '⊙' depict element-wise addition and multiplication. Rectangular operations represent activation functions: 'σ' represents the sigmoid function and 'tanh' represents the hyperbolic tangent function, which can be seen in Figure 2.5.

The forget gate decides what is relevant to keep from previous time steps. To decide this, the forget gate is applied to each value in the previous node's cell state. Information that is kept makes up part of the current node's cell state. The input gate decides what is relevant to add from the current time step. The input gate sigmoid function is applied to a vector of new candidate values that have been created using a tanh activation layer. This multiplication results in new candidate values that are scaled based on how much we've decided to update each value. Adding the information from the forget gate with the information from the input gate make up the current node's cell state that will be passed to the next node. The output gate is used to produce the hidden state that will be passed to the next node. The newly created cell state is passed through a tanh layer to normalize the values between

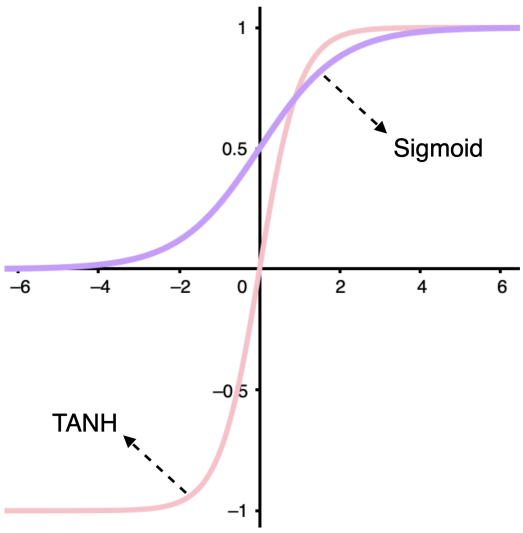

Figure 2.5: Activation functions sigmoid and hyperbolic tangent.

$$f(x) = tanh(x) \tag{2.1}$$

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

negative one and positive one. The resulting vector is multiplied by the output of the output gate's sigmoid function to adjust the weights of the output with regards to what is deemed important. This process is repeated for each node in the neural network.

The process of "remembering" or "forgetting" information at each node allows the model to learn what is important and to maintain those weighted connections at each time-step. GRUs were developed after LSTMs and are similar but use less gates within each node by combining some of those found in LSTM nodes.

Gated Recurrent Unit (GRU)

Gated Recurrent Unit neural networks work in a similar manner to LSTMs but combine states and gates to have overall less components. GRUs are composed of a hidden state, a reset gate, and an update gate [11]. The update gate decides what information from past time steps is discarded and what information is passed along [9],

in addition to determining what new information should be added [11]. The update gate acts as the memory cell of the unit, keeping track of information that is important and discarding information that isn't [11]. The reset gate is used to decide how much of the past information should be forgotten. When the reset gate is close to zero, the hidden state ignores the previous node's hidden state and is reset using the current input only [9]. The hidden units have separate reset and update gates and will therefore learn to capture dependencies over different time scales [9]. The reset gate will be frequently active in units that have learned to capture short-term dependencies and the update gate will be frequently active in units that have learned to capture long-term dependencies [9].

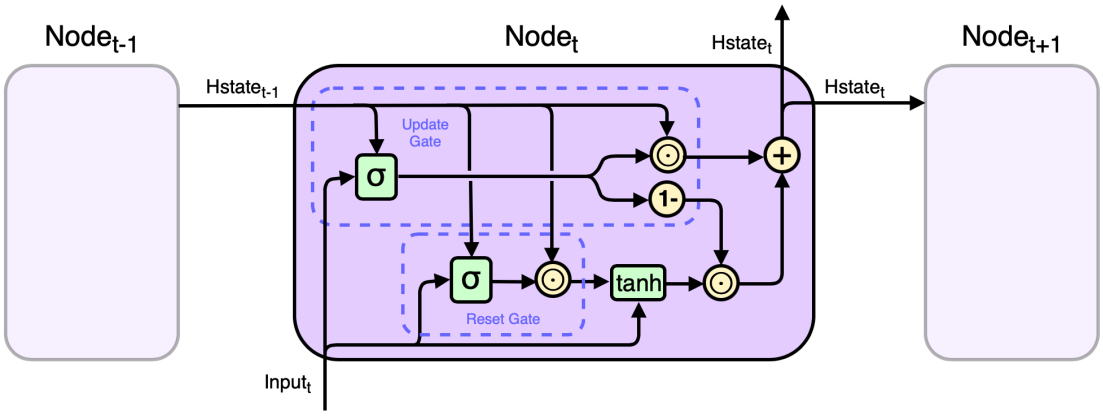

Figure 2.6: GRU diagram of a single node. Circular operations '+', '⊙', and '1-' depict element-wise operations. Rectangular operations represent activation functions: 'σ' represents the sigmoid function and 'tanh' represents the hyperbolic tangent function, which can be seen in Figure 2.5.

2.4.6 Attention Mechanism

A relatively newer concept introduced to neural networks is an attention mechanism. A model trained with attention determines what is more important from the input

with respect to output predictions and pays "attention" to those inputs more so than others in future iterations. When an attention mechanism is applied, the intermediate encoder states, represented in Figure 2.2, are utilized instead of discarded. Instead of encoding a whole input sequence into a single fixed-length vector, attention models encode input sequences into a series of vectors and chooses a subset of these vectors as the model decodes and predicts an output [4]. The model creates alignment scores that represent how well each encoded input matches the current output of the decoder. A softmax function is used to normalizes the scores so they can be treated as probabilities. The higher the score, the higher the probability for that input to be relevant to the current output. The context vector is produced by applying the probability weights to the encoded inputs, resulting in an attended context vector. The decoder uses the attended context vector for the current time-step to predict the current output. Attention models are useful when determining the "why" behind predicted output.

Discussing the components and difficulties of assembly code and natural languages is important to understand the processing requirements needed to prepare the data for the model. Considering the different neural network options for processing sequence-to-sequence learning and understanding which are most useful provides a comprehensive knowledge base for the experiments tested on the datasets. With an overview in the background of machine learning and neural networks in particular, it is possible to get more in-depth into the specifics used for the problem presented. The Methodology chapter of this thesis focuses on the processes and equations used to predict the final outputs.

Chapter 3

Literature Review

Little research has been conducted on aligning assembly code to natural language. Existing research focuses on the alignment of assembly code to source code and source code to natural language, respectively. The ability to skip the middle step involving source code will allow for quicker understanding of unknown programs, primarily for people that are not as well versed in the computer science world.

The summarization of assembly code into natural language touches on a few different fields of computing research. Data summarization research has given rise to many techniques and concepts that have proven to be particularly useful for summarizing texts and source code. This information allows for the alignment of source code to natural language for a variety of applications. Research into the analysis of binary code has proved useful for assembly code to source code alignment. This relationship is very important for malware analysis and similar tasks where functionality of a piece of code is unknown.

When discussing data summarization, it is also important to discuss data generation. The terms summarization and generation can often be used interchangeably as a summary is not always produced using only words from the input text, especially

with regards to code summarization. Generating method names from source code is considered a summarization of the code, but the output is considered a generated piece of text. An simple but important use for data generation is language translation; inputting text in one language with the goal of generating text in another language. A significant amount of code summarization revolves around the generation of method names and in line commenting based on the methods and functionality of the code. It is important the reader understands that these terms are used in conjunction in this research and that when discussing summarization of data we are also referring to the generation of data and vice versa.

The work accomplished by some of the researchers at the forefront of these developments are discussed in this chapter.

3.1 Data Summarization and Generation

Summarizing is the act of expressing the main ideas or most important facts about something in a short and clear form. There are two main types of summarization, extraction and abstraction. Extraction techniques extract words directly from the text that have been deemed significant and produce a summary by combining these important words [7]. Extraction summaries are generally not grammatically correct because of how they are formed. Abstraction methods use a similar technique but can produce additional words to give full sentence summaries that are grammatically correct [7]. Abstraction summaries produce more human readable results due to the added words.

Due to the complex nature of producing abstractive summaries, text summarization and code summarization are most often completed through extractive methods.

Abstractive techniques require significant use of natural language processors to ensure the resulting summary is written in proper English, or the origin language. Extractive techniques do not need this added processing because the summaries are created using only words found within the piece of text.

3.1.1 Text Summarization and Generation

With the introduction of the World Wide Web and the substantial increase in available and accessible information online, automatic methods for text summarizing has never been a more pertinent problem. This overload of information causes an issue as there is not enough time to read everything. Before the existence of computers, text summarization was performed manually. Although humans have an incredible ability to condense information, it can often be a very time consuming and tedious process to do manually. With the addition of technology and the advancements made in machine learning techniques, methods for producing summaries of text have come a long way. Applications for text summarization cover a broad range of topics that include producing headlines for news articles, outlines for student notes, minutes for meetings, previews for movies or TV shows, bibliographies for resumes or obituaries, etc.

Methods used to summarize pieces of text can be categorized into three main levels: surface-level, entity-level, and discourse-level. A surface-level approach uses features to represent information that are then combined to extract the important details about the piece of text. Features used in a surface-level approach include specified cue words and phrases; the location of words in the text, in a section, in a paragraph, etc.; background, which encompasses a user query or the presence

of terms from the title or headings within the text; and thematic features, which means terms that are statistically prominent in the field, based on term frequency statistics [32]. Entity-level approaches look to model the relationships in the text to represent patterns of connectivity. In doing so it is possible to identify what is most important. Some relationships to observe include similarity, for example vocabulary overlap seen within different sections of the text; proximity, meaning the distance between information in sections of text; the use of synonyms and other thesaurus type word commonalities; and logical relations like contradictions and consistency [32]. Lastly, discourse-level approaches attempt to model the whole structure of the text and what it is trying to communicate. Some examples of things to observe include the format of the document, different topic streams presented in the text, and the rhetorical structure of the text, meaning whether the text was intended to inform, persuade or motivate the reader [32].

As the years progressed, combinations of these approaches were explored to achieve better and more accurate results. Existing methods were primarily focused on extractive summaries given their reduced complexity when compared to abstractive methods, but this can often create incoherent results. With the advancements in machine learning and artificial intelligence, research into abstractive methods were given new life.

Abstraction approaches through artificial intelligence focus on natural language understanding in order to produce conceptual structures that represent the entire original text [19]. Much of the work done in this area focused on deletion-based sentence compression, meaning the removal of unnecessary words leaving only those

most important for representing the piece of text. Rush et al. proposed a fully data-driven approach that utilizes more paraphrasing, generalizing, and reordering of the text [36]. This differing view resulted in equivalent and better results than commonly used text summarization tools, proving that there is always room for improvements and new ideas.

The creation of text summaries using machine learning can be generalized down to inputting one piece of text and outputting another. Whether the input is a book, an article, a paragraph or a sentence, etc. with the output being paragraphs, sentences or individual words, etc., the goal is the same. A lot of older text summarization techniques revolved around analyzing different aspects of the text. In 2014, Sutskever et al. [40] and Cho et al. [9] introduced models that look more at the problem on a word-to-word or sequence-to-sequence basis. Their models used an encoder-decoder approach that tries to find direct relations between the input text and output so that future iterations of input would correctly predict the output [40] [9]. Both Sutskever et al. and Cho et al. had the goal of translating English to French using their encoder-decoder model trained to learn phrase representations and both achieved favorable results when compared to older models.

Although fewer new techniques are introduced, existing techniques become more widely applied to different problems related to texts. Tezcan et al. used a neural network architecture to detect grammatical errors in statistical machine translation [41]. Similarly and more recently, Muñoz-Valero et al. used recurrent neural networks to identify the subject and predicate of sentences [33]. Islam et al. applied an LSTM network to generate Bangla (Bengali language) text sequences, a difficult task given the limited resources in the Bangla field and the noise incurred during data [22].

Similar developments are being seen with respect to code summarizations as well.

3.1.2 Code Summarization and Generation

The primary goal of code summarization is to reduce simple functions and complex methods within a program to individual terms that, when combined, provide an accurate summary of the codes functionality. A variety of applications can benefit from modeling the relationship between source code and natural language. Some of these applications include code documentation, code generation, program comprehension, and software maintenance [3] [28]. Over the years, various methods have been tested in hopes of achieving effective summaries, but all have a common goal in explaining the source code in a clear and concise manner.

The comments included throughout source code are often a good means for producing an accurate summary of a codes purpose. In this regard, text summarization techniques are useful as comments are generally written in natural language. Due to potentially a multitude of reasons, software developers and programmers do not always include commenting in their code [38], leaving text summarization tools useless. This gap has given rise to automatic code commenting being a popular application of code summarization. The techniques utilized for this problem are essentially the same ones that are employed for standard source code summarizations. Song et al. observed that within the last decade, techniques for source code summarization have developed from information retrieval techniques to more deep neural network techniques [38]. This observation is proven true by Iyer et al., who created natural language summaries of C# and SQL source code using an LSTM model with attention [23]; or LeClair et al., who generate natural language summaries of program

subroutines using an attentional GRU encoder-decoder model [28].

This research supports the notion that developments are growing and making progress within the artificial intelligence realm. Extraction methods have remained at the forefront of code summarization as they require less computational power than abstractive summaries and are realistically not always needed for the comprehension of a program. This stands true for those in the field of computer science, but research into producing abstractive summaries would prove useful for those not in the field that are looking to understand all the same.

The more simplified approach to code summarization, adopted by many researchers, models it as a machine translation problem. The source or input is represented through a sequence of tokens in the code and the output or target being the natural language equivalent [3]. In doing so, it is then possible to apply a sequence-to-sequence neural network model to essentially convert the input into the output. Allamanis et al., Iyer et al., LeClair et al. and many others have successfully used this approach for their respective research. The machine translation-sequence to sequence neural network model is also the approach used for this research, making previous similar studies vital for understanding what worked and what could be improved on.

In particular, Allamanis et al. used a convolution neural network with attention to predict short and descriptive names for a given snippet of code [2]. The output produced by their model gives a statistical rating for a multitude of names that may be a good fit for the function. After observing the examples included in their research, it can be said that more often than not the name with the greatest percentage is the method name itself or a sub-string of the method name. Considering the programmer should create method names that describe the methods functionality, this is not a

surprise, but it must be kept in mind that these names could eventually be combined to give context for the entire source code. Although this method is promising and produces good results, the output lacks the simplicity needed for a broad range of individuals to understand the codes functionality.

A few years later, a more complicated approach to this problem was presented by Alon et al. Their approach utilizes the syntactic structure of programming languages by representing a piece of code as a set of compositional paths over its abstract syntax tree. An abstract syntax tree describes the data structure of a piece of code in a tree structure. The leaves of the tree generally refer to user-defined values that represent identifiers and names from the code, whereas non-leaves represent a restricted set of structures in the coding language such as loops and variable declarations [3]. Each path in the tree is compressed to a fixed-length where the model produces different weighted averages over the different path vectors. The result is multiple output tokens that can be used to summarize the snippet of code. Hundreds of paths are tested with the overall result being the top traveled path. The results of the research by Alon et al. outperform the baselines used within their study and the results of the research conducted by Allamanis et al [3].

Alon et al. take their research one step further by applying their model on short pieces of C# code in hopes of producing a full natural language sentence [3]. The dataset used is substantially smaller than code summarization datasets and therefore this method has been classified as code captioning instead. The model achieved a BLEU score of 23.04 (0.2304), improving on the current existing methods for similar tasks [3], but still leaving room for improvements. This research lacks a full dataset comparable to that of code summarization research, making it unclear how well this

model and others alike will perform on larger programs.

Code generation has been studied rather extensively since the rise of machine learning models. Generating methods names is a very useful application of code generation. Clear and concise method names are critical for the readability and maintainability of programs [24]. The research conducted by Allamanis et al. could be considered method name generation as their goal was to reduced a piece of code to a short and descriptive name [2]. Allamanis et al. did similar research a year prior that was specifically directed at predicting method names using their semantics in a neural probabilistic language model [1]. Gao et al. used the functional descriptions within source code to produce accurate method names for their research [14].

Taking code generation one step further, Wei et al. capitalized on the task of code summarization and trained a model to both summarize and generate code simultaneously [42]. Their goal for code generation went beyond method names by generating source code from natural language intent, meaning their purpose was to turn, for example, a description of requirements into a functional piece of code [42]. An encoder-decode architecture was first applied to code generation in this way by Ling et al. The encoder developed by Ling et al. took natural language descriptions as input and outputted source code through the decoder [30]. As advancements in the technological world continue to progress, innovations in this field and those of the like will also continue to progress.

3.2 Binary Analysis

Binary analysis is important for many applications, but it is particularly known as being very useful for malware analysis and patching software against known vulnerabilities [5] [37] [10]. Machine learning approaches to binary analysis have become popular due to their automated nature.

Xue et al. conducted a comprehensive study on machine learning-based analysis of program binaries that discusses many of the key aspects of machine learning algorithms and how they can be applied to a binary analysis problem [43]. A core aspect for successful binary analysis is determining the functionality of a section of instructions. Identifying this functionality gives rise to identifying a method or section of source code, as the analyst may not have access to the original source code. This aspect makes relating the assembly code to its source code counterpart an important field of research within binary analysis.

3.2.1 Relating Source Code to Assembly Instructions

Early work in the field attempted to directly decompile x86 assembly instructions to C code followed by full software analysis on the results [6]. The issue that arose in this situation was with control flow and how it depends on six status flags that are set, or not, dependent on the assembly instruction. Putting this into context, a simple add instruction in assembly cannot be directly converted to an add instruction in source code due to the side effects associated with the flags. The side effects are implicit, making it difficult to analyze control flow in the resulting source code [6]. To solve this problem, Brumley et al. explicitly encoded side effects into the generated source code, resulting in significant improvements to the success of the model. Both

models were deemed outdated as they were unable to handle bi-endian architectures, leaving way for a new iteration of the model to be created and tested. This new iteration divides the problem into two components, a front-end and a back-end. The front-end extracts binary code that fits into the supported architectures of the source code language [6]. Once the intermediate language has been created, the back-end implements the program analysis and verifications for the lower level aspects of source code [6]. Although successful, this method has since become relatively outdated due to advancements in machine learning and artificial intelligence and lacks the explanation of the resultant source code's functionality. As such, this model is often seen being used as a baseline in proceeding research.

Bao et al. improve on the models introduced by Brumley et al. by using a machine learning approach. The training set for the model consists of compiled source code references and their binaries with known start addresses. The algorithm creates a weighted prefix tree of start bytes for instruction sequences for functions [5]. This weighted tree is used during the testing phase to predict the start, and consequently the end, of functions within the assembly code. Their approach was successful in producing better results than previously tested models, including the commonly known disassembly tool IDA. Similar to the previous research discussed, this model is focused on relating the source code to the binary code, but no explanation as to functionality of the resulting code appears to be explored.

Neural networks have become a popular area of research that can extend across a vast array of domains, including binary analysis. Shin et al. propose a recurrent neural network (RNN) architecture that takes bytes of the binary as input and predicts, for each location, whether a function boundary is present or not [37]. Results

achieved are comparable or have better accuracy than prior tested models, but drastic reductions in computation time are observed when using the RNN model. A similar study done using RNNs looks at predicting function types from disassembled binary code functions [10]. The results from this research have proved comparable to previously tested models as well but utilize an architecture that resembles those successful in natural language processing problems, such as machine translation, automatic summarization, and sentence-generation [10]. This resemblance begs the question of whether the model can be extended to perform function type identification as well as an automatic summarization of the results, giving rise to some level of source code summary.

Combining the problems touched on in the previous research mentioned, Levy and Wolf propose an architecture for aligning the source code to its compiled binary equivalent in hopes of fully relating the two languages. The per-statement output from compilation is also dependent on other statements within the source code [29], making the task of aligning the two challenging. Two long short-term memory (LSTM) encoders are used to create representations of the source code statements and binary code statements respectively [29]. This is followed by a convolutional neural network (CNN) decoder to transform the statement pairs into alignment scores [29]. The network predicts pseudo-probabilities of aligning source code statements to binary code statements, taking the most probable element in each case as its proposed solution [29]. The results of the research are successful in predicting the alignment and were even found to outperform many existing literature baselines [29]. This research could be extended to summarizing the assembly code as well given function identification and other aspects of code identification within the binary code have already

been completed through this alignment process.

Katz et al. take on a similar approach to Levy and Wolf that uses recurrent neural networks. Katz et al. propose an LSTM-based RNN encoder-decoder model as a decompiler [26]. Binary machine code is used as input to the encoder with the goal of translating it into source code through the output of the decoder [26]. This approach is possible because RNNs are very good at handling sequence-to-sequence problems like language translation. English and French are considered natural languages where as binary and source code can be considered computer languages. My research aims to cross language barriers, per say, and translate a computer language into a natural language.

The research explored in this chapter covers text and source code summarization, text and source code generation, and different aspects surrounding binary analysis. Traditionally all time consuming manual tasks, the computer scientists discussed in this chapter were at the forefront of developing more automatic means for solving these problems. Neural networks have been shown to advance these fields over the years and will continue to do so as they are applied to more problems.

Chapter 4

Methodology

This chapter goes more in-depth into the concepts and networks introduced in the Background chapter that are directly related to experiments conducted to achieve results. The problem definition and how the data is represented are presented. The calculations for the different encoders are introduced followed by the information needed for the decoder to properly make predictions. The optimization techniques used for the experiments are explained. The chapter concludes with the different evaluations metrics tested to validate the results. Figure 4.1 gives a pictorial representation of the basic model schematic.

4.1 Problem Definition

The problem addresses the translation from assembly code to natural language. Assembly code is used as the input to the model where it is translated into natural language outputs that describe what the code does with regards to the vulnerability for that file. Equation 4.1 defines the problem.

$$x \rightarrow E \rightarrow CV \rightarrow D \rightarrow y \tag{4.1}$$

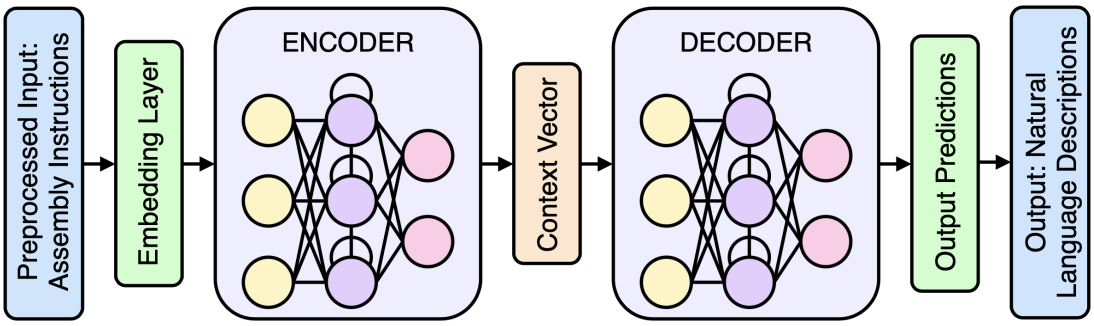

Figure 4.1: Preprocessed input passes through an embedding layer before being given to the encoder. The outputted context vector is passed to the decoder where output predictions are made and converted into natural language descriptions. The encoder and decoder networks are either LSTM, GRU, or bidirectional models.

where x is the assembly code input, E represents the encoder equations, CV is the context vector, D represents the decoder equations, and y is the natural language output. Further explanation into the encoder, decoder and context vector equations are discussed.

4.2 Data Representation

The input sequence is a concatenation of assembly code instructions. The output sequence is a natural language description that has been extracted from the source code file header. Both the input and output sequences start out as lists of strings and must be converted into a usable format for the model to learn from. The strings are converted to integers and padded as necessary to ensure all samples are the same length before being passed to the encoder.

4.3 Encoding the Assembly Code Sequence

The encoding process takes the input token-by-token and finds relevant relations between them that can be used to make accurate predictions for the model. Although the basic process is the same, different neural network models are used as the encoder which use different methods to find these relations. All neural network produce a context vector that will be passed to the decoder as its initial state.

4.3.1 LSTM as the Encoder

LSTM neural networks were used as they prevent important information from being lost while the model is trained, ensuring more accurate predictions overall. LSTM models preserve long-term information from past inputs using its cell state and short-term information from past inputs using its hidden state. The following set of equations are used to determine the cell state and hidden state for each LSTM unit. For the equations, W_* denotes the weight matrices for the respective variable, x_t represents the input at the current time-step, h_{t-1} is the previous time-step's hidden state, σ represents the logistic sigmoid function, tanh represents the hyperbolic tangent function, and b_* are the biases. Element-wise or pointwise multiplication is depicted by $\odot$; all other operations are assumed pointwise unless otherwise stated. The forget gate, input gate, output gate, cell state and hidden state are calculated using Eq.4.2 to Eq.4.7 below. A pictorial representation of these equations can be seen in the encoder-decoder model in Figure 4.2.

$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f) \tag{4.2}$$

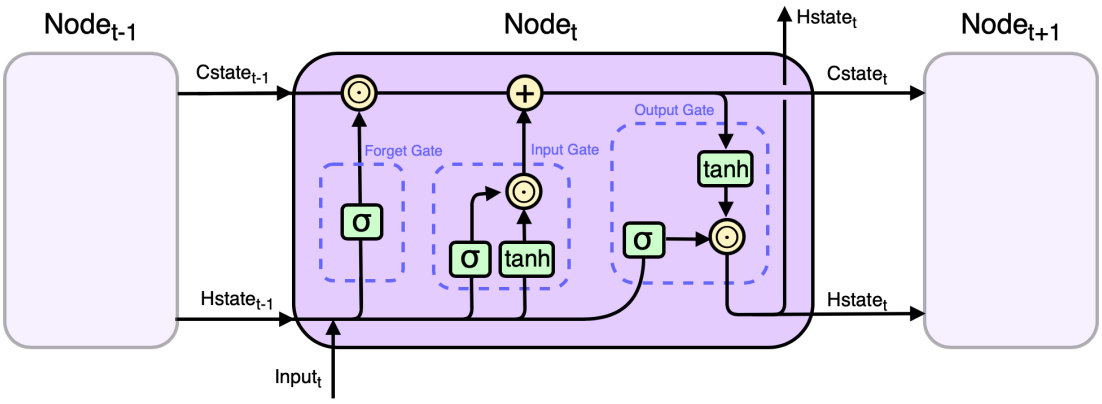

Figure 4.2: LSTM diagram showing the inner workings of a node.

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i) \tag{4.3}$$

$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o) \tag{4.4}$$

$$c'_t = \tanh(W_{xc'}x_t + W_{hc'}h_{t-1} + b_{c'}) \tag{4.5}$$

where c'_t are new candidate values that are scaled based on how much the model decides to update them in the following equation:

$$c_t = c_{t-1} \odot f_t + i_t \odot c'_t \tag{4.6}$$

$$h_t^{LSTM} = \tanh(c_t) \odot o_t \tag{4.7}$$

The final hidden state and cell state are passed to the decoder as initial inputs. The decoder is also an LSTM network meaning it functions in the same way, but is predicting and passing outputs through the network rather than states. An End-Of-Sentence token is passed to declare the end of that sequence. The model repeats this process with the remaining sequences in the dataset.

4.3.2 GRU as the Encoder

GRU neural networks work in a similar fashion as LSTMs but reduce the number of gates used and only pass a hidden state through the network, eliminating the cell state altogether. The information from the LSTM cell state is incorporated in the hidden state through the gates used in a GRU network. The following set of equations is used to calculate the hidden state for each unit of the network. The equations can be seen visually in Figure 4.3.

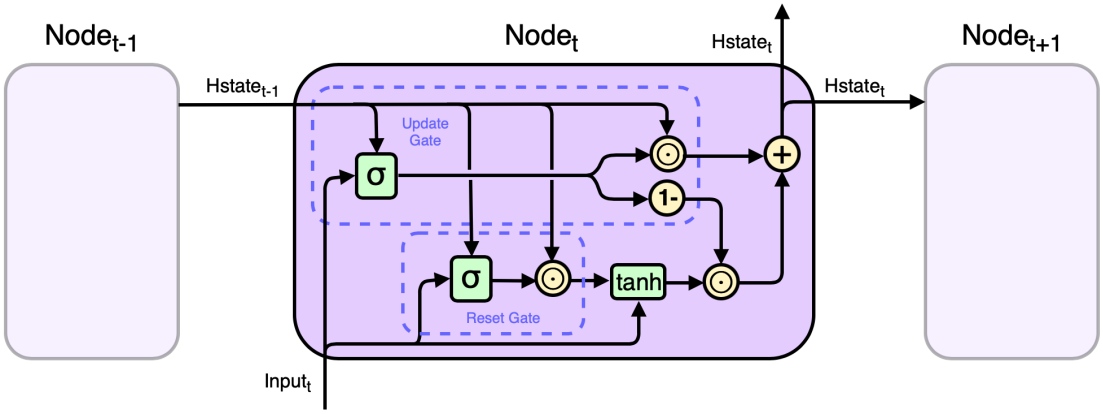

Figure 4.3: GRU diagram showing the inner workings of a node.

The same equation notation is used as the LSTM neural network equations above. The update gate, reset gate, and hidden state are calculated using Eq.4.8 to Eq.4.11.

$$u_t = \sigma(W_{xu}x_t + W_{hu}h_{t-1} + b_u) \tag{4.8}$$

$$r_t = \sigma(W_{xr}x_t + W_{hr}h_{t-1} + b_r) \tag{4.9}$$

$$h'_t = \tanh(W_{xh'}x_t + W_{hh'}(r_t \odot h_{t-1}) + b_{h'}) \tag{4.10}$$

where h'_t is new memory content that uses the reset gate to store relevant information

from the past.

$$h_t^{GRU} = u_t \odot h_{t-1} + (1 - u_t) \odot h'_t \tag{4.11}$$

With GRUs, only the final hidden state is passed to the decoder as its initial input. The decoder is modeled as a GRU network, passing and predicting outputs until an EOS token is found, declaring the end of that sequence. The model repeats this process until there are no more remaining sequences in the dataset.

4.3.3 Bidirectional Models

Bidirectional models were also tested given they take into account information from past to future and from future to past by combining the hidden states from both directions. In doing so, the model is preserving information from both past and future inputs [15]. A bidirectional network is composed of two separate recurrent neural networks, one given inputs in the positive forward direction and the other given the reverse of those inputs [12]. Figure /reffig.bidirectional shows a simple pictorial representation of a bidirectional model. The forward and backward nodes are either LSTM or GRU neural networks.

The outputs from the forward network and the backward network are connected to the same output layer, making the model a cohesive unit. The forward layer output sequence and backward layer output sequence are respectively denoted by $\overrightarrow{h_*}$ and $\overleftarrow{h_*}$, where h represents LSTM or GRU blocks calculated using Eq.4.2 to Eq.4.7 and Eq.4.8 to Eq.4.11 respectively, and $*$ represents time-steps $t - n$ to $t - 1$. Combining these output sequences gives the following equation as the context vector for the encoder,

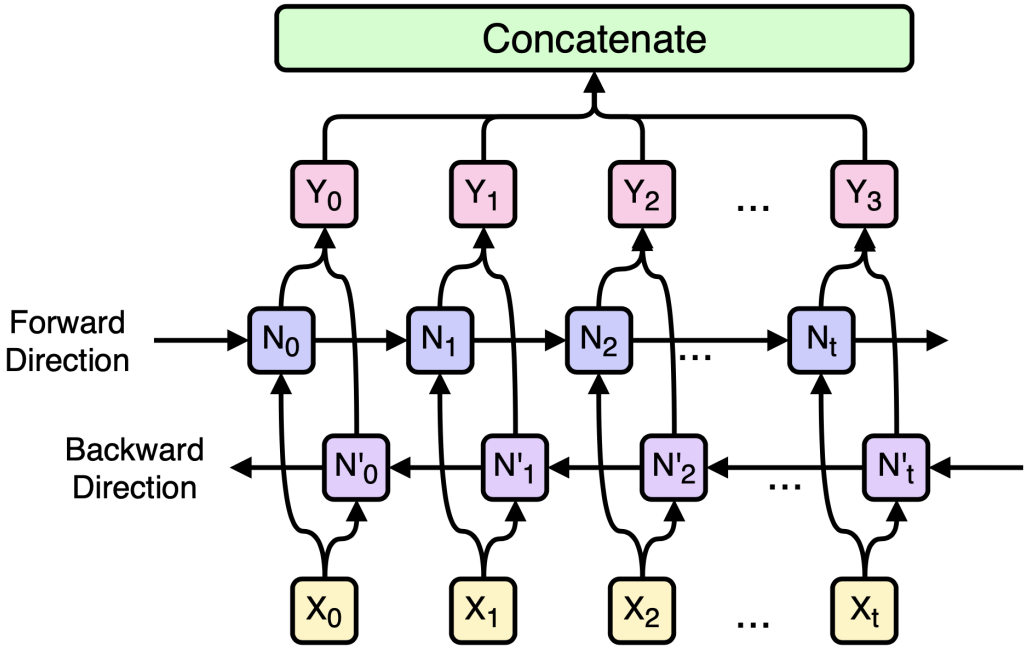

Figure 4.4: Pictorial representation of a simple bidirectional model. The input, x, is passed to the nodes, N, which are either LSTM or GRU networks. The outputs, y, are combined and concatenated.

CV^{Bi}, for the model at time t:

$$CV_t^{Bi} = F(\overrightarrow{h_{*(t)}}, \overleftarrow{h_{*(t)}}) \tag{4.12}$$

where F is the concatenation function used to combine the two output sequences.

4.4 Decoding Natural Language

The decoding process takes the final encodings from the encoder and uses them as the initial hidden state to the first unit of the decoder. The remaining units of the decoder use the hidden state from the previous decoder unit. Apart from the initial decoder input, all other inputs to decoder units are the previous units output, as

shown in Figure 2.2.

4.4.1 LSTM/GRU-based Context Representation

The context vector passed to the decoder as the initial state is dependent on the model used. Because the internal states differ, we must use LSTM or GRU for both the encoder and decoder respectively. Eq 4.13 to Eq 4.15 show the context vector equations.

$$CV_{LSTM} = F(h_f^{LSTM}, c_f) \tag{4.13}$$

where h_f and c_f are the final states of the encoder and F is the function used to concatenate the two states.

$$CV_{GRU} = h_f^{GRU} \tag{4.14}$$

where h_f is the final hidden state from the encoder.

$$CV_{Bi} = F(\overrightarrow{CV_*}, \overleftarrow{CV_*}) \tag{4.15}$$

where F is the concatenation function used to combine the two output sequences and CV_* is either CV_{LSTM} or CV_{GRU}.

The context vector is inputted to the decoder and the output, y, from the decoder, D, is calculated as follows:

$$y_t = D(h_{d(t-1)}, y_{t-1}) \tag{4.16}$$

where,

$$y_{t-1} = \begin{cases} y_{t-1}, & \text{if } t \geq 1 \\ SOS, & \text{otherwise} \end{cases} \tag{4.17}$$

and,

$$h_{d(t-1)} = \begin{cases} h_{d(t-1)}, & \text{if } t \geq 1 \\ CV_*, & \text{otherwise} \end{cases} \tag{4.18}$$

where $*$ represents the model being used.

4.4.2 Attention-based Context Representation

An attention mechanism was added to the best performing model, not only as an attempt for improvement, but also to determine the leading factors behind predicting the output. The bidirectional GRU model gave the overall best results so the attention mechanism was added to that network. It is important to determine which assembly code instructions and operands were used to predict the output so that it can be confirmed that the model is working based off of learned dependencies that make sense in the real world. The alignment scores, e, for mapping encoded input i to the current output at time-step t is calculated using Eq.4.19.

$$e_{t,i} = a(s_{t-1}, h_i) \tag{4.19}$$

where s_{t-1} is the previous hidden state of the decoder, h_i is the i-th hidden state of the encoder, and a is the alignment model used.

Bahdanau et al. [4] used this approach to calculate scores using a feed-forward neural network alignment model. When the lengths of the encoder sequences and the decoder sequences differ, this approach becomes less efficient as we must run through the network $decoder_{length} * encoder_{length}$ times to acquire all the attention scores. A more efficient model projects s and h onto a common space and applies a similarity

measure, in our case the dot product, to calculate the attention scores. The two projection vectors are called query, q, for decoder states, and key, k, for the encoder states. The query vector is equal to the decoder outputs and the key vector is equal to the encoder outputs, both are calculated using the GRU Eq.4.8 to Eq.4.11. These adjustments turn Eq.4.19 into Eq.4.20.

$$e_{t,i} = q_t \cdot k_i^T \tag{4.20}$$

where k^T is the transpose of k, needed to satisfy matrix dimension multiplication requirements.

The softmax function applied to the scores gives Eq.4.21. The outputs are used as the weights when computing the context vector.

$$\alpha_{t,i} = \frac{exp(e_{t,i})}{\sum_{k=1}^{n} exp(e_{i,k})} \tag{4.21}$$

The context vector for the output at the current time-step, y_t, is the sum of the weights multiplied by the encoders hidden states. In an attention model, the encoder hidden states are denoted as the value matrix, v. Therefore, the context vector at time-step t is denoted by Eq.4.22.

$$CV_t = \sum_{i=1}^{n} \alpha_{t,i} v_i \tag{4.22}$$

The context vectors are passed to the decoder and used on a per-output basis given each target output will have a different context vector. The alignment scores used to create the context vectors are obtained and studied to determine the leading input

factors used when predicting each output. This final experiment provides information on all others as attention looks into explaining how the model makes its choices for output. While attention provides insight into the inner workings of the model, it is also important to evaluate the output from the model.

4.5 Optimization

Optimization for neural networks use gradient decent algorithms to minimize or maximize an objective function. There are three main types of gradient decent algorithms, batch gradient decent, stochastic gradient decent, and mini-batch gradient decent. Batch gradient decent computes the gradient of the objective function for the entire dataset for each epoch before determining how to update it's parameters for optimization [35]. This method of gradient decent is very slow as it must work through the entire dataset for each epoch before computing the gradient. Stochastic gradient decent computes the gradient and updates the parameters for each training example [35]. This gradient decent method can cause significant fluctuations in the objective function because training example can differ greatly. Mini-batch gradient decent combines the previous two algorithms and computes the gradient decent over mini-batches of data within each iteration of training [35]. This method reduces the variances seen with stochastic gradient decent while maintaining a faster convergence than batch gradient decent. With this being said, the larger the batch size is the more accurate the estimate is for the gradient of the batch resulting in a faster convergence.

4.5.1 Gradient Decent Algorithm

Root Mean Squared prop (RMSprop) is the gradient decent algorithm used for training the model due to its adaptive learning ability and its use of mini-batches. The learning rate of a gradient decent algorithm follows the slope of the objective function downwards until a local minimum is found. The magnitude of gradients can differ for different weights and can change during learning which makes it difficult to choose an overall learning rate during training. RMSprop keeps a constantly moving average of the squared gradient and adjusts the weight updates appropriately based on the average as the algorithm progresses. This moving average means a learning rate does not need to be specified as it is constantly changing and being adjusted, making it an adaptive learning rate. Another benefit of RMSprop is that each parameter has its own adaptive learning rate, meaning each parameter will learn at a rate that optimizes its individual loss, thus reducing the loss as a whole.

4.5.2 Objective Function

The objective function being minimized is the cross-entropy. Due to the data being sparse, sparse categorical cross-entropy is the official objective function used. Cross-entropy measures the distance between the outputted probabilities calculated by the softmax function and the truth values. For example, if the next target token is “data” and based on the softmax function, “data” as the next predicted output has a 70% probability, then there is a 30% loss where the model could predict the wrong token. The model weights are continuously adjusted to reduce this loss during the training process so that predicted outputs are the best possible target predictions based on the statistics. To optimize the model it must know when to stop training so that

overfitting does not occur, which is where early stopping is introduced.

4.5.3 Early Stopping

Early stopping is set by choosing an output parameter of the model to minimize or maximize. In terms of this research, output parameters used for early stopping included the loss, the sparse categorical accuracy or adjusted accuracy as described in the next section, the loss on the validation set, or the sparse categorical accuracy on the validation set.

To optimize training, early stopping on the loss parameters are minimized and early stopping on the accuracy parameters are maximized. There were allowed five epochs of increased loss values from the minimum or five epochs of decreased accuracy values from the maximum before the model would stop. This level of patience was to ensure minimal optimization loss occurred in order to maximize the training process. All experiments underwent early stopping for each output parameter in order to find the best model and most accurate early stopping technique for this dataset.

The goal of optimizing the training process is to increase the likelihood that the model will make correct predictions. Another goal of optimization is to increase the metrics used during evaluation of the model. The better the model is optimized, the better the evaluation results should be.

4.6 Evaluation Metrics

Evaluation metrics provide a quantifiable way to determine how well the model can predict the desired output. There are many evaluation metrics that exist, but not all

are relevant to the research conducted. There are four main metrics used for evaluation of the results that will be discussed in detail, Bilingual Evaluation Understudy (BLEU), accuracy, recall, and precision.

4.6.1 Bilingual Evaluation Understudy Score (BLEU)

When dealing with natural language data, it is important to evaluate how readable each predicted sample is based on how well it was translated. The Bilingual Evaluation Understudy Score was designed to quantify translation performance [34] by evaluating a generated sentence to a reference sentence. The score ranges from 0 to 1, a score of 1 being a perfect match and a score of 0 being a perfect mismatch. The closer a machine translation is to a professional human translation, the better it is [34], resulting in a score closer to 1.

The idea of the BLEU score is to compare predicted sequences to a set of references sequences on a per token basis where position is deemed negligible. This comparison is adjusted to limit the number of tokens that can be flagged as correct for repeating tokens in a reference sequence. For example, if a reference sequence contains the word "the" three times and a the predicted sequence being evaluated contains the word "the" five times, only three of those can be flagged as correctly matching tokens. A benefit of taking into account the occurrence of words in the reference sentence is that the BLEU score will naturally decrease for a prediction length greater than all reference sentence lengths.

It is important to note that an increased number of reference sequences per translation will result in a higher score, as there are more options for the prediction sequence to match correctly. This makes it difficult to compare BLEU scores with other

research studies because the chance of using the exact same number of reference sentences is very slim. Papineni et al recorded a BLEU score of 0.3468 on a test set of 500 sentences using four references and a BLEU score of 0.2571 when using the same dataset with only two references [34]. The full set of extracted descriptions are inputted as the reference sequences when calculating the BLEU score, resulting in just over 20,000 possible reference sequences used for matching per generated sequence. The BLEU score is supported by more common neural network metrics like accuracy.

4.6.2 Accuracy

Accuracy is measured by comparing the true natural language description extracted from the source code with their corresponding predicted outputs of the model. The more truth-prediction pairs that match, the more accurate the model is. Although a quantifiably accurate model is ideal, the prediction does not necessarily have to match the true description exactly for it to be considered accurate. There are multiple ways to say the same thing, for example: "a number greater than zero" means the same thing as "a non-negative number", which also means the same thing as "a positive number". A prediction that maintains the underlying meaning of the original description is considered just as accurate, perhaps even more so considering it shows that the model is actually learning and not just replicating based off of training. Accuracy of the model was calculated with two sets of results. The difference in the decoder predictions in terms of how the accuracy is determined is seen in Figure 4.5

Adjusted Accuracy

The adjusted accuracy is used to determine if the model is capable of predicting the correct output given the correct input. The decoder inputs are adjusted to always pass the correct expected input for that specific target sequence. Decoder outputs that don't match the expected result are not used as the input for the following word prediction. The left-hand side of Figure 4.5 demonstrates that the model has predicted *musique* instead of *maison*. Where *musique* should be passed to the decoder as input, *maison*, the correct input, is passed instead. The accuracy is then calculated by comparing the target output, "*La maison rouge.*", with the actual output, "*La musique rouge.*". This method gives an over-estimated accuracy value for model, as we may not have outputted *rouge* without the *maison* input.

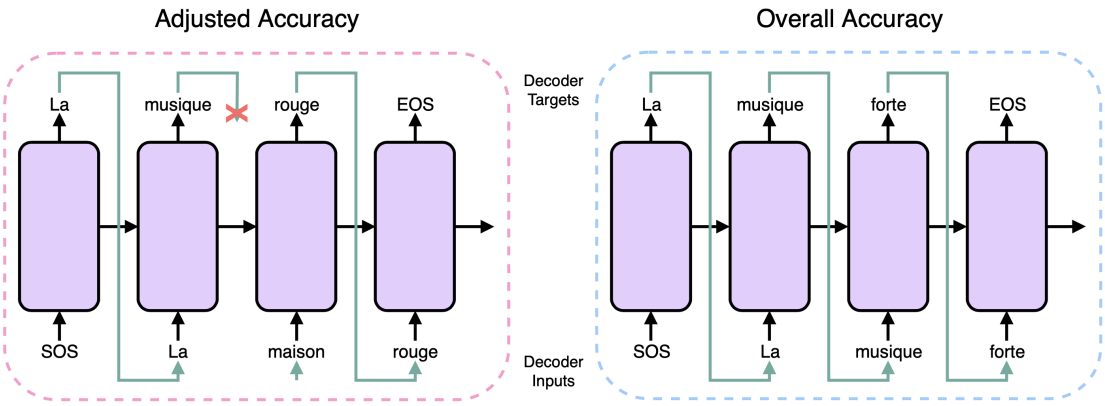

Figure 4.5: Demonstrating the differences between how the adjusted accuracy and the overall accuracy are determined.

Overall Accuracy

The overall accuracy computes how accurate the model is performing on it's own. Whatever the output is from the decoder, it is passed as the next input to the decoder.

The right-hand side of Figure 4.5 shows the target outputs being passed back to the decoder as inputs until EOS token is achieved. The accuracy is then calculated by comparing the target output, "*La maison rouge*.", with the actual output, "*La musique forte*.". This method gives a more realistic accuracy value for the model.

4.6.3 Recall and Precision

Where accuracy looks at the final predicted output as a whole and compares it to expected output, precision and recall look at the individual words. Precision and recall are essentially accuracy measurements on a word-to-word basis for this dataset. Given a correct input, can the model predict the correct target.

Recall

Recall is the percentage of correct inputs that produced correct outputs. Recall is also a good metric to assess grammatical correctness of the predicted outputs. The model is trained on grammatically coherent sentences so the natural flow of a sentence should be predicted by the model if training was a success.

A correctly predicted output token does not have to be the correct target for the current sequence. Any target that is found to follow the input from one of the extracted descriptions is considered correct when calculating recall. With this being said, a high recall does not necessarily mean the overall prediction is correct. Using Figure 4.5 as example, if the model is given "La" as input, the output of "musique" will be identified as a correct output because that is an accurate word to follow "La" in a French sentence. Recall is assessing if the model will output "musique", another target that has been identified as a correct output for "La", or if the model will

output nonsense. Only the latter will result in a reduction of recall. Continuing with this example, given the correct input of "musique", the model could predict the output "forte" or "doux". Both are correct answers to follow an input of "musique", but both cannot be the input the model should be predicting based on the sequence being assessed. Precision is the metric used to distinguish between the two.

Precision

Given the correct input, precision is the percentage of outputs identified as correct that are actually correct in terms of the true output. It aims to answer the question, is the output correct for the prediction we were assessing? Going back to the example displayed in Figure 4.5; although "musique" is a correct output following "La" as an input, it is not the correct target the model should have given with respect to this sequence. The correct target the model should have predicted was "maison" following the input "La". Continuing with this example, if "musique" was the appropriate target output, when passed as input in the following time-step, will the model output "forte" or "doux", both correct outputs, but only one will match the meaning of the sentence being translated. Most descriptions in the dataset are unique given they span a wide array of vulnerabilities that stem from different issues in the code. This distinction allows us to conclude that the higher the precision is for tokens at the beginning of a sentence, the higher chance the model will give an overall correct prediction for that sample.

This chapter focuses on the lower level components of the network introduced. The calculations needed to build the model's encoder and decoder are presented. Following this are the optimization methods used to ensure the model is working to

the best of its ability. Concluding the chapter are the evaluation metrics applied to test how well each experiment can predict natural language descriptions.

Chapter 5

Experimental Evaluation

Neural network models are used as the encoder and decoder units. Recurrent neural networks are tested due to their increased ability to predict sequential data. Gated RNNs are favored to eliminate any problems when calculating the gradient during backpropagation. In particular, long short-term memory (LSTM) and gated recurrent unit (GRU) networks are the primary models for experimentation due to their favorable results when dealing with natural language processing. Bidirectional aspects are added to both models with the goal of improving evaluation metrics. Finally, an attention mechanism is added to the best performing model as the last experiment so that we are able to understand the dependencies being created during the training process.

5.1 Dataset

The Juliet Test Suite is the primary dataset used, unless stated otherwise, when referring to the dataset in this report we are referring to the Juliet Test Suite. A secondary dataset, the NDSS18 dataset, is also tested to ensure the model is not biased to a specific set of data; when referring to this dataset in this report it will

be stated. Both datasets are vulnerability datasets, meaning all of the code samples contain some sort of software vulnerability. Some of the vulnerabilities present have to do with process control, stack-based buffer overflow, information leakage, uncaught exceptions, and divide by zero. The extraction of the necessary information from each file in the datasets is done in the same way because both sets of data are formatted the same.

The Juliet Test Suite and NDSS18 datasets consist of source codes samples which have been complied into assembly language files. These files can be divided into good cases and bad cases. The good cases have eliminated the vulnerabilities whereas the bad cases showcase the software vulnerability for that sample. There are multiple assembly code files for each source code sample that are organized based on the type of vulnerability and how it is implemented. Each source code file includes a header explaining why it would be "good" or "bad". The file name for each source file matches that of its corresponding assembly file so relating the two files can be done easily.

A lot of time is spent looking for vulnerabilities in source code. Discovering vulnerabilities is important for software maintenance, software analysis, and software development. This dataset consists of 97,492 usable source code files highlighting their vulnerabilities. Compiling these files using GNU Compiler Collection gives their equivalent x86 assembly code with contained vulnerabilities. The Juliet Test Suite is one of the first datasets that has the ability to map assembly code to vulnerabilities in the source code. The NDSS18 dataset was introduced in 2019 and is a relatively new contribution to the field [27]. With the goal of this research being to map assembly code to natural language, these are ideal datasets to use.

5.1.1 Data Extraction

Extraction of the data is simplified by the commonalities found within the files of both datasets. The naming convention for all files, source code and assembly code, indicate whether that particular file is a good case or a bad case. This distinction allows for the matching of good and bad assembly code files to good and bad source code files, respectively. In addition to this, the beginning of each assembly code filename matches that of the related source code filenames. Splitting the assembly code filename to include only the common aspects allows for all corresponding source code files relevant to that specific vulnerability and implementation to be found. With an understanding of what files are correlated, it is possible to extract the necessary information needed for the model to learn the relations.

Assembly code instruction are used as input to the model. To convert the assembly code into usable input for the model, we must first compile the source code. Compilation is achieved using the GNU Compiler Collection (GCC), which translates source code into x86 assembly code. Following compilation, the assembly code must be disassembled to identify the different blocks of code. Within the blocks of code it is then possible to identify individual instructions, followed by the individual mnemonics and operands. The IDA Pro Disassembler was used to disassemble the machine language and organize it into sections capable for extraction. Once the assembly code is organized, it is possible to iterate through each block of code, extracting the mnemonics and operands within each instruction. The assembly instructions associated with one source code file are combined into a string of tokens. This string of tokens is related to the description extracted from the source code file used to create these assembly instructions.

Natural language descriptions are used as the output of the model. Each source code files contains a header providing information on the vulnerability. The header contains descriptions for *BadSource*, *GoodSource*, *BadSink*, and *GoodSink*. The description for *GoodSource* and *GoodSink* are combined to make up the description for the *good* assembly file associated with this source code. The same thing is done for the *BadSource* and *BadSink* descriptions; they are combined and used for the *bad* assembly file associated with this source code. Figure 5.1 shows an example of the format for each header including only the relevant descriptions.

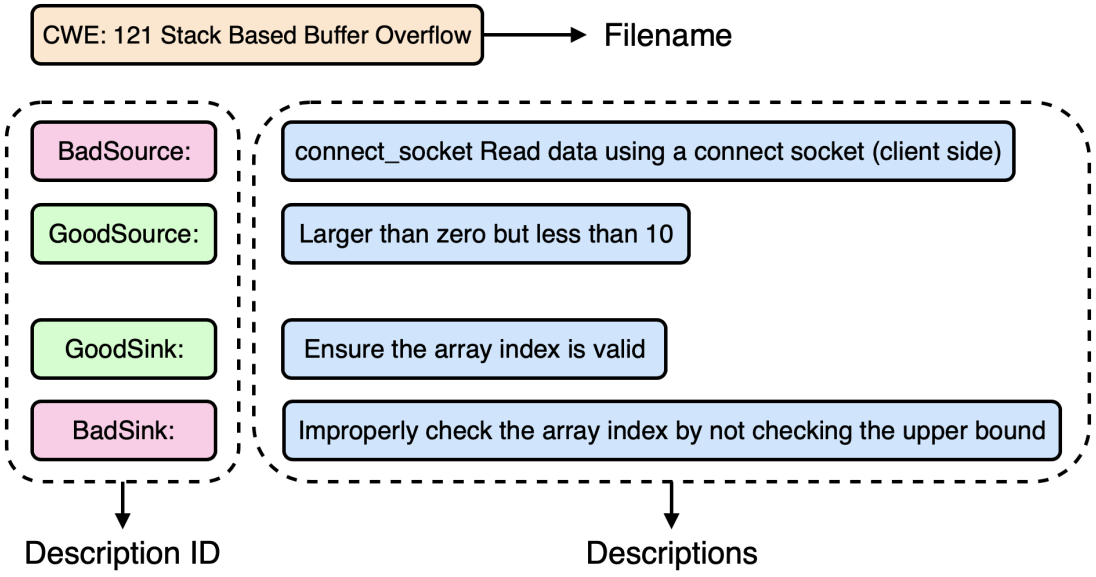

Figure 5.1: An example showing the format of the header for each source code file containing only the relevant information. The filename in this representation does not include whether it is a "good" file or a "bad" file. The description ID is used to split the data and extract the appropriate description dependent on if it is "good" or "bad".

The information is in the form of a natural language description. The layout of the header is the same for each file, apart from the description itself, making it possible to extract each "good" or "bad" description iteratively. With the assembly

code filename indicating if it is a good case or a bad case, the appropriate information is extracted from the header of the source code. This information is processed into a usable format for the the natural text output of the model.

5.1.2 Data Preprocessing

After extraction, preprocessing of the data is needed to convert the data into a usable format for the model. The assembly code instructions are the input for the model and start out as sequences of strings. The natural language descriptions are output for the model and also start out as sequences of strings. These string sequences must be converted to sequences of integers for the model to use.

The strings are tokenized, creating a dictionary of mappings between strings and integers. Each integer corresponds to the index of a token in the dictionary. The maximum sequence length is set to 256. If a sequence is longer than the maximum length, it is truncated at 256, and if a sequence is not long enough, it is padded with zeros until it is the appropriate length. This preprocessing step ensures all samples are the same length. A simple example of this preprocessing step can be seen in Figure 5.3 where arbitrary assembly code instructions are combined and then converted to integers.

After converting the strings to integers, the assembly code input data is passed through an embedding layer. The embedding layer transforms the individual integers into mathematical objects that can be operated on [15]. One calculation that is useful is the distance between vectors as this represents the distance between words. Knowing this distance makes it easier to generalize the behaviour from one word to another [15], which is very useful during the training process. The embedding layer

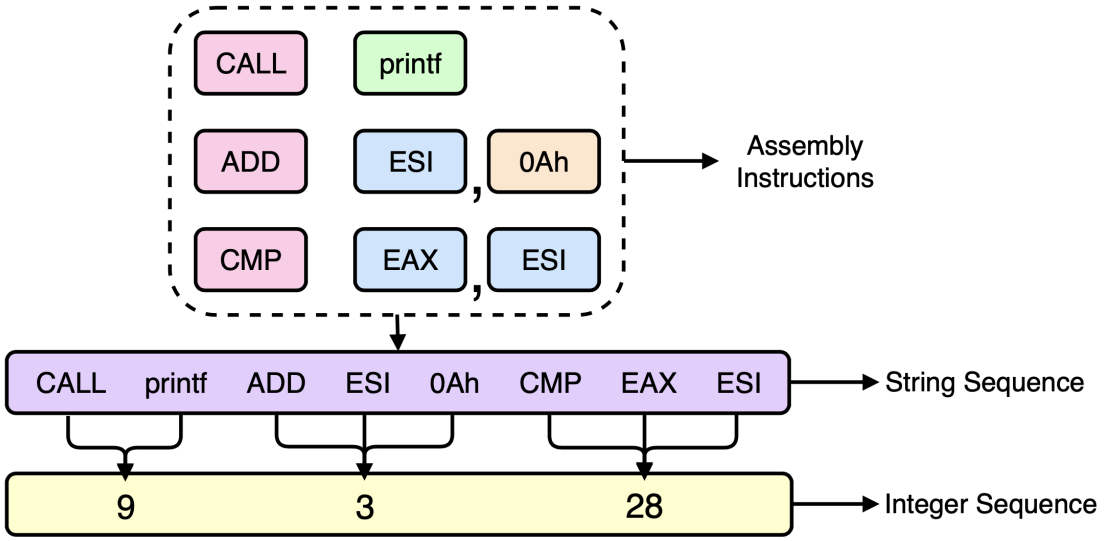

Figure 5.2: A sample representation of the preprocessing procedure for the assembly code input. Assembly instructions are combined to form a sequence of strings that are then converted into integers. Integers in this figure are not necessarily the correct values from the token vocabulary.

is the final preprocessing step for the input data before being passed to the encoder of the model.

The natural language outputs require a little more preprocessing. After it has been converted to sequences of integers it must be padded with zeros. The inputs to the decoder are padded with leading zeros that represent the start-of-sequence (SOS) tokens. The SOS token is needed so that the model is able to predict the first character in the target sequence. The output data of the decoder are padded with trailing zeros that represent the end-of-sequence (EOS) tokens. The EOS token is needed so that the model knows when the current predicted sequence is complete. The additional step of padding the data is shown in Figure 5.3.

The SOS and EOS tokens tell the model the boundaries of each individual input so we ensure it doesn't group sequences together and learn incorrect dependencies

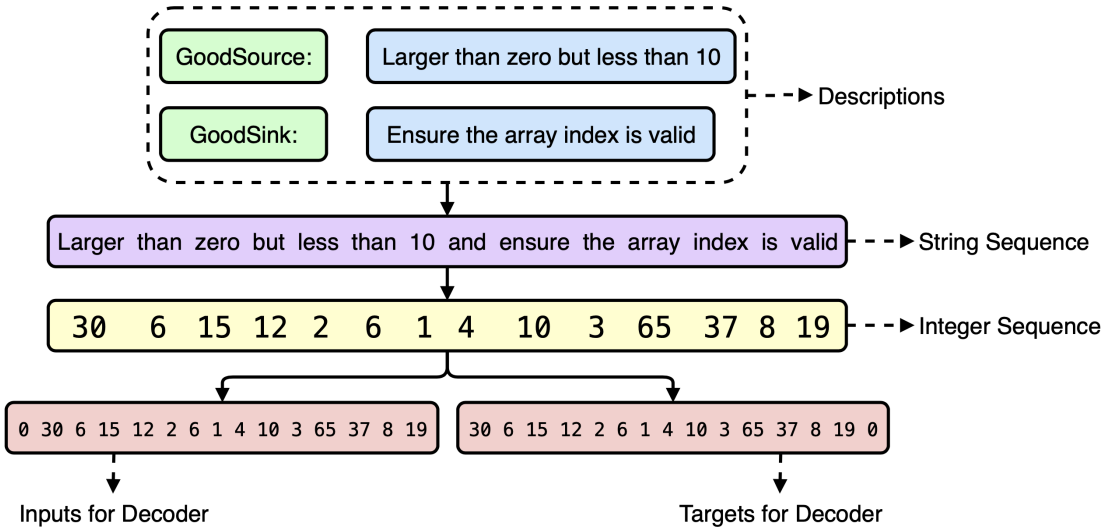

Figure 5.3: A sample representation of the preprocessing procedure for the natural description output. Descriptions are combined into a sequence of strings that are then converted into a sequence of integers. Integers in this figure are not necessarily the correct values from the token vocabulary. Padding is added to either the beginning or end of the integer sequences to represent SOS and EOS tokens needed for training and testing the model.

and patterns. The padding process also ensures that the target data is one time-step ahead, which is required as we are training the model to predict the next character of the target sequence given previous characters of the target sequence.

5.1.3 Characteristics

The characteristics of the data are used to determine different hyperparameters in order to optimize the model's performance.

The Juliet Test Suite dataset consisted of 272,431 files, the combined set of assembly files and source code files. After some initial preprocessing to eliminate incorrectly formatted files, the dataset was reduced to 83,326 samples consisting of assembly-description pairs. Splitting the data into training and testing sets reduced

the training data to 62,494 assembly-description pairs. This leaves 20,832 assembly-description pairs for the testing dataset.

The NDSS18 dataset consisted of 35,955 Windows samples and 28,608 Linux samples consisting of both assembly files and source code files. After initial preprocessing to eliminate incorrectly formatted files, the Windows dataset was reduced to 7,160 assembly-description pairs, and the Linux dataset was reduced to 7,006 assembly-description pairs. Combining these samples, the total dataset extracted from the NDSS18 data is 14,166 assembly-description pairs. Splitting the data into training and testing sets gives 10,624 training samples and 3,543 testing samples.

5.2 Experiments

Each experiment was conducted in the same way using different neural networks for the encoder-decoder. The datasets were divided into a training set, a validation set, and a testing set. The *train_test_split*() method from the Scikit-learn machine learning library was used to split the data. The data is split into 75% training and 25% testing. The training set is further split into 80% training and 20% validation. A pictorial explanation of the split dataset can be seen in Figure 5.4.

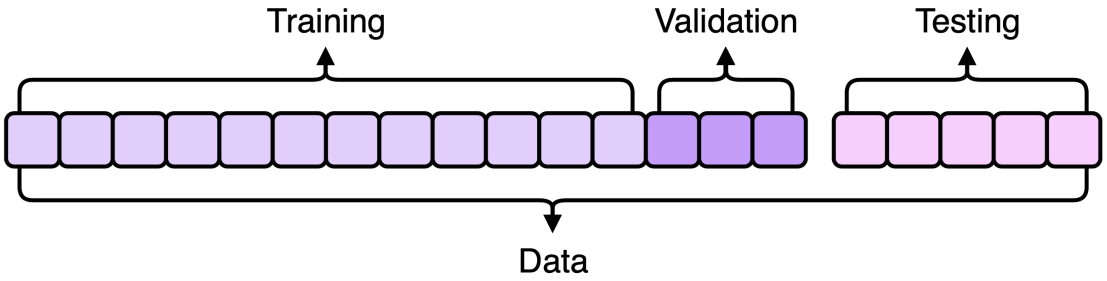

Figure 5.4: Pictorial representation of the train-test split. Training set is 75% of the data, testing set is 25% of the data, and validation set is 20% of the training set.

The training set is the sample of data used to fit the model. The model learns the appropriate weights and biases using this reduced dataset. The validation set is the sample of data used to evaluate the model on the training dataset. This evaluation is used frequently to fine-tune the hyperparameters of the model. The neural network does not learn from the validation set. The testing set is the sample of data used to evaluate the final model after training is complete. The adjusted accuracy metric uses the testing dataset to get its values.

Hyperparameters are variables used to determine the structure of the network and how the network is trained. Adjusting these variables will directly coincide with the performance of the network. Some of the most common hyperparameters include batch size, number of hidden units in the encoder and decoder, number of epochs, optimizer, loss function, etc. These parameters are chosen based on a number of factors, the size of the dataset, the memory space of the GPU, the type of data that is used, etc.

Batch size is one of the most common hyperparameters to tune and can have a significant impact on how fast the model is able to converge on it's loss function. For this research, 192 samples were set as the batch size. Our goal was to max out the batch size based on the GPU memory, ensuring we get a more accurate estimate of the gradient for the batch. Maximizing the batch size which will lead to a faster convergence and consequently faster training of the network.

The number of epochs used for training is another widely fluctuating parameter. The final number of epochs used to train the network was 150. Preliminary testing was done on a greater number of epochs and fewer epochs, but with early stopping, 150 consistently produced better results.

The optimizer and loss function go hand-in-hand with each other as the goal of the optimizer is to optimize the loss function used. RMSprop was chosen for the optimizer due to its adaptive learning rate, taking care of any need to manually optimize the learning rate during training. Cross-entropy was the chosen loss function because the model makes predictions based on a set of probabilities for each target token.

Additional hyperparameters are assigned values commonly found to be favorable in sequence-to-sequence networks or were based on the size of the dataset. The maximum sequence length for the encoding process was set to 256. This value ensure all tokens in the description sequences were included unless the sequence was excessively long; and that assembly sequences were not overly large as assembly instructions can be tens of thousands of lines of code. The maximum vocabulary size for the token dictionary was set to 20,000 to ensure all tokens were accounted for. Tuning of the hyperparameters determine how well the model will function overall.

The encoder processes the input through several encoding layers and generates encodings (internal hidden states) containing information about which parts of the input are relevant to each other. These encodings are passed to the next encoder layer as part of its input. This process continues down each layer of the encoder for each input until all the data has been processed for that sample and a final encoder vector is produced. This encoder vector is the final hidden state for that sample and aims to encapsulate all the necessary information from the input elements in order to help the decoder make an accurate prediction.

The decoder for the model does the opposite; it takes encodings and processes them to generate an output sequence. The decoder is given the final encoder vectors as initial hidden states and propagates the information forward through several decoder

layers until a final output is given. A softmax function is used to normalize the final output into a probability distribution. The distribution covers the range of output classes possible for prediction. When the model begins predicting outputs, the prediction with the highest probability is used as the final prediction for the given input.

5.3 Results

Different evaluations are considered and discussed to determine how well the experiments performed. All result evaluations discussed use the best iteration for each experimental model tested. The results for the evaluation metrics on the NDSS18 dataset are only discussed for the best model so as not to be redundant. The evaluation of the models are dependent on how well they were trained. If the training process was completed successfully, evaluation metrics should produce favorable results. When comparing the predicted outputs to their true descriptions, it is important to identify whether or not the model was capable of learning the appropriate dependencies. The attention model shows insight into what aspects of the input are key to predicting the correct output, which are important to showcase for further understanding of how the model works.

5.3.1 Training Convergence

Apart from the Bi-GRU model, all other experiments produces favorable results when early stopping on the loss metric during training. When early stopping on the loss metric, all models completed every epoch or came close to completing every epoch, meaning these experiments continued to improve, however small, until training was

complete. Figure 5.5 depicts the loss convergence of four different models. The four models shown produced the best results when early stopping on loss. The validation set performs comparatively to the training set for all experiments. Another key point to note is how quickly the models converge; the loss value evens out within roughly 20 iterations for each.

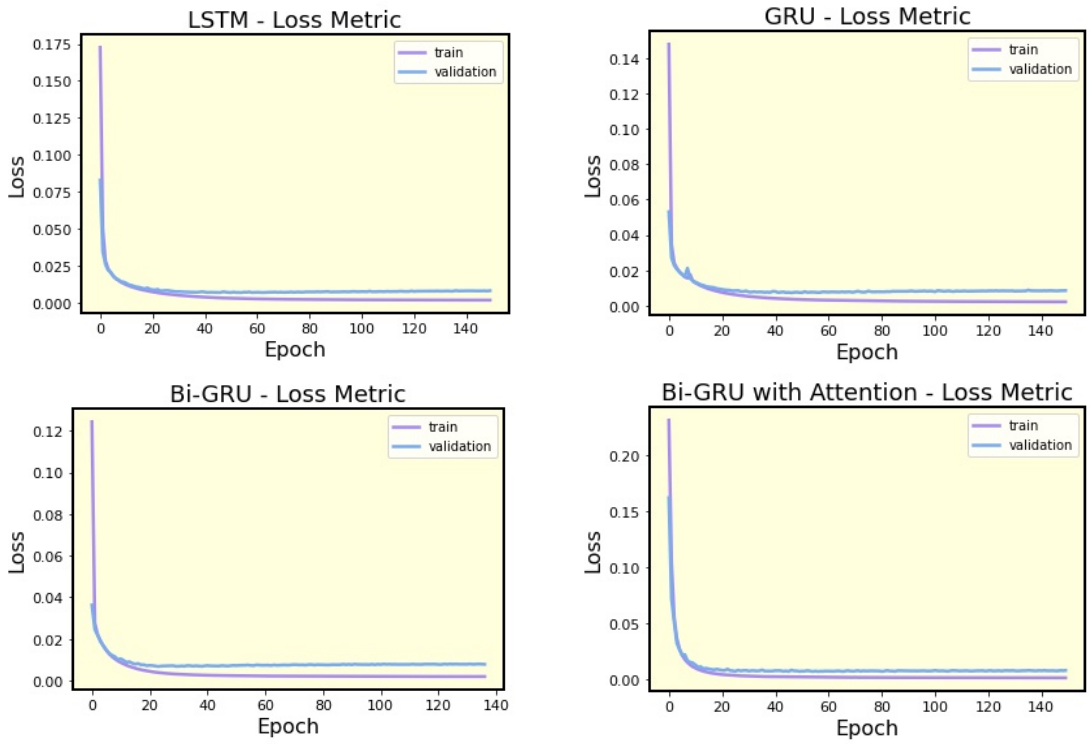

Figure 5.5: Loss metric convergence during training for four different models, LSTM, GRU, Bi-GRU, and Bi-GRU with Attention.

The Bi-GRU experiment produced the best results with early stopping on the sparse categorical accuracy metric (SCA). Figure 5.6 shows the convergence of SCA for the GRU and Bi-GRU models as they produced the best results when early stopping on SCA. Similar to the loss metric, both models converge quite quickly within 20 iterations of the training process before evening out. Neither model is able to

complete all epochs before the sparse categorical accuracy begins to decline. Early stopping is favorable to ensure overfitting doesn't occur.

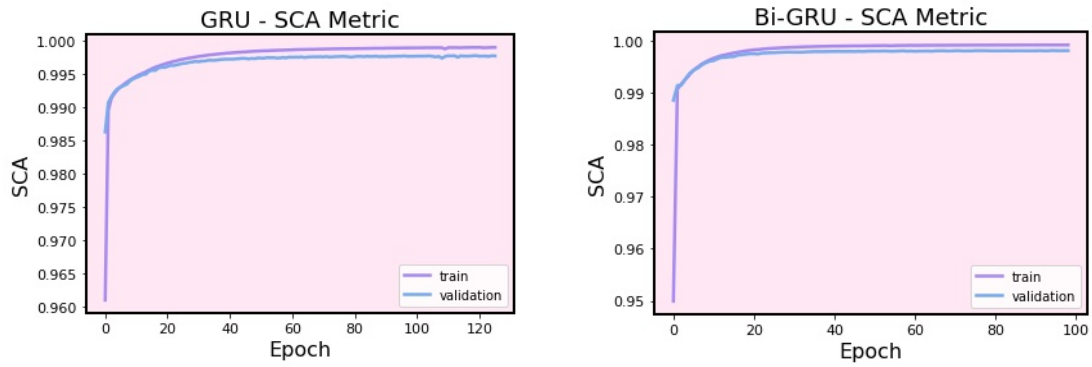

Figure 5.6: Sparse Categorical Accuracy (SCA) metric convergence during training for the two best models with early stopping on SCA.

5.3.2 BLEU Score

BLEU score is an indication of how well the predicted outputs match one of the extracted descriptions, which can also mean how readable the predicted outputs are. Due to the differing number of reference sequences and differing lengths of reference sequences, it can be difficult to draw comparisons between already published work and BLEU score results for this research. Two studies with similarly sized reference sequences were used to determine if the BLEU scores calculated are considered good values or not with respect to the size of the reference set. This dataset uses 20,832 reference sequences to calculate the BLEU score. The NDSS18 dataset uses only 3,542 reference sequences when calculating the BLEU score. Results for each of the five tested sequence-to-sequence models can be seen in Table 5.1.

LeClair et al. used just over 100,000 reference sequences for testing their BLEU score and achieved a result of 0.209 for its best model [28]. On average predicted

Table 5.1: BLEU Score Results

Juliet Test Suite

Seq2Seq Model	BLEU
LSTM	0.523
Bi-LSTM	0.506
GRU	0.513
Bi-GRU	0.534
Bi-GRU+Attention	0.537

NDSS18

Seq2Seq Model	BLEU
LSTM	0.512
Bi-LSTM	0.492
GRU	0.481
Bi-GRU	0.516
Bi-GRU+Attention	0.880

outputs for LeClair et al. were about half the size when compared to those used for this research. Iyer et al. used less than 10,000 reference sequences for calculating their BLEU score and achieved a best results of 0.205 [23]. Predicted outputs for Iyer et al. were on average comparable in size or smaller than those in this research. Based of these results from other research, a BLEU score of 0.537 for the best model on the Juliet Test Suite and 0.880 for the best model on the NDSS18 dataset are relatively high in comparison. LeClair et al. had significantly more reference sequences and Iyer et al. had about half as many reference sequences and they achieved a comparable result to each other. This observation leads to the conclusion that the model was trained well and is giving really good natural language descriptions for the sizes of the datasets.

5.3.3 Accuracy

Accuracy is used to determine how well the full predicted output sequence matches the true description for that sample. Accuracy measurements for the Juliet Test Suite models can be seen in Table 5.2. Some of the results are interesting considering they are the best models for each experiment. The LSTM model and the Bi-GRU with attention model were the only two models that produced a lower adjusted accuracy

when compared to the overall accuracy. Although it is ideal for the overall accuracy to be greater than the adjusted accuracy, these discrepancies are likely due to numeric stability issues.

There are a couple possible explanations for the low adjusted accuracy results. The models use backpropagation training through a gradient decent process to search through the possible weight values in order to minimize error. This process can results in the model getting stuck in local minimum of a particular weight-space causing it to become unstable and output potentially inaccurate results. The inconsistencies seen in the LSTM model and the attention model do not occur on every trial but were chosen as the best models regardless based on the other metrics being evaluated.

Another possibility observed showed that even after the model produced the EOS token, additional tokens were sometimes predicted. These additional tokens were considered errors and would inevitably decrease the adjusted accuracy rate even though they are not included in the actual prediction. When decoding the sentences, predictions end at the EOS token so additional tokens following do not occur.

Table 5.2: Accuracy results for the Juliet Test Suite dataset.

Juliet Test Suite

Seq2Seq Model	**Adjusted Accuracy (%)**	**Overall Accuracy (%)**
LSTM	66.8	77.7
Bi-LSTM	99.8	74.9
GRU	99.8	76.3
Bi-GRU	99.8	79.2
Bi-GRU+Attention	7.0	79.6

The remainder of the accuracy measurements produce favorable results. The high values for the adjusted accuracies indicates that the models for both datasets consistently produce the correct target sequence given the correct input tokens for

that sample. The overall accuracy is expected to decreased in comparison because a correct input is not always given, resulting in the overall output sequence to naturally have fewer correctly predicted target sequences. The overall accuracy averages around 77% for the Juliet Test Suite data, meaning the models predict the correct output 77% of the time.

The NDSS18 data accuracy results are displayed in Table 5.3. Similar to the Juliet Test Suite, adjusted accuracy are high for all models. The increased overall accuracy for the Bi-GRU with attention model could be due to the reduced size in the dataset. The model is likely able to learn dependencies better because the variance in possible outputs is significantly reduced.

Table 5.3: Accuracy results for the NDSS18 dataset.

NDSS18

Seq2Seq Model	**Adjusted Accuracy (%)**	**Overall Accuracy (%)**
LSTM	94.3	65.9
Bi-LSTM	94.0	59.6
GRU	92.9	57.7
Bi-GRU	93.8	69.2
Bi-GRU+Attention	98.1	84.1

With respect to both datasets, it is expected that the overall accuracy for Bi-GRU with attention is the highest of all models. Adding attention allows the model to understand the dependencies that are learned through training which should produce better results overall.

5.3.4 Precision and Recall

Recall is used to determine how well the model is able to produce a correct output given a correct input. Precision is the measurement of how many correctly flagged

outputs are actually correct in the context of the current target sequence being predicted.

Precision and Recall produced consistent results across all models for the Juliet Test Suite as seen in Table 5.4. The model averages 67% for recall indicating that 67% of the time a correct input is given, the correct output will be predicted. Of those 67% correct target tokens, an average of 76% of them are the correct target output for the current sample being predicted.

Table 5.4: Precision & Recall results for Juliet Test Suite dataset.

Juliet Test Suite

Seq2Seq Model	**Precision (%)**	**Recall (%)**
LSTM	76.3	66.9
Bi-LSTM	73.8	64.2
GRU	76.9	65.8
Bi-GRU	79.2	69.0
Bi-GRU+Attention	77.9	68.6

Table 5.5 shows the precision and recall results for the NDSS18 dataset. Numbers are significantly lower than those of the Juliet test Suite, which could be due to the reduced size of the dataset. With a limited number of samples, every incorrect results has more of an impact on the final percentages when compared to a much larger dataset.

Some of the loss in precision and recall for either dataset can be accounted for in samples that take on different meanings but are still accurate translations. A predicted output of “non negative” vs. the true target of “greater than zero” will reduce precision and recall as the input-to-output pair and just the output itself will be flagged as incorrect. The interpretation of these two outputs have the same meaning so technically this is an appropriate translation even though the metrics are

Table 5.5: Precision & Recall results for NDSS18 dataset.

NDSS18		
Seq2Seq Model	**Precision (%)**	**Recall (%)**
LSTM	37.0	32.9
Bi-LSTM	40.0	31.8
GRU	31.0	26.8
Bi-GRU	38.6	32.7
Bi-GRU+Attention	68.6	60.0

unable to detect it.

5.3.5 Prediction vs. Truth

The results from the metrics analysis can be expanded and demonstrated by comparing the predicted outputs to their true descriptions. Some interesting samples of predicted descriptions compared to true descriptions for two of the best models are displayed in Figures 5.7 and 5.8.

Figure 5.7 includes three examples of predictions and their actual descriptions from the GRU model. The first sample is simply a correctly predicted description. This sample showcases that the model is capable of correctly predicting a description that matches true description perfectly. The second and third samples are used to demonstrate different ways the model has learned dependencies.

The second samples in Figure 5.7 matches completely up until the last word. Both the “memmove” and “memcpy” functions are used to copy data from one location to another. Although not all instances of “memcpy” can be replaced with “memmove”, this substitution indicates that the model was capable of learning how similar these functions are. This sample is also a good example to demonstrate how the recall increases but the precision decreases. Both descriptions produce appropriate outputs

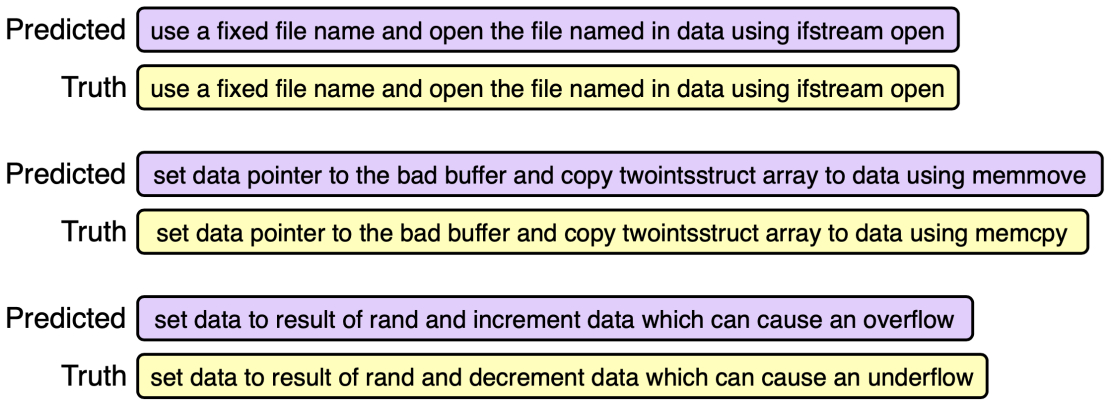

Figure 5.7: Samples of the predicted output compared to the true description for the GRU model.

given the correct input, which would increase the recall, but the final output is not correct given the true descriptions, which would decrease the precision.

The third sample is an interesting demonstration of how an incorrect prediction earlier in the sequences can produce an overall different output description. Another thing to note is how the model knew to predict "overflow" in accompaniment with "increment" and "underflow" in accompaniment with "decrement". Although the model made the wrong description, it is still displaying the model's ability to learn patterns within the data.

Figure 5.8 includes three examples of predictions and their actual descriptions from the Bi-GRU model. Similar to the last set of examples, sample one demonstrates the model's ability to give a correct prediction that matches the true description perfectly. It is important to show that each model is capable of producing accurate results.

The second sample is used to demonstrate how, even though the model didn't produce the correct output, the information is still applicable. The predicted output

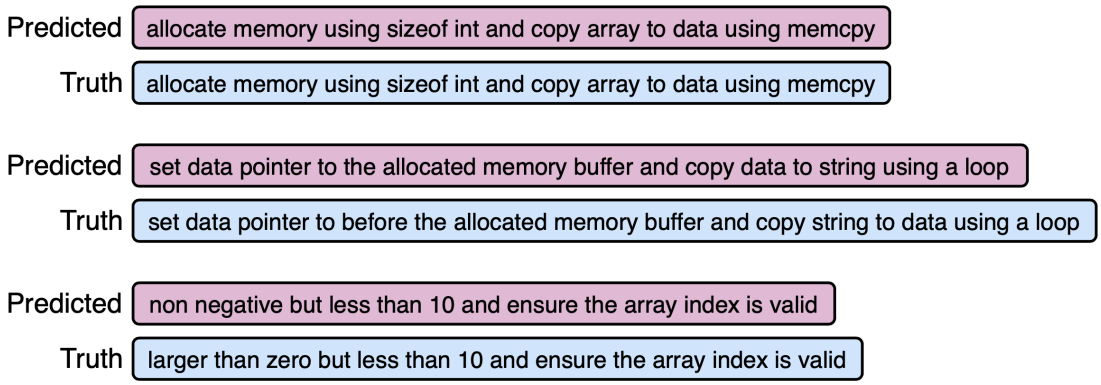

Figure 5.8: Samples of the predicted output compared to the true description for the Bi-GRU model.

can be used as a starting point for further analysis of the assembly code where additional information can fill in the blanks. The time needed to analyze the original assembly code would be reduced by the predicted output because the majority of the information needed to explain the vulnerability has been predicted correctly.

The last example of Figure 5.8 is by far the most important. This example is an excellent demonstration of how well the model was able to learn and not just memorize patterns in the data. A non-negative integer is the same thing as an integer larger than zero, while the remainder of the prediction matches the true description. This showcases how the model was able to understand that these words had the same meaning while still being able to finish the prediction correctly. This prediction-truth pair is a perfect example of what we hope the model is able to achieve.

The NDSS18 dataset output produced comparative results with the majority either being a perfect match or containing a few incorrect tokens somewhere in the middle of the sequence.

5.3.6 Attention Scores

Adding attention gives information about what aspects of the input are most relevant to predicting the output. Attention scores are used to show the most important assembly instruction sections that were used to identify a particular output token. Table 5.6 through 5.8 showcase some of the assembly instructions and their corresponding attention scores for various words in the predicted natural language output sequences.

Table 5.6 gives the top two section of assembly instructions and attention scores for various predictions of the word "data" on the Juliet Test Suite data. For each sample shown, the top two sets of instructions have a score equaling over 50% of the attention. These attention scores create a high correlation between the assembly instructions and the associated natural language token within the model which can be used for future predictions of this target output.

Table 5.6: Attention scores and assembly instructions associated with them for various predictions of the word "data".

Word	Assembly Instructions	Attention Scores
data	'mov' 'rbp' 'rsp' 'sub'	0.38
	'mov' 'rbp' 'rsp' 'sub'	0.14
	'rbp' 'rsp' 'sub' 'rsp'	0.38
	'mov' 'rbp' 'rsp' 'sub'	0.27
	'rbp' 'rsp' 'pop' 'rbp'	0.62
	'rbp' 'var' '25' 'dl'	0.08
	'mov' 'rbp' 'rsp' 'sub'	0.39
	'mov' 'rbp' 'rsp' 'sub'	0.16

Analyzing the assembly instructions, we can see how similar they are. The "rsp" and "rbp" instructions are associated with registries where data is often stored; whereas the "mov" and "sub" instructions are action instructions often associated with a transfer or calculation of data. Seeing a consistent output of the same assembly instructions used to predict the same output token further indicates how the

model was able to find appropriate correlations within the data to use for predictions.

A similar observation can be seen in Table 5.7 which showcases the top two sets of assembly instructions used to predict two different functions found within the natural language descriptions. The attention scores for these samples are even higher than in table 5.6 meaning the correlation between the instructions and the target tokens are even better.

Table 5.7: Attention scores and assembly instructions associated with them used for the prediction of two functions.

Function	Assembly Instructions	Attention Scores
memmove	'20h' 'call' 'memmove' 'chk'	0.88
	'jmp' 'cs' 'memmove' 'chk'	0.02
	'mov' 'rbp' 'rsp' 'call'	0.57
	'rbp' 'mov' 'rbp' 'rsp'	0.10
strncpy	'rbp' 'var' 'c8' 'call'	0.84
	'var' 'c8' 'call' 'strncpy'	0.03
	'jmp' 'cs' 'strncpy' 'chk'	0.40
	'64h' 'call' 'strncpy' 'chk'	0.26

We can see each function is found in the assembly instructions in addition to the "call" instruction, which indicates how the model was able to identify the tokens as functions. The registry instructions are also indicative of the model creating appropriate connections as both of these functions deal with data.

Lastly, Table 5.8 showcases the top two sets of assembly instructions used to predicted various important words within a description for this dataset. The word "execute" is predicted by instructions such as "jmp", "execlp", and "call" which are all correlated with executing a function or a piece of code. The word "result" is predicted by popping data stored in a registry off of the stack and returning it, which is very often seen when outputting the result of a function of piece of code. Finally, the word "memory" is predicted by registry instructions and "mov" instructions which

very often have to do with storing data or transferring data within the memory of the computer.

Table 5.8: Attention scores and assembly instructions associated with them used for the prediction of different words.

Word	Assembly Instructions	Attention Scores
execute	'jmp' 'cs' 'execlp' 'ptr'	0.34
	'rbp' 'rsp' 'call' 'rand'	0.12
result	'rsp' 'pop' 'rbp' 'retn'	0.31
	'rsp' 'pop' 'rbp' 'retn'	0.13
memory	'8' 'mov' 'rcx' 'rbp'	0.49
	'8' 'rax' 'mov' 'rbp'	0.20

Similar to Table 5.6, the combination of the two highest attention scores make up over 50% of the attention when the model predicted those output tokens. The models showed favorable results in all evaluation metrics, but limitations to its ability exist nonetheless.

5.4 Limitations and Future Work

With any supervised learning algorithm, the model requires input and output samples for proper training, and therefore predicting. This requirement limits the ability of the model to give predictions given only the assembly code of a program. Although still beneficial for analysis, using supervised learning does limit the applications of the model to some extent. Applying unsupervised or semi-supervised learning methods to this problem is a good next step for any future work to expand its possible applications.

5.4.1 External Validity

The model is only tested on two datasets, potentially limiting the models generalizability. This limitation is addressed by splitting the data into training, validation, and testing sets to ensure overfitting of the data does not occur. The datasets used are real world vulnerability data giving actual variations that increase the models generalizability as well. Additional datasets used only for testing would be a good future step in order to validate the findings further and demonstrate the model's ability to correctly make predictions for different data.

There are two other main factors that limit the models transferability, the assembly code input and the English text output. The model is built to have x86/AMD64 assembly code as input so other architectures such ARM assembly cannot be applied without significant adjustments. The model is also designed to output English natural language predictions. Without testing additional datasets in different languages, it is not clear if the current model would be able to function properly on datasets comprised of languages other than English. Future work could expand the possible inputs or outputs to encompass a more diverse set of data. In addition to natural language descriptions for output, expanding the output to include the type of vulnerability would be very beneficial. The vulnerability type could be extracted from the filenames and used as the sole output or additionally to the functionality descriptions.

A statistical analysis will be required to further validate the results of the model. Doing so will ensure there is no cloned data and that training, validation, and testing sets are unique and not causing any inflation to the evaluation metrics due to repeat samples during training and testing.

Truncated length of extracted assembly code inputs pose questions into whether

or not a sufficient amount of instructions were extracted from the code to perform the task at hand. The sequence length was not a tested factor, leaving the possibility of biased results. Although results produced were good, extracting more assembly instructions or more specific assembly instructions related to the vulnerabilities would increase the validity of the results.

Due to the limited number of datasets and similarities between the data tested, the explainability results may be partial to possible repetitive assembly instructions and words within the descriptions. The model needs to be assessed on more diverse datasets and varying case studies other than vulnerability related aspects. Doing so would support the explainablility results given in this thesis and justify further many of the results as well.

The research focuses om vulnerabilities found within the data. The applications of a model like this can be greatly expanded and potentially directed more at general functional summaries that will prove to be useful for more general malware analysis and reverse engineering.

5.4.2 Internal Validity

Numerical stability issues can interfere with the evaluation results of the model. Numerical stability issue can arise from the gradient decent training method used. As the model searches through weight values to minimize error, the model can get stuck in a local minimum of a particular weight-space. If this occurs, the values become unstable and output potentially inaccurate results. This issue doesn't speak to the accuracy of the model, it more so provides an explanation as to why outlying results are sometimes seen in experimental trials.

Optimization techniques are explored and applied to reduce limitations in the experimental protocol. The methodology is designed in a generic way, but characteristic of the model still need to be defined based on the problem. Using optimization methods ensures external factors decide some of the key characteristics. Some characteristics must still be manually inputted though, which leaves the potential for selection bias within the design of the model. Testing various optimizations variables and different characteristic values would provide additional insight into the model's functionality.

This chapter focuses on how the experiments were conducted and the results achieved from testing each of the models. The input and output data required preprocessing before it was able to be passed to the models. The data was divided into training, validation, and testing sets. The different experiments tested included an LSTM model, a GRU model, Bidirectional models for both LSTM and GRU, and an attention mechanism added to the best performing model. Results showed favorable outcomes for all experiments with Bi-GRU and Bi-GRU with attention performing the best. Comparing the true descriptions to the predicted outputs gives insight into how the model was able to learn dependencies and understand the information outside of direct pattern repetition. Lastly, the limitations and future work associated with this problem are discussed.

Chapter 6

Conclusions

The goal of the research was to successfully create a neural network model capable of predicting natural language description outputs from assembly language inputs. Various models were assessed to determine the best performing model. Results indicated that the bidirectional GRU network produced the most favorable results with an overall accuracy of 79.2% and a BLEU score of 0.534.

Attention was added to the Bi-GRU model to gain insight into how it determines the output predictions. One of the most obvious relations was with the token word "data". The model primarily used registries, where data is stored, and action instructions associated with the transfer of calculation of data. Both of these explanations demonstrate that the patterns and dependencies formed during training are appropriate.

Comparing the predicted outputs to their true description counterparts revealed that the model not only learned surface level dependencies, but learned to understand the word tokens on a deeper level. The model is observed predicting "non-negative" as an output for "larger than zero" followed by a perfectly matching remaining prediction. This demonstrates the models ability to learn the meaning behind the words

and predict equivalent outputs that stem from the same intention.

Two different datasets were tested on the model to eliminate any question of bias. Both dataset trials produced beneficial results when trying to determine the functionality of a vulnerability in assembly code. Overall, after studying the resultant evaluation metrics, prediction-truth comparisons, and attention scores, it can be concluded that the goal of predicting natural language outputs from assembly code inputs was achieved. This thesis presented a feasible solution with explainability for a novel problem using a newly created dataset in a field that continues to grow.

Bibliography

[1] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, page 38–49. Association for Computing Machinery, 2015.

[2] Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code. In *Proceedings of The 33rd International Conference on Machine Learning*, volume 48, pages 2091–2100, New York, New York, USA, 20–22 Jun 2016. PMLR.

[3] Uri Alon, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations*, 2019.

[4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2015.

[5] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. BYTEWEIGHT: Learning to recognize functions in binary code. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 845–860, San Diego, CA, August 2014. USENIX Association.

[6] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. Bap: A binary analysis platform. In *Computer Aided Verification", year="2011*, pages 463–469, Berlin, Heidelberg. Springer Berlin Heidelberg.

[7] Giuseppe Carenini, Jackie Chi, and Kit Cheung. Extractive vs. nlg-based abstractive summarization of evaluative text: The effect of corpus controversiality. 06 2008.

[8] Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder–decoder approaches. In *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, pages 103–111, Doha, Qatar, 2014. Association for Computational Linguistics.

[9] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. 2014.

[10] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. Neural nets can learn function type signatures from binaries. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 99–116, Vancouver, BC, August 2017.

[11] Junyoung Chung, aglar Gehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014.

[12] Zhiyong Cui, Ruimin Ke, Ziyuan Pu, and Y. Wang. Stacked bidirectional and unidirectional lstm recurrent neural network for forecasting network-wide traffic

state with missing values. *Transportation Research Part C: Emerging Technologies*, 118, 2020.

[13] Giovani Dematos, Milton S. Boyd, Bahman Kermanshahi, Nowrouz Kohzadi, and Iebeling Kaastra. Feedforward versus recurrent neural networks for forecasting monthly japanese yen exchange rates. *Financial Engineering and the Japanese Markets*, 3:59–75, 1996.

[14] S. Gao, C. Chen, Z. Xing, Y. Ma, W. Song, and S. Lin. A neural model for method name generation from functional description. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 414–421, 2019.

[15] Yoav Goldberg. *Neural network methods for natural language processing*. Synthesis Lectures on Human Language Technologies, Lecture number 37. San Rafael, California, 2017.

[16] A. Graves, A. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6645–6649, 2013.

[17] A. Graves and J. Schmidhuber. Framewise phoneme classification with bidirectional lstm networks. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 4, pages 2047–2052, 2005.

[18] Alex Graves. *Supervised Sequence Labelling with Recurrent Neural Networks*. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2012.

[19] Udo Hahn and Inderjeet Mani. The challenges of automatic summarization. *Computer*, 33(11):29–36, 2000.

[20] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, and Jurgen Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. *A Field Guide to Dynamical Recurrent Neural Networks*, 2001.

[21] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[22] Md. Sanzidul Islam, Sadia Sultana Sharmin Mousumi, Sheikh Abujar, and Syed Akhter Hossain. Sequence-to-sequence bangla sentence generation with lstm recurrent neural networks. *Procedia Computer Science*, 152:51 – 58, 2019. International Conference on Pervasive Computing Advances and Applications-PerCAA 2019.

[23] Srini Iyer, Ioannis Konstas, A. Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, ACL '16, page 2073–2083, 2016.

[24] L. Jiang, H. Liu, and H. Jiang. Machine learning based recommendation of method names: How far are we. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 602–614, 2019.

[25] Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. An empirical exploration of recurrent network architectures. In *Proceedings of the 32nd International Conference on Machine Learning*, volume 37, page 2342–2350, 2015.

[26] D. S. Katz, J. Ruchti, and E. Schulte. Using recurrent neural networks for decompilation. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 346–356, 2018.

[27] Tue Le, Tuan Nguyen, Trung Le, Dinh Q. Phung, P. Montague, O. Vel, and Lizhen Qu. Maximal divergence sequential autoencoder for binary software vulnerability detection. In *ICLR*, 2019.

[28] Alexander LeClair, Siyuan Jiang, and Collin McMillan. A neural model for generating natural language summaries of program subroutines. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 795–806. IEEE Press, 2019.

[29] Dor Levy and Lior Wolf. Learning to align the source code to the compiled object code. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70, pages 2043–2051, International Convention Centre, Sydney, Australia, 06–11 Aug 2017.

[30] Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiský, Fumin Wang, and Andrew Senior. Latent predictor networks for code generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 599–609. Association for Computational Linguistics, 2016.

[31] Ashok Vardhan Makkuva, Sewoong Oh, Sreeram Kannan, and Pramod Viswanath. Learning in gated neural networks. *CoRR*, abs/1906.02777, 2019.

[32] Inderjeet Mani and Mark Maybury. *Advances in Automatic Text Summarization*. The MIT Press, Cambridge, Massachusetts, 2nd. edition, 2001.

[33] David Muñoz-Valero, Luis Rodriguez-Benitez, Luis Jimenez-Linares, and Juan Moreno-Garcia. Using recurrent neural networks for part-of-speech tagging and subject and predicate classification in a sentence. *International Journal of Computational Intelligence Systems*, 13:706–716, 2020.

[34] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, page 311–318. Association for Computational Linguistics, 2002.

[35] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2017.

[36] Alexander M. Rush, Sumit Chopra, and Jason Weston. A neural attention model for sentence summarization. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, page 379–389. Association for Computational Linguistics, 2015.

[37] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing functions in binaries with neural networks. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 611–626, Washington, D.C., August 2015. USENIX Association.

[38] Xiaotao Song, Hailong Sun, Xu Wang, and Jiafei Yan. A survey of automatic generation of source code comments: Algorithms and techniques. *IEEE Access*, 7:111411–111428, 2019.

[39] M. Sundermeyer, I. Oparin, J. . Gauvain, B. Freiberg, R. Schlüter, and H. Ney. Comparison of feedforward and recurrent neural network language models. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 8430–8434, 2013.

[40] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, page 3104–3112. MIT Press, 2014.

[41] Arda Tezcan, Véronique Hoste, and Lieve Macken. A neural network architecture for detecting grammatical errors in statistical machine translation. *Prague bulletin of mathematical linguistics*, 108:133–145, 2017.

[42] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. Code generation as a dual task of code summarization. *CoRR*, abs/1910.05923, 2019.

[43] H. Xue, S. Sun, G. Venkataramani, and T. Lan. Machine learning-based analysis of program binaries: A comprehensive study. *IEEE Access*, 7:65889–65912, 2019.