Asm2Seq: Explainable Assembly Code Functional Summary Generation for Reverse Engineering and Vulnerability Analysis

SCARLETT TAVISS*, STEVEN H. H. DING*, MOHAMMAD ZULKERNINE*, PHILIPPE CHARLAND§, SUDIPTA ACHARYA*, * School of Computing, Queen's University, Canada and § Mission Critical Cyber Security Section, Defence R&D Canada, Canada

Reverse engineering is the process of understanding the inner working of a software system without having the source code. It is critical for firmware security validation, software vulnerability research, and malware analysis. However, it often requires a significant amount of manual effort. Recently, data-driven solutions were proposed to reduce manual effort by identifying the code clones on the assembly or the source level. However, security analysts still have to understand the matched assembly or source code to develop an understanding of the functionality, and it is assumed that such a matched candidate always exists. This research bridges the gap by introducing the problem of assembly code summarization. Given the assembly code as input, we propose a machine-learning-based system that can produce human-readable summarizations of the functionalities in the context of code vulnerability analysis. We generate the first assembly code to function summary dataset and propose to leverage the encoder-decoder architecture. With the attention mechanism, it is possible to understand what aspects of the assembly code had the largest impact on generating the summary. Our experiment shows that the proposed solution achieves high accuracy and the Bilingual Evaluation Understudy (BLEU) score. Finally, we have performed case studies on real-life CVE vulnerability cases to better understand the proposed method's performance and practical implications.

Additional Key Words and Phrases: Cyber threat intelligence, Representation learning, Adversarial learning, Vulnerability and malware analysis, Reverse engineering

1 INTRODUCTION

Reverse engineering is the process of understanding the inner working of a software system without having access to the system's source code. It is a critical step in the workflows for firmware security validation, software vulnerability research, malware analysis, and copyright infringement investigation. Typically, reverse engineering is achieved by a manual process of inspecting and understanding the assembly code extracted from the software system. After, a high-level understanding of the inner working and program logits are understood. However, it often requires a significant amount of manual effort.

Recently, data-driven solutions were proposed to reduce the manual effort by identifying the code clones on the assembly or the source level. Machine learning and binary analysis are all vast fields that have been studied and combined extensively in previous years for this purpose. Data-driven binary analysis research has generally been focused on relating the assembly language to its source code counterparts [10, 22, 25, 30] and analyzing the assembly code to further understand how a specific program functions [7, 35]. However, security analysts still have to understand the matched assembly or source code to develop an understanding of the functionality. This implies that security analysts have to be equipped with extensive reverse engineering knowledge. Additionally,

Author's address: Scarlett Taviss*, Steven H. H. Ding*, Mohammad Zulkernine*,
Philippe Charland§, Sudipta Acharya*, * School of Computing, Queen's University, 99 University Ave, Kingston, Canada and § Mission Critical Cyber Security Section, Defence R&D Canada, , Valcartier, Canada, *{scarlett.taviss,steven.ding,mz}@queensu.ca, philippe.charland@drdc-rddc.gc.ca, s.acharya@queensu.ca.

2576-5337/2023/5-ART
https://doi.org/10.1145/3592623

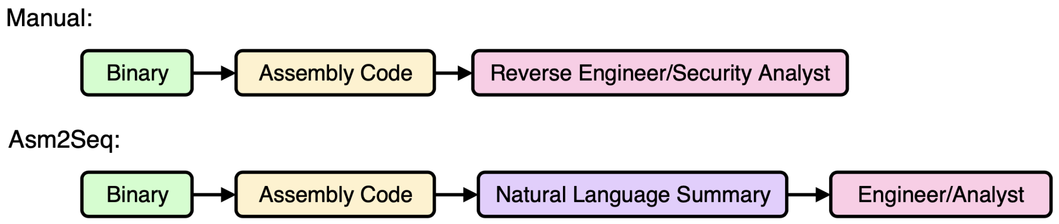


Fig. 1. The simple flow of information from binary to engineer/analyst for two different models, manual reverse engineering and Asm2Seq.

it is assumed that such a matched candidate always exists in order to understand the target. In this study, we propose to generate high-level natural language summaries directly from the assembly code, removing the need to understand low-level code and logic. The literature on machine learning for summarization is primarily focused on summarizing natural language texts [2][3][9][17][19][27][33] and source code [14][20][21][24][34]. Machine learning techniques have been widely used to improve the results in all of these cases. This study presents the first work in assembly code summarization.

Figure 1 shows the flow of information for manual reverse engineering and the Asm2Seq (assembly-to-sequence) model. Adding the assembly-to-sequence step gives reverse engineers and security analysts a head start to decode a binary program. A solution to the presented problem will be helpful in 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 summaries of the malicious code will not only guide malware analysts but also allows individuals to understand better and grasp the concepts of how malicious software functions on their devices. Although we don't claim that our automated system can fully replace the knowledge and expertise of a reverse engineer, but the generated summary from our system can be used as a trusted head start towards reverse engineering by a malware/security analyst. Another security-related aspect that will benefit from this research is security adaptations for existing hardware and software. This is becoming a significant issue with security measures in the modern world which always require updates to ensure users are protected. Understanding the binary code with security issues will allow reverse engineers and program developers to produce results faster and potentially better solutions. These improvements and modifications will be mainly due to an increased understanding of what the computer does with a given source code. The contributions of the presented study can be summarized as follows.

- We define the assembly code summary generation problem, which is vital in the domain of reverse engineering for vulnerability and malware analysis. 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. According to the best of our knowledge, we are among the first to study this problem.
- We created the first labelled dataset for binary code summarization. Specifically, we compile the Juliet Test Suite [6] and the NDSS18 [23] 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 for future work on developing an AI model that can truly understand the semantics of assembly code. The complete datasets are available at https://github.com/L1NNA/Juliet-NDSS18-BinaryCodeSum.
- This study presents one of the first feasible solutions to this problem. We evaluated the proposed solution with the created vulnerability datasets and show that it can adequately predict English sentence outputs for sections of assembly code.

- 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 reverse engineers and security analysts understand why a certain functional summary is generated.

After significant training, the model demonstrates a decent ability to predict natural language descriptions from assembly code. The results also show how the model could learn not only the dependencies between the input and output, but also some of the meaning behind the words. This was observed when comparing the predictions from the model with their true descriptions. This comparison showed how the model was able to predict "non-negative" as opposed to the true description "larger than zero" with the remaining prediction matching perfectly.

2 RELATED WORKS AND MOTIVATION

In this section, we have discussed some related research works. We surveyed on aligning assembly code to natural language. Some existing research works focus on the alignment of assembly code to source code. Again some works perform source code to natural language alignment tasks. However, we have yet to have significant work performed on assembly code to generate natural language summaries. Some of the relevant existing works are discussed in brief as follows.

2.1 Text Summarization and Generation

The creation of text summaries using machine learning can be generalized down to inputting one piece of text and outputting another. Traditional text summarization techniques revolved around analyzing different aspects of the text. In 2014, Sutskever et al. [32] and Cho et al. [9] introduced models that look 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[32] [9]. Both studies translated English to French using encoder-decoder models trained to learn phrase representations, achieving favourable results compared to older models.

Tezcan et al. used a neural network architecture to detect grammatical errors in statistical machine translation [33]. Similarly and more recently, Muñoz-Valero et al. used recurrent neural networks (RNN) to identify the subject and predicate of sentences [27]. Islam et al. applied a long short-term memory (LSTM) network to generate Bangla text sequences, a difficult task, given the limited resources in the Bangla field and the noise incurred during data processing [19]. Similar developments are being seen concerning code summarizations as well.

2.2 Code Summarization and Generation

The main goal of code summarization is to reduce functions and complex methods to individual terms that, when combined, provide an accurate summary of the core functionality. 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 [31]. Studies include Iyer et al., who created natural language summaries of C# and SQL source code using an LSTM model with attention [20]. In addition, LeClair et al. generated natural language summaries of program subroutines using an attentional GRU encoder-decoder model [24].

A more simplified approach models code summarization as a machine translation problem, where the input is represented through a sequence of tokens, and the output is the natural language equivalent [3]. Allamanis et al., Iyer et al., LeClair et al., and many others have successfully used this approach for their respective research. Using a similar approach for this study, previous studies are vital to understanding what worked and could be improved. In particular, Allamanis et al. used a convolution neural network to predict short and descriptive names for a given snippet of code using a statistical rating method [2].

A more complicated approach to this problem was presented by Alon et al., who use the syntactic structure of programming languages to represent a piece of code as a set of compositional paths over its abstract syntax tree.

The overall result is the top travelled path. The results of the research by Alon et al. outperform the baselines used in the study, including the research conducted by Allamanis et al. [3]. Alon et al. take their research further by applying their model to short pieces of C# code to reproduce a complete natural language sentence [3]. However, this study lacks a full dataset comparable to code summarization research, making it unclear how well this model will perform on larger programs.

Generating method names is a very useful application of code generation, because clear and concise method names are critical for the readability and maintainability of programs [21]. A different study by Allamanis et al. looked 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].

Wei et al. capitalized on the task of code summarization and trained a model to summarize and generate code simultaneously [34]. Their goal for code generation went beyond method names by generating source code from natural language intent [34]. An encoder-decoder architecture was first applied to code generation in this way by Ling et al., who took natural language descriptions as input and outputted source code through the decoder [26].

2.3 Binary Analysis

Binary analysis is known to be very useful for malware analysis and patching software against known vulnerabilities [5][30][10]. Machine learning approaches for 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 several key aspects and how they can be applied to binary analysis problems [35]. Identifying the functionality of binary instructions gives rise to identifying a method or section of source code. This aspect makes relating source code to its assembly counterpart an important field of research within binary analysis.

Relating Source Code to Assembly Instructions: Early work in the field attempted to directly decompile x86 assembly instructions to C code, followed by a full software analysis of the results [7]. Although successful, this method has become outdated due to machine learning and artificial intelligence advancements. It is often used as a baseline in later research. Bao et al. improve on the models introduced by Brumley et al. by using a machine-learning approach to create a weighted prefix tree of start bytes of instruction sequences for functions [5]. Their approach produced better results than previously tested models, including the commonly known disassembly tool IDA Pro[1]. This model focused on relating source code to binary code, but explanations as to the functionality of the resulting code need to be explored.

Shin et al. propose an RNN architecture that takes bytes of a binary as input and predicts whether a function boundary is present[30]. Results are comparable to or better than previous models, with drastic reductions in computation time observed when using the RNN model. A similar study using RNNs predicts function types from disassembled binary code functions [10]. This research utilizes 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 automatic summarization of the results.

Levy and Wolf propose an architecture for aligning the source code to its compiled binary equivalent to fully relate the two languages. Two LSTM encoders are used to create representations of the source code statements and binary code statements, respectively, followed by a CNN decoder to transform the statement pairs into alignment scores [25]. This study could be extended to summarize the assembly code, and the information already identified through the alignment process. Katz et al. take on a similar approach that uses RNNs by proposing an LSTM-based encoder-decoder model as a decompiler [22]. Binary code is inputted to the encoder to translate it to source code through the output of the decoder [22].

[1]https://hex-rays.com/ida-pro/

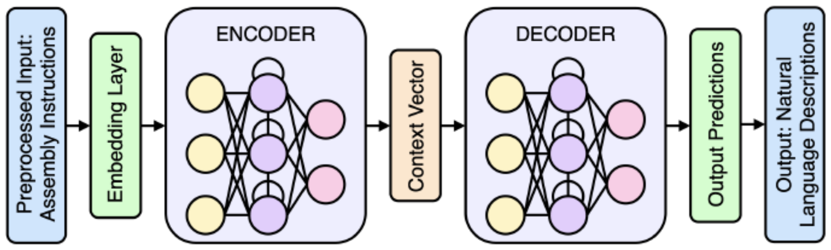

Preprocessed Input: Assembly Instructions
Embedding Layer
ENCODER
Context Vector
DECODER
Output Predictions
Output: Natural Language Descriptions

Fig. 2. An overview of the basic model schematic with the encoder and decoder networks being either LSTM, GRU, or bidirectional models.

We have found no existing literature on generating natural language summaries from the assembly code of a software system. Hence according to best of our knowledge, this is the first-ever attempt to automate natural language summary generation task from assembly language.

Also, from the survey, we observed that in text summarization [9, 19, 32], code summarization [20, 24], and method name generation [1, 21] tasks encoder-decoder model, LSTM model, and GRU encoder-decoder model were proven to perform best. Also, our survey on existing machine-learning based binary analysis has shown that the RNN, CNN, LSTM encoder/decoder models were popularly utilized for relating source code to assembly instructions [10, 22, 25]. However, these well-established and popular models have not yet been utilized to address our chosen problem. Our research focuses on analyzing and seeing how these existing well-known sequence-to-sequence models perform the natural language summary generation task from assembly input. This factor motivates us to choose these models and perform a comparative experimental analysis on chosen datasets.

3 METHODOLOGY OVERVIEW

This section goes more in-depth into the calculations behind the processes and procedures. The problem addresses the translation from assembly code to natural language and is defined in Equation 1 and shown pictorially in Figure 2. Assembly code is inputted into the model, where it is translated into natural language outputs that describe what the code does about the vulnerability of that file.

$$x \longrightarrow E \longrightarrow CV \longrightarrow D \longrightarrow y \tag{1}$$

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. Sequence-to-sequence learning is about training models to convert sequences from one domain into sequences 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 the output. However, in the general sequence-to-sequence case, input and output sequences have different lengths. When the lengths differ, the entire input sequence is needed to predict the target output. To address the different lengths in the input and output sequences, an encoder and decoder are used to help find relations within the data. Figure 3 depicts a simplified version of the components of an encoder-decoder model.

3.1 Encoder

An encoder is a neural network that processes the variable-length input sequences to a fixed-length vector [9][4]. The resultant vector is often called the context vector and is supposed to be a good summary of the entire input

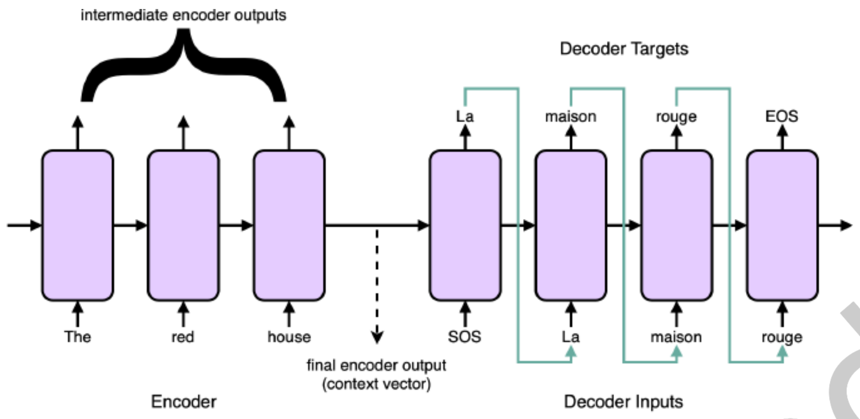


Fig. 3. Encoder-Decoder basic schematic.

sequence. For short 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 the performance of the model declines [4][8]. The left-hand side of Figure 3 depicts the encoder model. The final output of the encoder, the context vector, is passed to the decoder as its initial state.

3.2 Decoder

A decoder maps the fixed-length vectors back into variable-length sequences [9]. 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 3 depicts the decoder model. Beginning with a Start-Of-Sequence token, the target data is offset by a 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 depends on the type of problem being solved. When dealing with sequential data, it can be beneficial to have information preserved from the past and future for predicting outputs. It is possible to obtain this information using bidirectional networks.

4 ENCODING ASSEMBLY CODE

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. 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 4.

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 because assembly code is a direct representation of a possibly very complex program that has been broken down into thousands of individual steps. These concepts are important to understand because this research extracts the mnemonics and operands from the assembly code of compiled programs to use them as input in machine learning algorithms. The goal is to convert these machine instructions into human-readable English natural language descriptions.

4.1 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 encoders. These encoders use different methods to find these relations.

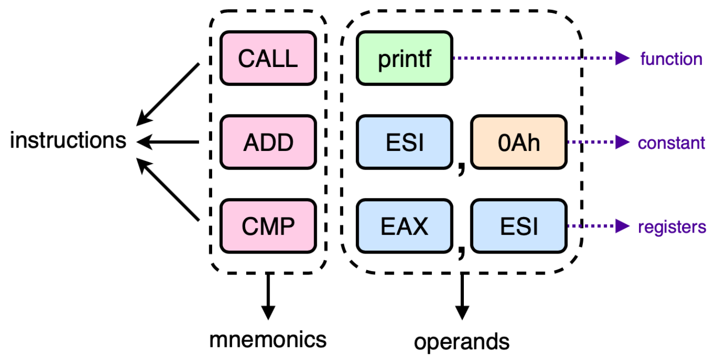


Fig. 4. Examples of assembly code instructions.

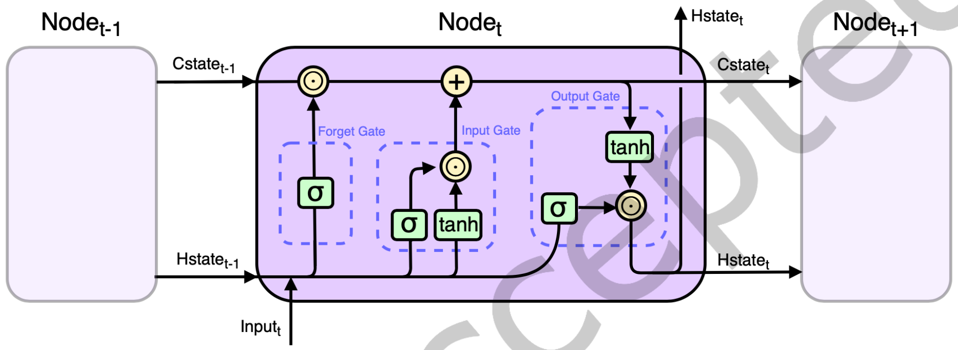


Fig. 5. LSTM diagram of a single node. Circular operations depict element-wise addition and multiplication. Rectangular operations represent activation functions.

4.1.1 LSTM as the Encoder. 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 [16]. A diagram of a simple LSTM can be seen in Figure 5.

LSTM neural networks were used to prevent important information from being lost while the model was trained, thus ensuring more accurate predictions overall. LSTM models preserve long-term information from past inputs using their cell state and short-term information from past inputs using their hidden state. The final hidden state and cell state are combined to form the context vector - *CV* for the encoder that will be passed to the decoder as its initial state. Equation 2 represents how the context vector for an LSTM network was calculated.

$$CV_{LSTM} = F(h_f^{LSTM}, c_f) \tag{2}$$

Where h_f and c_f are the final hidden and cell states of the encoder, respectively, encoder and F is the function used to concatenate the two states.

4.1.2 GRU as the Encoder. Gated Recurrent Unit (GRU) neural networks are composed of a hidden state, a reset gate, and an update gate [11]. It works similarly to LSTMs, but reduces the number of gates used and only passes a hidden state through the network, eliminating the cell state. The information from the LSTM cell state is incorporated into the hidden state through the gates used in a GRU network. With GRUs, only the final hidden state is used, as the context vector is passed to the decoder as its initial input. Equation 3 represents how the

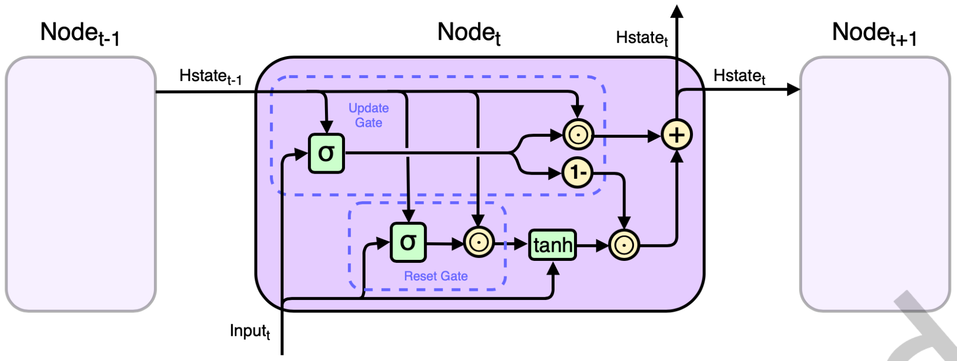


Fig. 6. GRU diagram of a single node. Circular operations depict element-wise operations. Rectangular operations represent activation functions.

context vector for a GRU network was calculated.

$$CV_{GRU} = h_f^{GRU} \tag{3}$$

Where h_f is the final hidden state from the encoder.

4.1.3 Bidirectional Models. Bidirectional networks preserve information by integrating past and future inputs into their decision to predict an output [15]. These models are composed of two separate RNNs connected to the same output layer, making the model a cohesive unit [18][12]. Many sequence labelling problems can benefit from having access to both the future and past contexts of a given sequence [16]. 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.

The forward layer output sequence and backward layer output sequence are respectively denoted by $\overrightarrow{h_*}$ and $\overleftarrow{h_*}$, where $*$ represents time-steps $t - n$ to $t - 1$. Combining these output sequences gives the following equation as the context vector for the encoder, CV^{Bi}, for the model at time t:

$$CV_t^{Bi} = F(\overrightarrow{h_{*(t)}}, \overleftarrow{h_{*(t)}}) \tag{4}$$

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

5 DECODING NATURAL LANGUAGE

The decoding process takes the final encodings from the encoder in the form of a context vector 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 Start-Of-Sequence (SOS) token, all other inputs to decoder nodes are the output of the previous unit, as shown in Figure 3. The context vector is inputted to the decoder, and the output from the decoder is calculated as follows.

$$y_t = D(h_{d(t-1)}, y_{t-1}) \tag{5}$$

where,

$$y_{t-1} = \begin{cases} y_{t-1}, & \text{if } t \geq 1 \\ SOS, & \text{otherwise} \end{cases} \tag{6}$$

and,

$$h_{d(t-1)} = \begin{cases} h_{d(t-1)}, & \text{if } t \geq 1 \\ CV_{*}, & \text{otherwise} \end{cases} \tag{7}$$

Where $*$ represents the model being used.

An End-Of-Sentence (EOS) token is passed to declare the end of the current sequence. The model repeats this process with the remaining sequences in the dataset until all predictions have been made.

5.1 Attention-Based Context Representation

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 concerning output predictions and pays "attention" to those inputs more so than for others in future iterations. When an attention mechanism is applied, the intermediate encoder states, represented in Figure 3, are used instead of being discarded. Rather than encoding a whole input sequence into a single fixed-length vector, attention models encode input sequences into a series of vectors and choose a subset of these vectors as the model decodes and predicts an output [4]. In addition 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 normalize the scores so they can be treated as probabilities. The higher the score, the higher the probability the input will be relevant to the current output. The context vector is produced by applying the probability weights to the encoder 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 the predicted output.

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 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. This is to confirm that the model is working based on learned dependencies that make sense in the real world. The attention mechanism used 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 a query, q, for decoder states, and a 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. The equation for the alignment scores is then given as Equation 8.

$$e_{t,i} = q_t \cdot k_i^T \tag{8}$$

Where k^T is the transpose of k, needed to satisfy matrix dimension multiplication requirements. The softmax function applied to the scores gives Equation 9. The outputs are used as the weights when computing the context vector.

$$\alpha_{t,i} = \frac{exp(e_{t,i})}{\sum_{k=1}^{n} exp(e_{i,k})} \tag{9}$$

The context vector for the output at the current time-step, y_t, is the sum of the weights multiplied by the encoder's 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 Equation 10.

$$CV_t = \sum_i = 1^n \alpha_{t,i} v_i \tag{10}$$

The context vectors are passed to the decoder and used on a per-output basis, given that 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 for predicting each output. This final experiment provides information explaining how the model makes its choices for output.

6 OPTIMIZATION

Optimization for neural networks uses gradient descent algorithms to minimize or maximize an objective function. Mini-batch gradient descent computes the gradient of the objective function for mini-batches of data within each iteration of training, and adjusts the parameters for each training example [29].

6.1 RMSprop Algorithm

Root Mean Squared Propagation (RMSprop) [13] is the gradient descent algorithm used for training the model due to its adaptive learning ability and its use of mini-batches. It keeps a constantly moving average of the squared gradient and appropriately adjusts the weight updates 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 adaptive learning rate that optimizes its loss, thus reducing the loss as a whole.

6.2 Objective Function

The objective function being minimized is sparse categorical cross-entropy. 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 statistically the best target predictions. To optimize the model, it must know when to stop training so that overfitting does not occur. This is why early stopping is introduced.

6.3 Early Stopping

Early stopping is set by choosing an output parameter of the model to minimize or maximize. In this research, output parameters used for early stopping include 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 is minimized, and early stopping on the accuracy parameters is maximized. Five epochs of increased loss values from the minimum or five epochs of decreased accuracy values from the maximum are allowed before the model stops. This level of patience was to ensure that minimal optimization loss occurred to maximize the training process. All experiments underwent early stopping for each output parameter to find the best model and most accurate early stopping technique for this dataset.

7 EVALUATION METRICS

Among many existing evaluation metrices, we have chosen four relevant ones to determine our model’s efficacy. They are Bilingual Evaluation Understudy (BLEU), accuracy, recall, and precision.

Bilingual Evaluation Understudy Score (BLEU). The Bilingual Evaluation Understudy Score is a popular metric for automatic machine translation [28] by evaluating a generated sentence against one or more reference sentences. It is calculated by the measuring geometric mean of the test corpus’ modified precision scores and then multiplying the result by an exponential brevity penalty factor. The score ranges from 0 to 1, with a score of 1 being a perfect match and a score of 0 being a perfect mismatch.

Adjusted Accuracy and Overall Accuracy. The adjusted accuracy is used to determine if the model can predict 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 do not match the expected result are not used as the input for the following word prediction. Using the same example from Figure 3, which depicts the encoder-decoder model, the difference in the decoder predictions in terms of how the accuracy is determined can

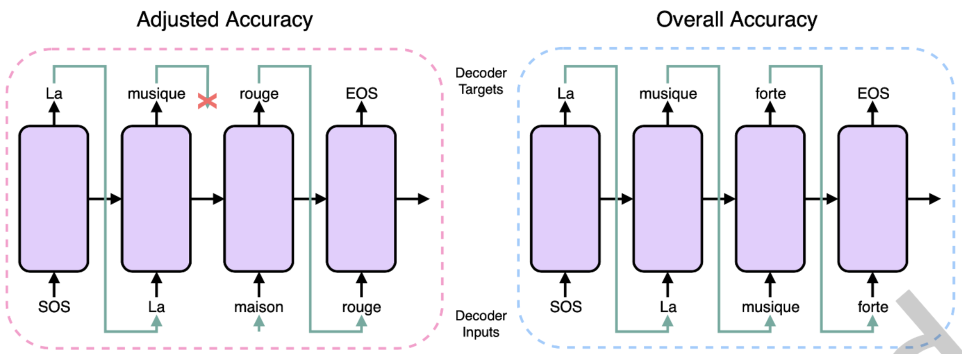


Fig. 7. Demonstrating the differences between how the adjusted accuracy and the overall accuracy are determined.

be seen in Figure 7. The left-hand side of Figure 7 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 the model, as we may not have outputted *rouge* without the *maison* input.

Whereas the overall accuracy computes how accurate the model performs on its own. Whatever is the output from the decoder, it is passed as the next input to the decoder. The right-hand side of Figure 7 shows the target outputs being passed back to the decoder as inputs until the EOS token is reached. 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.

To evaluate our obtained results, we have reported both kinds of accuracy measures (adjusted and overall) in this article. It is because they capture two different aspects of the accuracy of the model. For example, incorrect target output from one earlier layer of the decoder (as shown in Figure 7) can misalign all later outputs hence causing reduced overall accuracy. Hence, adjusted accuracy is used to check how accurately each layer of the corresponding decoder model works when a right input is given.

Precision & Recall. Where accuracy looks at the final predicted output as a whole and compares it with the expected output, precision and recall look at the individual words. Precision, and recall are accuracy measurements on a word-to-word basis for a dataset.

8 EXPERIMENTAL EVALUATION

8.1 Dataset

The Juliet Test Suite is the primary dataset used unless otherwise stated. A secondary one, the NDSS18 dataset, is also tested to ensure the model is not biased toward a specific dataset. Both the datasets are established and well-known vulnerability dataset where all of the source code samples contain some sort of software vulnerabilities. Some of the vulnerabilities present involve process control, stack-based buffer overflow, information leakage, uncaught exceptions, and division by zero. However, there are no existing benchmark assembly code vulnerability datasets available so far. Therefore, to prepare these two datasets suitable for this work (assembly to natural language summaries), we have performed feature engineering on them by compiling the source code to assembly language files.

Both datasets consist of source code samples, which have been compiled into assembly language files. These files can be divided into good and bad cases. The good cases have eliminated the vulnerabilities, whereas the bad cases showcase the software vulnerability for that sample.

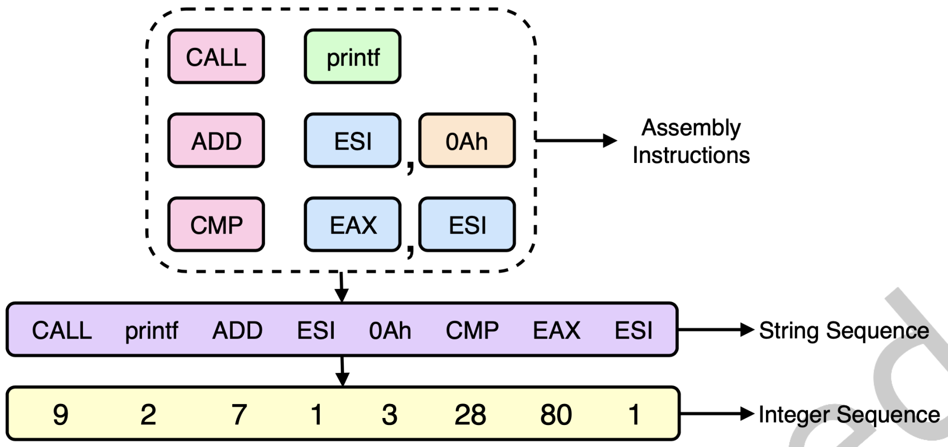


Fig. 8. A sample representation of the pre-processing procedure for the assembly code input.

Discovering vulnerabilities in source code is important for software maintenance, software analysis, and software development, but it is very time-consuming process. This dataset consists of thousands of source code files highlighting their vulnerabilities or how they have been mitigated. Compiling the source code files gives their equivalent assembly code files with the contained vulnerabilities. The Juliet Test Suite is one of the first datasets 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 [23]. With the goal of this research being to map assembly code to natural language, these are ideal datasets.

8.1.1 Data Extraction. Data extraction is simplified by the commonalities found within the files of both datasets. The naming convention for all files, source and assembly code, indicates whether that particular file is a good or a bad case. This distinction allows for matching good and bad assembly code files to good and bad source code files, respectively. The naming convention for the files makes it possible to relate all source files to their corresponding assembly code file. With an understanding of what files are correlated, it is possible to extract the necessary information needed for the model to learn the relations.

To convert the assembly code into usable input for the model, we must first compile the source code and then disassemble the resultant assembly code to identify the different blocks of code. Once the assembly code is organized, it is possible to iterate through each block of code and extract the mnemonics and operands within each instruction. The instructions for each assembly code file are combined into a string of tokens, which 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 file 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.

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 or a bad case, the appropriate information is extracted from the header of the source code and processed into a usable format for the natural text output of the model.

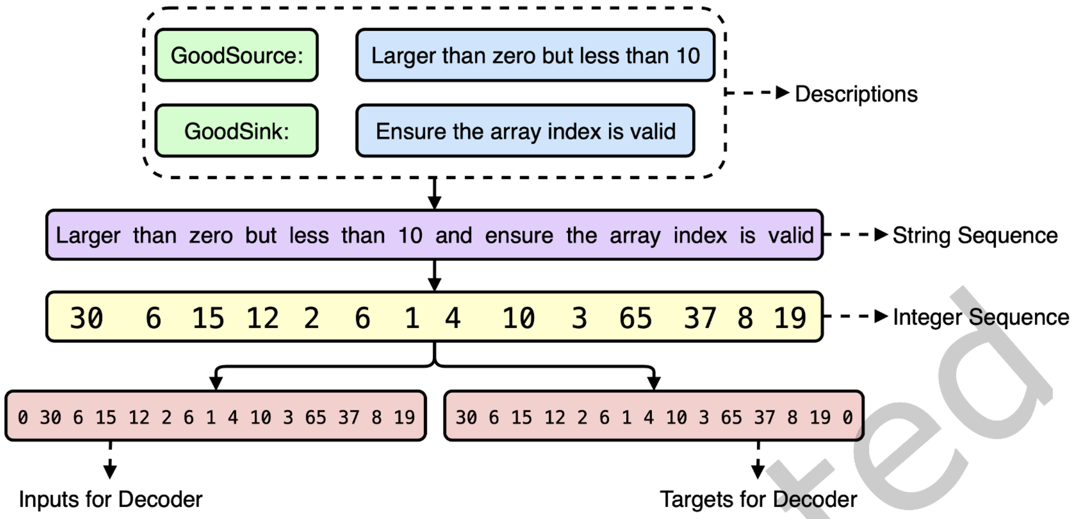


Fig. 9. A sample representation of the pre-processing procedure for the natural description output.

8.1.2 Data Pre-processing. After extraction, pre-processing of the data is needed to convert the data into a usable format for the model. The assembly code instructions and natural language descriptions are sequences of strings that 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 has a maximum sequence length of 256 and corresponds to the index of a token in the dictionary. Sequences are truncated or padded with zeroes until they have the appropriate length to ensure all samples are of the same length. A simple example of this pre-processing step can be seen in Figure 9, where arbitrary assembly code instructions are combined and then converted to integers.

An embedding layer is the final pre-processing step for the input data. It is used to transform individual integers into mathematical objects that can be operated on [15]. The distance between vectors represents the distance between words, making it easier for the model to generalize the behaviour from one word to another during the training process [15].

The natural language outputs require additional pre-processing. The sequences of integers for the decoder input descriptions are padded with leading zeros that represent the start-of-sequence (SOS) tokens, needed for the model 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. This is needed for the model to know when the current predicted sequence is complete. The additional step of padding the data is shown in Figure 9.

The SOS and EOS tokens tell the model the boundaries of each input to ensure it does not group sequences together and learn incorrect dependencies and patterns. The padding process also ensures that the target data is a one-time step ahead, a requirement for training the model to predict the next character of the target sequence given the previous characters of the target sequence.

8.1.3 Data characteristics. The Juliet Test Suite dataset consisted of 272,431 files, which is the combined set of assembly and source code files. After some initial pre-processing 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 pairs, leaving 20,832 pairs for testing. The pre-processed Juliet test suite and NDSS18 datasets can be found in https://github.com/L1NNA/Juliet-NDSS18-BinaryCodeSum.

The NDSS18 dataset consisted of 35,955 Windows samples and 28,608 Linux samples of both assembly and source code files. After initial pre-processing to eliminate incorrectly formatted files, the Windows dataset to 7,160 assembly-description pairs and the Linux dataset was reduced to 7,006 assembly-description pairs. Combining these samples and splitting them into training and testing sets give 10,624 training and 3,543 testing samples.

8.2 Experiments

Each experiment was conducted 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 training set is the sample of data used to fit the model, where it learns the appropriate weights and biases. The validation set is the sample of data used to evaluate the model on the training dataset and is frequently used to fine-tune the hyperparameters. The testing set is the sample of data used to evaluate the final model after training is complete.

The hyperparameters are chosen based on several factors, such as the dataset's size, the GPU's memory space, and the type of data used. For example, the batch size is one of the most common hyperparameters to tune and can significantly impact how fast the model can converge on its loss function. For this research, 192 samples were set as the batch size to max it out based on the GPU memory to ensure we get a more accurate estimate of the gradient for the batch.

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

Additional hyperparameters are assigned values commonly found favourable in sequence-to-sequence networks or based on the dataset's size. After some trial and error, the number of epochs was set to a maximum of 150. The maximum sequence length for the encoding process was set to 256. This ensures that all tokens in the description sequences were included unless they were 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 determines 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 to 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. The final encoder vector aims to encapsulate all the necessary information from the input elements to help the decoder make an accurate prediction.

The decoder for the model 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 through several decoder layers until a final output is given. A softmax function is used to normalize the final output into a probability distribution that 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.

9 RESULTS

This section presents findings for the different datasets and evaluation metrics discussed. When evaluating experiments, it is important to identify whether or not the models were capable of learning the appropriate dependencies. The addition of the attention mechanism gives insight into what aspects of the input is key to predicting the correct output, which is important to understand further how the model works.

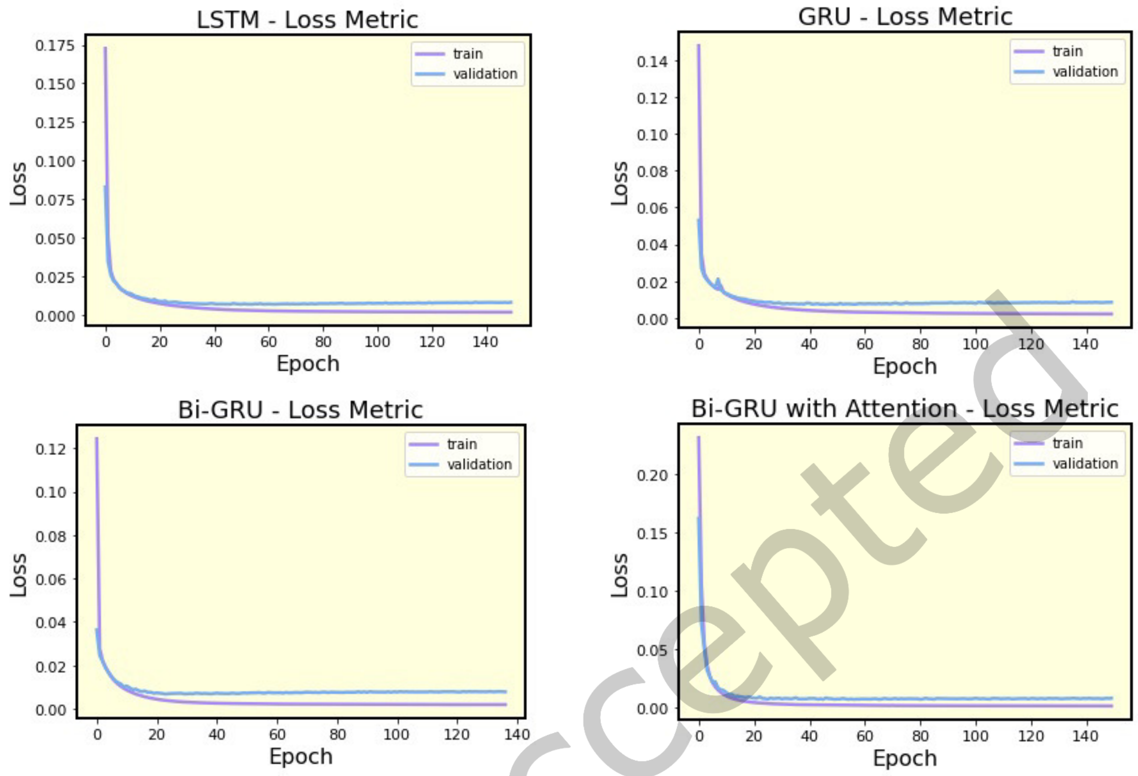


Fig. 10. Loss metric convergence during training for four different models, LSTM, GRU, Bi-GRU, and Bi-GRU with Attention.

Training Convergence. Models with early stopping on the loss parameter produced the best outcomes. Models completed every epoch or came close to completing every epoch with early stopping on the loss metric. This means that these experiments continued to improve, although slightly until the training was complete. Figure 10 depicts the loss convergence of four different experiments. Another key point to note is how quickly the models converge: the loss value evens out within roughly 20 iterations for each.

The Bi-GRU experiment produced the best results with early stopping on the sparse categorical accuracy metric (SCA). Figure 11 shows the convergence of SCA for the GRU and Bi-GRU models as they produced the best results with early stopping on SCA. Similar to the loss metric, both models converge quite quickly within 20 iterations of the training process before evening out. Early stopping is favorable to ensure overfitting does not occur.

Obtained BLEU Scores: The obtained BLEU scores corresponding to the output summary for each of the five tested sequence-to-sequence models on both datasets are reported in Table1. In our work's current context, BLEU score indicates how well the predicted outputs match one of the extracted descriptions, which also means how readable the predicted outputs are.

It is important to note that an increased number of reference sequences per translation will result in a higher BLEU score, as there are more options for the prediction sequence to match correctly. For example, 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 [28].

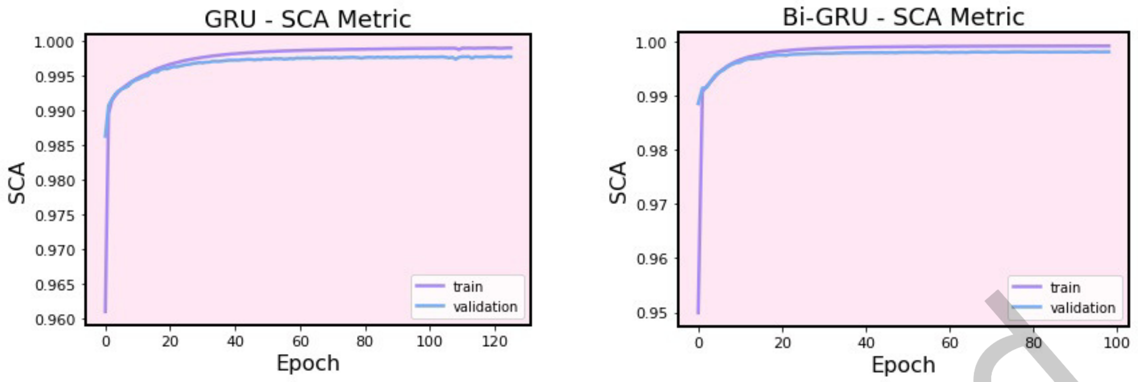


Fig. 11. Sparse Categorical Accuracy (SCA) metric convergence during training for the two best models with early stopping on SCA.

Table 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

In this work, the complete set of extracted descriptions is inputted as the reference sequences when calculating the BLEU score, resulting in 20,832 possible reference sequences for the Juliet Test Suite dataset and 3,542 reference sequences for the NDSS18 used for matching per generated sequence. For a fair comparison, the BLEU scores of corresponding summaries from all five sequence-to-sequence models were calculated using same number of reference sequences for each dataset as mentioned above. As reported in Table 1, we can see Bi-GRU model with attention model performs best for both datasets (BLEU score 0.537 for Juliet Test Suite and 0.880 for NDSS18 dataset) compared to other four models in terms of quality of obtained summaries.

Moreover, we have compared our obtained BLEU scores of best-performing model with two previous studies (LeClair et al. [24] and Iyer et al. [20]) with similarly sized reference sequences. LeClair et al. used just over 100,000 reference sequences for calculating their BLEU score and achieved a best result of 0.209 [24]. Whereas, Iyer et al. used less than 10,000 reference sequences and achieved the best result of 0.205 [20]. Our obtained highest BLEU score of 0.537 for the Juliet Test Suite and 0.880 for the NDSS18 dataset are relatively better than both of these previous works despite our much lower reference sequence sets size. This observation leads to the conclusion that our model was trained well and capable of generating high-quality natural language descriptions for both of the chosen datasets.

Obtained Accuracy: The resultant accuracy measurements for the Juliet Test Suite and NDSS18 dataset are provided in Table 2 and Table 3, respectively. As we can see there, most accuracy measurements produce favourable results. The high values for the adjusted accuracy indicate 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 decrease in comparison because a correct input is not always given, resulting in the overall output sequence naturally having fewer correctly predicted target sequences.

According to our obtained results, the highest adjusted accuracy and overall accuracy of 99.06% and 79.6%, respectively, are obtained by the Bi-GRU model with attention model for the Juliet Test Suite dataset. Similarly, for NDSS18 dataset also the Bi-GRU model with attention model performs best with 98.1% adjusted accuracy and 84.1% overall accuracy. The increased overall accuracy for NDSS18 dataset compared to the Juliet Test suite

Table 2. Accuracy results for the Juliet Test Suite dataset.

Seq2Seq Model	Adjusted Accuracy (%)	Overall Accuracy (%)
LSTM	99.28	77.7
Bi-LSTM	99.8	74.9
GRU	99.8	76.3
Bi-GRU	99.8	79.2
Bi-GRU+Attention	99.06	79.6

Table 3. Accuracy results for the NDSS18 dataset.

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

could be due to the reduced size of the NDSS18 dataset. Concerning 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 causes to produce better results overall. The model is likely able to learn dependencies better because the variance in possible outputs is significantly reduced.

Obtained 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 correct in the context of the predicted target sequence. In our conducted experiments, we observed that Precision and Recall produced consistent results across all models for the Juliet Test Suite and NDSS18 dataset, as shown in Table 4 and 5, respectively. For the Juliet test suite dataset, the best model achieves 68.6% for recall, which indicates that the correct output will be predicted 68.6% of the time a correct input is given. Of those 68.6% correctly flagged target tokens, 77.9% of them are the actual correct target output in terms of the current sample being predicted.

Table 4. Precision & Recall results for Juliet Test Suite dataset.

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

On the other side, most of the Precision and recall results for the NDSS18 dataset are almost half of those presented for the Juliet Test Suite data. These lower values could be due to the reduced size of the dataset. With a limited number of samples, every incorrect result has more of an impact on the final percentages when compared to a much larger dataset. However, also for this dataset we observe that the Bi-GRU model with attention performs best with 68.6% of Precision and 60% of Recall.

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. For example, a predicted output of “non-negative” vs. the true target

Table 5. Precision & Recall results for NDSS18 dataset.

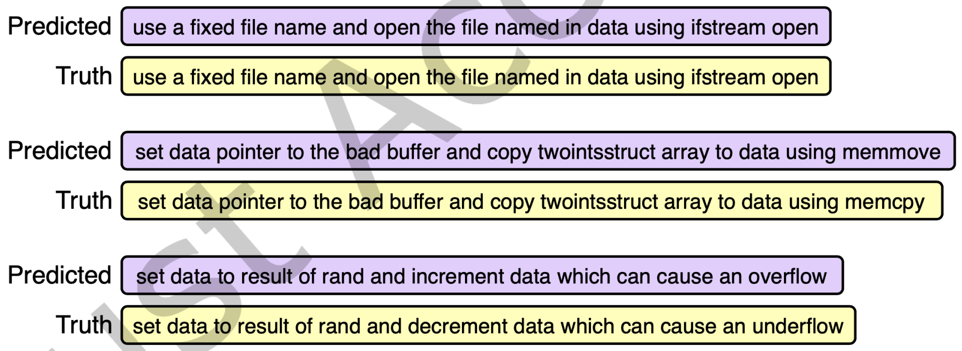
Seq2Seq Model	Precision (%)	Recall (%)
LSTM	37.0	32.9
Bi-LSTM	40.0	31.8
GRU	31.0	26.8
Bi-GRU	38.6	32.7
Bi-GRU+Attention	68.6	60.0

of "greater than zero" will reduce precision and recall, as the input-to-output pair and the output itself will be flagged as incorrect. The interpretation of these two outputs has the same meaning, making this an appropriate translation, even though the metrics are unable to detect it. However, it is a common issue for sentence generation tasks, and unfortunately, there is no proven way to verify whether two sentences are semantically equivalent. In the following section, we have manually evaluated a few such false instances and analysed them in detail.

9.1 Prediction vs. Truth analysis

The results from the metrics analysis can be expanded and demonstrated by comparing the predicted outputs to their true descriptions. Some interesting dataset samples of predicted descriptions compared to true descriptions for two of the best models are displayed in Figures 12 and 13.

Figure 12 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 can correctly predict a description that matches a true description perfectly. The second and third samples are used to demonstrate different ways the model has learned dependencies.



Predicted: use a fixed file name and open the file named in data using ifstream open
Truth: use a fixed file name and open the file named in data using ifstream open

Predicted: set data pointer to the bad buffer and copy twointsstruct array to data using memmove
Truth: set data pointer to the bad buffer and copy twointsstruct array to data using memcpy

Predicted: set data to result of rand and increment data which can cause an overflow
Truth: set data to result of rand and decrement data which can cause an underflow

Fig. 12. Samples of the predicted output compared to the true description for the GRU model.

The second sample in Figure 12 matches completely, up to 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 of how recall increases but decreases precision. Both descriptions produce appropriate outputs given the correct input, increasing the recall, but the final output is not correct given the true descriptions, which would decrease the precision.

The third sample is an exciting 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 is well-trained to predict "overflow" in accompaniment with "increment" and "underflow" in accompaniment with "decrement". This proves although it is the wrong description, it is still displaying the model's ability to learn patterns within the data.

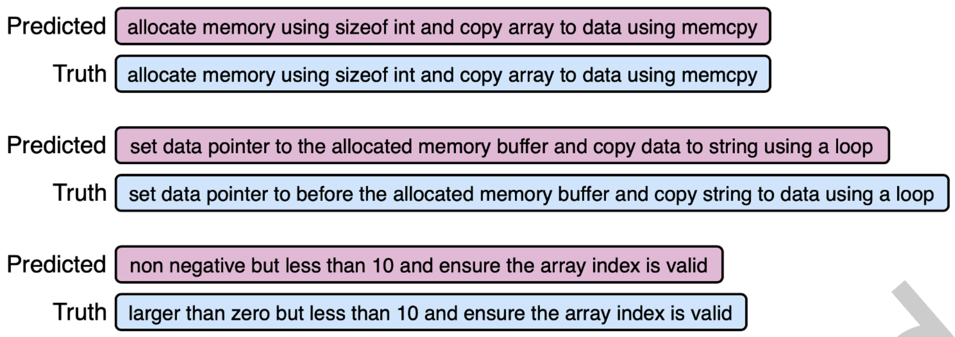

Predicted
allocate memory using sizeof int and copy array to data using memcpy
Truth
allocate memory using sizeof int and copy array to data using memcpy
Predicted
set data pointer to the allocated memory buffer and copy data to string using a loop
Truth
set data pointer to before the allocated memory buffer and copy string to data using a loop
Predicted
non negative but less than 10 and ensure the array index is valid
Truth
larger than zero but less than 10 and ensure the array index is valid

Fig. 13. Samples of the predicted output compared to the true description for the Bi-GRU model.

Figure 13 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.

The second sample demonstrates how, even though the model did not produce the correct output, the information is still applicable. The predicted output can be used as a starting point for further analysis of the assembly code, where additional information can fill in the blanks. This predicted output reduces the time needed to analyze the original assembly code because the majority of the information needed to explain the vulnerability has been correctly predicted

The last example of Figure 13 is an excellent demonstration of how well the model was able to learn and not just memorize patterns in the data. For example a non-negative integer is the same thing as an integer larger than zero, showcasing how the model was able to understand the meaning behind these words, while still being able to finish the prediction correctly.

This prediction-truth pair is a perfect example of what we hope the model can achieve.

9.2 Attention Scores

Adding attention gives information about what aspects of the input are the 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 6 and 7 showcase some of the assembly instructions and their corresponding attention scores for various words in the predicted natural language output sequences.

Table 6 gives the top two sections of assembly instructions and attention scores for various predictions of the word "data" on the Juliet Test Suite. 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.

Analyzing the assembly instructions, we can see how similar they are. The "rsp" and "rbp" instructions are associated with registers 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 indicate how the model was able to find appropriate correlations within the data to use for predictions.

A similar observation can be seen in Table 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 6, i.e., the correlation between the instructions and the target tokens is even better.

Table 6. Attention scores and assembly instructions are associated with them for various predictions of the word "data".

Assembly Instructions	Attention Scores
'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

Table 7. Attention scores and assembly instructions associated with them are 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 that each function is found in the assembly instructions in addition to the "call" instruction, which indicates how the model identified the tokens as functions. The register instructions also indicate of the model creating appropriate connections, as both of these functions deal with data. The models showed favourable results in all evaluation metrics, but limitations to their ability exist nonetheless.

10 CVE CASE STUDIES

The Juliet and NDSS datasets in the experiments above primarily consist of well-crafted test cases to demonstrate the typical causes of various CWE categories. However, it does not have the vulnerability-irrelevant semantic noisy code or code rearrangements introduced by the contextual or functional requirements of an actual library that has a CVE-recorded vulnerability. Therefore, in this section, we conduct a case study on real-life CVE vulnerabilities from open-source libraries to better understand the proposed method's performance. We focus on qualitatively evaluating the summary generated from the previously trained model from Juliet and NDSS dataset on both in-sample and out-of-sample CWE categories. In-sample CWE implies that the training set already covers the CWE category, but the actual vulnerable case is not. Out-of-sample CWE implies that the training data does not cover the actual CWE of a given CVE and the model may run into the out-of-vocabulary issue where it does not have the words needed in describing the functionalities.

For in-sample CWE, we analyze the well-known Shellshock (CVE-2014-6271) vulnerability presented in Bash version 4.3. Bash shells use environment variables to pass shared procedures across different processes. GNU Bash through 4.3 improperly processes the tailing strings of environment variables, which leads to an arbitrary code execution vulnerability that allows the attackers to execute originally unintended system commands by injecting them into environment variables. It affects a wide range of web service applications such as Apache HTTP Server that use Bash to handle certain requests. This vulnerability corresponds to CWE-78: Improper Neutralization of Special Elements used in an OS Command. The Juliet dataset test cases cover CWE-78. The test cases include many different types of sources, such as sockets, console input, or environment variables, from where the program reads data and then executes the OS command potentially presented in data through various sinks such as the execvp and wexecv API. We apply the previously trained model on Bash 4.3, and it generates the summary below for the vulnerable target function ‘initialize_shell_variables’:

```
read input data and execute command in data and execute command in data
```

The summary does reflect the main functionality of the target. The target function first detects function definitions in the environment variables and loads the syntactically correct ones. The summary does not include the environment variable loading part which seems mismatched with the CVE description at first glance. However, looking deeper into the source code that contains this vulnerability, the environment variables are loaded in the other function and are passed along into the target function as string pointers. Therefore the summary is accurate as it only mentions input data.

```
void initialize_shell_variables (env, privmode)
      char **env;
      int privmode;
{...}
```

We also notice that the phrase ‘ execute command in data’ is repeated twice - which is a typical observation when the encoder-decoder model needs more data and iterations for training. We notice that the summary missed part of the functionalities where the target function tries to extract and verify procedure definitions from environment variables. However, in the context of the current CWE-28, its main issue is about executing commands from the function input, which is typically discouraged.

For out-of-sample CVE, we analyze another critical vulnerability Heartbeat CVE-2014-6271. This vulnerability is presented in OpenSSL 1.0.1 before 1.0.1g, caused by improper handling of Heartbeat packets and allows remote attackers to access restricted process memory area by sending a crafted heartbeat package where the specified package size is significantly larger than the actual package size. It is caused by the function call:

```
memcpy(bp, pl, payload);
```

where the target function copies a user-specified number of bytes, through the ‘payload’ variable, from buffer pl to bp. When the user-specified size is larger than the actual size of pl, the target function will copy the sounding memory data of pl, potentially containing sensitive SSL data, to the bp buffer and returns to the attacker. This vulnerability corresponds to CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer. The Juliet Test Suit we used for model training does include memory-related CWEs such as CWE121 Stack-Based Buffer Overflow. However, CWE-119 is more general and not included in the training data. We feed the corresponding assembly code of the target vulnerable function ‘dtls1_process_heartbeat’. The model generates the summary below:

```
read data of a number does not copy the string and a string
```

The first part of the summary is quite accurate regarding the above buffer overread issue. The target function reads data based on a number. However, the summary does not include the details that the number is user-controllable. Looking at the assembly code, this specific number, representing the number of bytes to be read, is part of the SSL heartbeat data structure. Likely, the model does not have the contextual information on where the given number is coming from, given the scope of the target vulnerable function ‘dtls1_process_heartbeat’. The second part of the summary is inconsistent with the first part - another indication of undertraining and insufficient expressiveness of the model given the limited vocabulary in trained Juliet Test Case descriptions, especially when CWE-119 wasn’t included in the training dataset. In the future, the decoder uptrained from other pre-trained natural language decoders such as BERT can potentially help mitigate this issue.

The case studies conducted are limited by the scope and the coverage due to the difficulties in finding vulnerable binaries with the known exact locations of cause. Based on the current findings, the trained model can generate summaries that help one to identify and understand CVE vulnerability or potentially bad implementation practices, with room for improvements in its expressiveness and richness of details provided. Potentially, the assembly code functionality summarization task can be integrated into the vulnerability discovery process by producing summaries of potential targets identified by other automated tools such as binary-only fuzzers or static code analyzers. The summary can help one better understand or identify the plausible root cause or the CWE of the vulnerability. Especially for data-driven static code analyzers, finding and verifying the vulnerability is time-consuming and such a summary could potentially help reduce the time for analysis, as shown in the above case studies. Alternatively, a good solution to this summarization task enables the comparison between the generated summary and the CWE text summary, as a way to statically analyze the code detecting vulnerable function with a textual explanation.

11 LIMITATIONS AND FUTURE WORKS

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

External Validity: The model is only tested with two datasets, potentially limiting the model's generalizability. This limitation is addressed by splitting the data into training, validation, and testing sets to ensure data overfitting does not occur. In addition, the datasets used are real-world vulnerability data giving actual variations that increase the model's generalizability. Additional datasets would be a good future step to validate the findings further and demonstrate the model's ability to predict different data correctly.

Two other main factors limit the model's transferability: the assembly code input and the English text output. The model is built to have x86/AMD64 assembly code as input, preventing other architectures, such as ARM, from applying without significant adjustments. The model is designed to output English natural language predictions, making it unclear if the current model would be able to function properly on datasets comprised of other languages without additional testing. Future work could expand the inputs or outputs to encompass a more diverse dataset.

Statistical analysis will further validate the results of the model to ensure there is no cloned data. The model also needs to be assessed on more diverse datasets and varying case studies to support the explainability results and justify further the results. This research focuses on vulnerabilities within the data, but applications of this model can be greatly expanded and directed at general functional summaries useful for more general malware analysis and reverse engineering.

Sequence-to-sequence model: The chosen sequence-to-sequence models like LSTM, Bi-LSTM, GRU, Bi-GRU models are well-established and popular models for various related tasks like text summarization, code summarization or binary analysis. However, these models have not been utilized to address our chosen problem yet. This work focuses on analyzing how these existing well-known seq-to-seq models perform the natural language summary generation task from assembly input. In this research, we have shown how successfully the Bi-GRU model with attention outperforms other models to generate the assembly language to natural language summary with an overall accuracy of 79.6% for the Juliet test suite and 84.1 % for the NDSS18 dataset. However, as there is always room for further improvement, we will extend this research by choosing other state-of-the-art seq-to-seq models or proposing modified versions. Authors are working in that direction.

Internal Validity: Numerical stability issues can interfere with the evaluation results of the model and can arise from the gradient descent 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 does not speak against the model's accuracy but explains the outlying results sometimes seen in experimental trials.

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

12 CONCLUSION

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 indicate that the bidirectional GRU network produced the most favourable results, with an overall accuracy of 79.6% and a BLEU score of 0.537 for the Juliet test suite. For the NDSS18 dataset, again it achieves the best with an overall accuracy of 84.1% and a BLEU score of 0.88.

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 registers, 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 model's 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 research presented a feasible solution with explainability for a novel problem using a newly created dataset in a field that continues to grow.

REFERENCES

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

[2] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. In *Proceedings of The 33rd International Conference on Machine Learning*, Vol. 48. PMLR, New York, New York, USA, 2091–2100.

[3] Uri Alon, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. In *International Conference on Learning Representations*. https://openreview.net/forum?id=H1gKYo09tX

[4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. 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. 2014. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 845–860.

[6] Tim Boland and Paul E Black. 2012. Juliet 1. 1 C/C++ and java test suite. *Computer* 45, 10 (2012), 88–90.

[7] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. [n. d.]. BAP: A Binary Analysis Platform. In *Computer Aided Verification", year="2011*. Springer Berlin Heidelberg, Berlin, Heidelberg, 463–469. https://doi.org/10.1007/978-3-642-22110-1_37

[8] Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. 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*. Association for Computational Linguistics, Doha, Qatar, 103–111.

[9] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. (2014). arXiv:1406.1078 [cs.CL]

[10] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. 2017. Neural Nets Can Learn Function Type Signatures From Binaries. In *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC, 99–116.

[11] Junyoung Chung, aglar Gehre, KyungHyun Cho, and Yoshua Bengio. 2014. 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. 2020. 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] EM Dogo, OJ Afolabi, NI Nwulu, Bhekisipho Twala, and CO Aigbavboa. 2018. A comparative analysis of gradient descent-based optimization algorithms on convolutional neural networks. In *2018 international conference on computational techniques, electronics and mechanical systems (CTEMS)*. IEEE, 92–99.

[14] S. Gao, C. Chen, Z. Xing, Y. Ma, W. Song, and S. Lin. 2019. A Neural Model for Method Name Generation from Functional Description. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 414–421.

[15] Yoav Goldberg. 2017. *Neural network methods for natural language processing*. San Rafael, California.

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

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

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

[19] Md. Sanzidul Islam, Sadia Sultana Sharmin Mousumi, Sheikh Abujar, and Syed Akhter Hossain. 2019. Sequence-to-sequence Bangla Sentence Generation with LSTM Recurrent Neural Networks. *Procedia Computer Science* 152 (2019), 51 – 58. International Conference on Pervasive Computing Advances and Applications- PerCAA 2019.

[20] Srini Iyer, Ioannis Konstas, A. Cheung, and Luke Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics* (Berlin, Germany) *(ACL '16)*. 2073–2083.

[21] L. Jiang, H. Liu, and H. Jiang. 2019. Machine Learning Based Recommendation of Method Names: How Far are We. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 602–614.

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

[23] Tue Le, Tuan Nguyen, Trung Le, Dinh Q. Phung, P. Montague, O. Vel, and Lizhen Qu. 2019. Maximal Divergence Sequential Autoencoder for Binary Software Vulnerability Detection. In *ICLR*.

[24] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A Neural Model for Generating Natural Language Summaries of Program Subroutines. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 795–806.

[25] Dor Levy and Lior Wolf. 2017. Learning to Align the Source Code to the Compiled Object Code. In *Proceedings of the 34th International Conference on Machine Learning*, Vol. 70. International Convention Centre, Sydney, Australia, 2043–2051.

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

[27] David Muñoz-Valero, Luis Rodriguez-Benitez, Luis Jimenez-Linares, and Juan Moreno-Garcia. 2020. Using Recurrent Neural Networks for Part-of-Speech Tagging and Subject and Predicate Classification in a Sentence. *International Journal of Computational Intelligence Systems* 13 (2020), 706–716. Issue 1.

[28] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: A Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics* (Philadelphia, Pennsylvania) *(ACL '02)*. Association for Computational Linguistics, 311–318.

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

[30] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing Functions in Binaries with Neural Networks. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 611–626. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/shin

[31] Xiaotao Song, Hailong Sun, Xu Wang, and Jiafei Yan. 2019. A Survey of Automatic Generation of Source Code Comments: Algorithms and Techniques. *IEEE Access* 7 (2019), 111411–111428. https://doi.org/10.1109/ACCESS.2019.2931579

[32] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2* (Montreal, Canada) *(NIPS'14)*. MIT Press, 3104–3112.

[33] Arda Tezcan, Véronique Hoste, and Lieve Macken. 2017. A Neural Network Architecture for Detecting Grammatical Errors in Statistical Machine Translation. *Prague bulletin of mathematical linguistics* 108 (2017), 133–145.
[34] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code Generation as a Dual Task of Code Summarization. *CoRR* abs/1910.05923 (2019).
[35] H. Xue, S. Sun, G. Venkataramani, and T. Lan. 2019. Machine Learning-Based Analysis of Program Binaries: A Comprehensive Study. *IEEE Access* 7 (2019), 65889–65912.