Semantic Positional Encoding for Better Vulnerability Detection.

## Introducing Juliet: a classification model based on encoder decoder neural networks fine-tuned on a NIST dataset of software vulnerabilities.

## Submitted for completion of the NASA WVSGC 2023-2024 Undergraduate Research Fellowship.

June 2024

## Andrew Gautier, Dale Dzielski, Thomas Devine

Lane Department of Computer Science and Electrical Engineering, West Virginia University

# Abstract

Software vulnerabilities are structural issues and programming flaws that can lead to compromised performance or inappropriate access to a malicious actor. Researchers have built tools to predict these weaknesses, called static analysis tools. Until recently, these tools have implemented detection algorithms by learning patterns in syntax that are known predictors of these faults, called code smells. A primary shortcoming of static analysis tools that we wish to address with this work is their lack of generalizability and tendency to overfit to specific threat domains. Our objective is applying sequence to sequence (Seq2Seq) deep learning architecture to train a vulnerability classification model. This approach intends to learn semantic positional features of these vulnerabilities, instead of relying on syntax comparison. Early results have been promising as proof of concept for these architectures. Our self-attention-based model, Juliet\_Attention has achieved an AUROC score (Area Under the Receiver Operating Characteristics) of 94.2% on our validation set.

# Introduction

The field of deep learning is an application of machine learning that draws inspiration from the structure of the brain. Deep learning algorithms work by mimicking the way parts of the brain handles signals to recognize patterns and relationships. These algorithms, better known as neural networks, are highly capable of learning on enormous datasets without overfitting. Neural networks differ based on the domain of the data they can process. For sequence learning tasks, recurrent neural networks (RNN) are the established architecture. RNN’s are an innovation over feedforward networks which only take inputs in one direction, and provide internal state representation, resembling a form of memory. A diagram of a state

Description automatically generated

Figure 1: LSTM Cell

RNN’s have many implementations but the most successful for language modeling is the Long short-term memory (LSTM) network [1]. LSTMs reduce RNN’s computational overhead by changing the way it processes large amounts of weights during training, commonly called the vanishing gradient problem. The LSTM cell (Figure 1) allows certain inputs from other states to be stored for a long series of time steps and forgotten or removed from calculation outside of this window. What’s remarkable about these networks is that the information forgotten will be relearned because of repeatable patterns in natural languages [2]. LSTM’s offer valuable encoding

A drawback even with the success of these new LSTM-based RNN’s is that positional encoding of words tended to be favored towards the end of the sentence, another appearance of the vanishing gradient problem. Neural machine translation (NMT) researchers addressed this with a modification for LSTM networks called Self-Attention (Attention hereafter) [3]. Attention allows RNNs to access weights from each part of a sentence directly, making it more aware of intra-sentence positional context. Not only did this make models better at Seq2Seq semantic encoding, but the weights of the sentence could be computed in parallel rather than one word at a time in the case of traditional RNNs. The innovation that brought us into the world of large language models (LLM) was the introduction of multi-head Attention mechanism, known as the Transformer [4]. Transformers removed the RNN component and vastly reduced required training time, making language models capable of crawling more data than ever before. Natural language processing (NLP) has advanced so rapidly, generative-pretrained-transformers like GPT-4 are capable of truly remarkable semantic and contextual learning and generation [5].

The field of NMT modeling outside of natural languages is a state-of-the-art level of deep learning research. Challenges for building these models include a lack of semantically parallel data to confirm translations, and extremely large or unstructured corpuses. Our interest lies in the Seq2Seq learning to semantically model programming languages. This led to the goal of developing a classifier model which can recognize vulnerabilities and common faults in programs by positional sequence encoding. Seq2Seq encoder decoder models have been built for tasks such as comment generation, source code summarization, and often use natural language prompts to revise or suggest changes within IDEs [6]. To the best of our knowledge, there have been no known attempts to build a multi-lingual code-based language with the task of classifying software vulnerabilities.

# Methodology

## Dataset

A close-up of a list

Description automatically generatedAs mentioned previously, a critical resource for training a NMT model is parallel data. To collect a wide range of test cases with known vulnerabilities we used the Juliet Test Suite C/C++ and Juliet Java test cases that were compiled by the NSA’s Center for Assured Software for evaluating static analysis tools. We retrieved these from a NIST reference dataset [7] and collected over 68,000 program files (43,344 from C/C++ and 25,344 Java files) which comprised 80 distinct CWE’s and entailed over 12 million lines of code. This curation will be referred to as the Juliet corpus. See *Figure 2* for the largest subgroups in the corpus.

Figure 2: CWE List

## Preprocessing

To prepare the raw source code to be passed into a learning model, we first needed to remove all comments from the code. The reasoning for doing this was twofold: comments would be highly noisy throughout the code and could lead to overfitting, and they are also in natural language, threatening our ability to evaluate the model’s ability to generalize between programming languages. We developed a tool for large corpus comment removal for several coding languages.

A graph of a line

Description automatically generated

Once we had comments removed, the next task was tokenizing and choosing how to batch our sequence data for a classification experiment. Initially we began experiments with creating our tokenizer by running the Sentencepeice [8] byte-pair-encoding (BPE) algorithm on the Juliet corpus. This was surpassed in performance, however, by the release of the aix-coder-7b-base tokenizer. This achieved a reported tokenization accuracy of 100% (Figure 3) across our entire corpus with extremely fast performance. It provided a vocabulary size of 49,152.

Figure 3: Aix-Coder-7b Tokenizer Results on Juliet Corpus

Inspired by Avelstein, we chose a context window of 40 sentences. Once the entire corpus was tokenized line by line, we created a sequence of 40 tokenized lines with a random location containing a vulnerability. Then one-hot encoded labels of each group were created for the training process of this our classifier model. These sequence batches along with their labels we referred to as vulnerability line windows (VLW.) Our VLW’s were padded to a length of 98 tokens based on the longest sentence in the corpus. We converted these into tensors using the PyTorch library so they could make use of GPU acceleration.

## Model Architecture

#### Embedding Layer

This layer maps input tokens to dense vectors, in our case using the weights obtained from the aix-coder-7b model [9]. This resulted in an embedding dimension of 4096. Our first pass of our model is a tensor of size 40x98x4096.

#### LSTM Layer

Our LSTM layer processes the embedded sequences bidirectionally, allowing the model to capture dependencies in both forward and backward directions. We used 256 LSTM nodes.

#### Mean Pooling / Attention Layer

A diagram of a complex function

Description automatically generatedWe trained two different models that had the same architecture except how they aggregate outputs from the LSTM layer. Our first model, Juliet\_Max\_Pool simply averages the LSTM outputs across the sequence length. Juliet\_Attention instead uses an Attention [3] calculation to compute weights for each output in the sentence (Figure 4). The goal of this side-by-side comparison is to see how applicable established natural language modeling techniques like Attention are relevant to the NMT field.

Figure 4: Self-Attention Calculation

#### Dropout Layer

To prevent overfitting, we used a drop out layer which randomly zeroes out input tensor elements, sampled randomly from a Bernoulli distribution.

#### Fully Connected Layer

The fully connected layer takes the transformed dropout layer and converts it to the desired output dimension, in our case, our model returns a tensor of 40x40, 40 sequences and there corresponding probability that it is apart of the vulnerable class we are trying to predict.

#### A mathematical equation with black text Description automatically generatedSigmoid Activation Layer

In this final layer, the outputs are scaled between a value of 0 and 1, which is desirable for our binary classification task. This is done using the common sigmoid activation function (Figure 5).

Figure 5: Sigmoid activation function

# Results

A black text on a white background

Description automatically generatedJuliet\_Max\_Pool and Juliet\_Attention was trained with the following parameters: A learning rate of 0.001, a loss function of binary cross entropy (BCE), and Adam [10] as an optimizer.

Figure 6: Binary-Cross-Entropy

A graph with a line

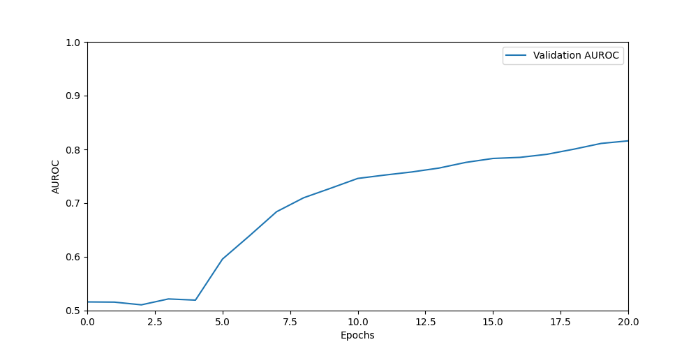
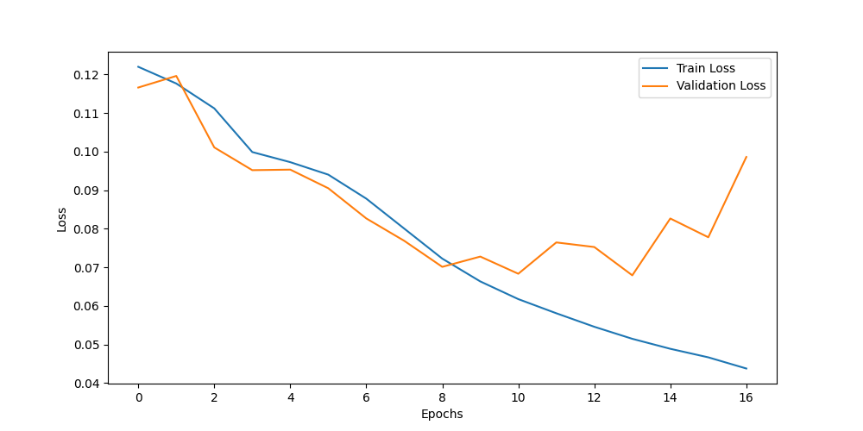
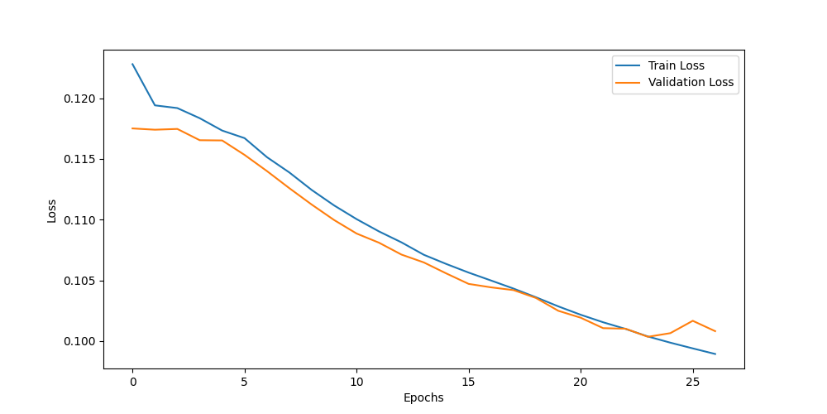
Description automatically generatedBoth models were evaluated using the AUROC metric. This score grades classifiers according to their ratio of true to false classifications. All models were trained until they returned three epochs without an improvement in validation loss. Juliet\_Max\_Pool achieved an AUROC score of 82.6% after training for 27 epochs and the Juliet\_Attention model ended with an AUROC score of 94.2% after 17 epochs.

Figure 8: Juliet\_Attention AUROC

Figure 9: Juliet\_Max\_Pool Loss

Figure 10: Juliet\_Max\_Pool AUROC

Figure 7: Juliet\_Attention Loss

# Findings & Future Work

The Juliet\_Attention model performed very well in this experiment. We are still in the early stages of evaluating this model for performance as a classifier. It should be noted that our dataset has yet to be fully stratified and we hope that upon doing so we can gain more insight about how Juliet performed on subsets of vulnerabilities. This is a valuable proof of concept that applying natural language modeling techniques to code can yield semantic context. Additionally, strong cross-language performance is a promising step towards greater generalizability for vulnerability detection and forecasting approaches.

In terms of architecture, the logical next step for this project would be to move beyond the RNN Seq2Seq network and leverage cross attention and transformer architecture [4] that NLP researchers have developed. This will allow further innovation for fine-tuning code-based models like we have done here. There remain many questions of the best ways to extract semantic features from code, and we hope others will continue this exploration with us.

This report, alongside its relevant code and sources can be found at <https://github.com/Andrew-Gautier/Juliet>.

# Acknowledgements

Computational resources were provided by the WVU Research Computing Dolly Sods HPC cluster, which is funded in part by NSF OAC-2117575. Thanks to the West Virginia Space Grant Consortium for funding this research.

Thanks to the aiXcoder-7b model [9] team, this was an enormous contribution to the field of NMT research and a timely resource for this project.

# References

[1] S. Hochreiter and J. Schmidhuber, “Long Short-Term Memory,” Neural Comput., vol. 9, no. 8, pp. 1735–1780, Nov. 1997, doi: 10.1162/neco.1997.9.8.1735.

[2] Y. Lakretz, G. Kruszewski, T. Desbordes, D. Hupkes, S. Dehaene, and M. Baroni, “The emergence of number and syntax units in LSTM language models,” 2019, doi: 10.18653/v1/N19-1002.

[3] J. Cheng, L. Dong, and M. Lapata, “Long Short-Term Memory-Networks for Machine Reading,” Sep. 20, 2016, arXiv: arXiv:1601.06733. doi: 10.48550/arXiv.1601.06733.

[4] A. Vaswani et al., “Attention Is All You Need,” Aug. 01, 2023, arXiv: arXiv:1706.03762. doi: 10.48550/arXiv.1706.03762.

[5] OpenAI et al., “GPT-4 Technical Report,” Mar. 04, 2024, arXiv: arXiv:2303.08774. doi: 10.48550/arXiv.2303.08774.

[6] I. Siroš, D. Singelée, and B. Preneel, “GitHub Copilot: the perfect Code compLeeter?,” Jun. 17, 2024, arXiv: arXiv:2406.11326. doi: 10.48550/arXiv.2406.11326.

[7] P. E. Black and others, “Sard: A software assurance reference dataset,” in Anonymous Cybersecurity Innovation Forum.(), 2017.

[8] T. Kudo and J. Richardson, “SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing,” Aug. 19, 2018, arXiv: arXiv:1808.06226. doi: 10.48550/arXiv.1808.06226.

[9] aiXcoder team. aixcoder-7b code large language model. https://github.com/aixcoder-plugin/aiXcoder-7B, 2024. Accessed: June 7, 2024

[10] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” Jan. 29, 2017, arXiv: arXiv:1412.6980. doi: 10.48550/arXiv.1412.6980.